The fundamentals of pandas

You've learned that Python has many open-source libraries and packages—including NumPy and pandas—that make it one of the most useful coding languages. In this reading, you will review the basics of pandas dataframes and learn more about how to work with them. Understanding the fundamentals of pandas is essential to becoming a capable and competent data professional.

Primary data structures

Pandas has two primary data structures: **`Series`** and **`DataFrame`**.

- **`Series`:** A Series is a one-dimensional labeled array that can hold any data type. It's similar to a column in a spreadsheet or a one-dimensional NumPy array. Each element in a series has an associated label called an index. The index allows for more efficient and intuitive data manipulation by making it easier to reference specific elements of your data.
- **`DataFrame`:** A dataframe is a two-dimensional labeled data structure—essentially a table or spreadsheet—where each column and row is represented by a Series.

Create a DataFrame

To use pandas in your notebook, first import it. Similar to NumPy, pandas has its own standard alias, **`pd`**, that's used by data professionals around the world:

```
1
```

```
import pandas as pd
```

Once you've imported pandas into your working environment, create a dataframe. Here are some of the ways to create a **`DataFrame`** object in a Jupyter Notebook.

From a dictionary:

```
1
2
3
```

```
d = {'col1': [1, 2], 'col2': [3, 4]}
df = pd.DataFrame(data=d)
df
```

RunReset

From a numpy array:

```
1
2
3
```

```
df2 = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
                   columns=['a', 'b', 'c'])
df2
```

RunReset

From a comma-separated values (csv) file:

(Note that this cell will not run, but is provided to illustrate syntax.)

```
1
df3 = pd.read_csv('/file_path/file_name.csv')
```

Attributes and methods

The **`DataFrame`** class is powerful and convenient because it comes with a suite of built-in features that simplify common data analysis tasks. These features are known as attributes and methods. An attribute is a value associated with an object or class that is referenced by name using dotted expressions. A method is a function that is defined inside a class body and typically performs an action. A simpler way of thinking about the distinction between attributes and methods is to remember that attributes are *characteristics* of the object, while methods are *actions* or *operations*.

Common `DataFrame` attributes

Data professionals use attributes and methods constantly. Some of the most-used **`DataFrame`** attributes include:

Attribute	Description
columns	Returns the column labels of the dataframe
dtypes	Returns the data types in the dataframe
iloc	Accesses a group of rows and columns using integer-based indexing
loc	Accesses a group of rows and columns by label(s) or a Boolean array
shape	Returns a tuple representing the dimensionality of the dataframe
values	Returns a NumPy representation of the dataframe

Common `DataFrame` methods

Some of the most-used **`DataFrame`** methods include:

Method	Description
apply()	Applies a function over an axis of the dataframe
copy()	Makes a copy of the dataframe's indices and data
describe()	Returns descriptive statistics of the dataframe, including the minimum, maximum, mean, and per values of its numeric columns; the row count; and the data types
drop()	Drops specified labels from rows or columns
groupby()	Splits the dataframe, applies a function, and combines the results
head(*n*=5)	Returns the first *n* rows of the dataframe (default=5)
info()	Returns a concise summary of the dataframe
isna()	Returns a same-sized Boolean dataframe indicating whether each value is null (can also use **`isnu`** as an alias)
sort_values()	Sorts by the values across a given axis

Method	Description
value_counts()	Returns a series containing counts of unique rows in the dataframe
where()	Replaces values in the dataframe where a given condition is false

These are just a handful of some of the most commonly used attributes and methods—there are many, many more! Some of them can also be used on pandas `Series` objects. For a more detailed list, refer to the pandas DataFrame documentation, which includes helpful examples of how to use each tool.

Selection statements

Once your data is read into a dataframe, you'll want to do things with it by selecting, manipulating, and evaluating the data. In this section, you'll learn how to select rows, columns, combinations of rows and columns, and basic subsets of data.

Row selection

Rows of a dataframe are selected by their index. The index can be referenced either by name or by numeric position.

`loc[]`

`loc[]` lets you select rows by name. Here's an example:

```
df = pd.DataFrame({
   'A': ['alpha', 'apple', 'arsenic', 'angel', 'android'],
   'B': [1, 2, 3, 4, 5],
   'C': ['coconut', 'curse', 'cassava', 'cuckoo', 'clarinet'],
   'D': [6, 7, 8, 9, 10]
   },
   index=['row_0', 'row_1', 'row_2', 'row_3', 'row_4'])
df
```

RunReset

The row index of the dataframe contains the names of the rows. Use `loc[]` to select rows by name:

```
print(df.loc['row_1'])
```

RunReset

Inserting just the row index name in selector brackets returns a **`Series`** object. Inserting the row index name as a list returns a **`DataFrame`** object:

```
print(df.loc[['row_1']])
```

RunReset

To select multiple rows by name, use a list within selector brackets:

```
print(df.loc[['row_2', 'row_4']])
```

RunReset

You can even specify a range of rows by named index:

```
print(df.loc['row_0':'row_3'])
```

RunReset

Note: Because you're using named indices, the returned range includes the specified end index.

`iloc[]`

`iloc[]` lets you select rows by numeric position, similar to how you would access elements of a list or an array. Here's an example.

```
print(df)
print()
print(df.iloc[1])
```

RunReset

Inserting just the row index number in selector brackets returns a **`Series`** object. Inserting the row index number as a list returns a **`DataFrame`** object:

```
print(df.iloc[[1]])
```

RunReset

To select multiple rows by index number, use a list within selector brackets:

```
print(df.iloc[[0, 2, 4]])
```

RunReset

Specify a range of rows by index number:

```
print(df.iloc[0:3])
```

RunReset

Note that this does not include the row at index three.

Column selection

Bracket notation

Column selection works the same way as row selection, but there are also some shortcuts to make the process easier. For example, to select an individual column, simply put it in selector brackets after the name of the dataframe:

```
print(df['C'])
```

RunReset

And to select multiple columns, use a list in selector brackets:

```
print(df[['A', 'C']])
```

RunReset

Dot notation

It's possible to select columns using dot notation instead of bracket notation. For example:

```
print(df.A)
```

RunReset

Dot notation is often convenient and easier to type. However, it can make your code more difficult to read, especially in longer statements involving method chaining or condition-based selection. For this reason, bracket notation is often preferred.

`loc[]`

You can also use `loc[]` notation:

```
print(df)
```

```
print()

print(df.loc[:, ['B', 'D']])
```

RunReset

Note that when using `loc[]` to select columns, you must specify rows as well. In this example, all rows were selected using just a colon (`:`).

`iloc[]`

Similarly, you can use `iloc[]` notation. Again, when using `iloc[]`, you must specify rows, even if you want to select all rows:

```
print(df.iloc[:, [1,3]])
```

RunReset

Select rows and columns

Both `loc[]` and `iloc[]` can be used to select specific rows and columns together.

`loc[]`

```
print(df.loc['row_0':'row_2', ['A','C']])
```

RunReset

Again, notice that when using `loc[]` to select a range, the final element in the range is included in the results.

`iloc[]`

```
print(df.iloc[[2, 4], 0:3])
```

RunReset

Note that, when using rows with named indices, you cannot mix numeric and named notation. For example, the following code will throw an error:

```
print(df.loc[0:3, ['D']])
```

RunReset

To view rows `[0:3]` at column '`D`' (if you don't know the index number of column D), you'd have to use selector brackets after an `iloc[]` statement:

```
# This is most convenient for VIEWING:
print(df.iloc[0:3][['D']])

# But this is best practice/more stable for assignment/manipulation:
print(df.loc[df.index[0:3], 'D'])
```

RunReset

However, in many (perhaps most) cases your rows will not have named indices, but rather numeric indices. In this case, you can mix numeric and named notation. For example, here's the same dataset, but with numeric indices instead of named indices.

```
df = pd.DataFrame({
   'A': ['alpha', 'apple', 'arsenic', 'angel', 'android'],
   'B': [1, 2, 3, 4, 5],
   'C': ['coconut', 'curse', 'cassava', 'cuckoo', 'clarinet'],
   'D': [6, 7, 8, 9, 10]
   },
   )
df
```

RunReset

Notice that the rows are enumerated now. Now, this code will execute without error:

```
print(df.loc[0:3, ['D']])
```

RunReset

Key takeaways

Pandas dataframes are a convenient way to work with tabular data. Each row and each column can be represented by a pandas **`Series`**, which is similar to a one-dimensional array. Both dataframes and series have a large collection of methods and attributes to perform common tasks and retrieve

information. Pandas also has its own special notation to select data. As you work more with pandas, you'll become more comfortable with this notation and its many applications in data science.