CSE 4510/5310
BIG DATA

Instructor: Fitzroy Nembhard, Ph.D.

Week 3-4

Pandas:
Python Data Analysis Library

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY®

Distribution

- All slides included in this class are for the exclusive use of students and instructors associated with Management and Processing of Big Data(CSE 4510/5310) at the Florida Institute of Technology
- Redistribution of the slides is not permitted without the written consent of the author.

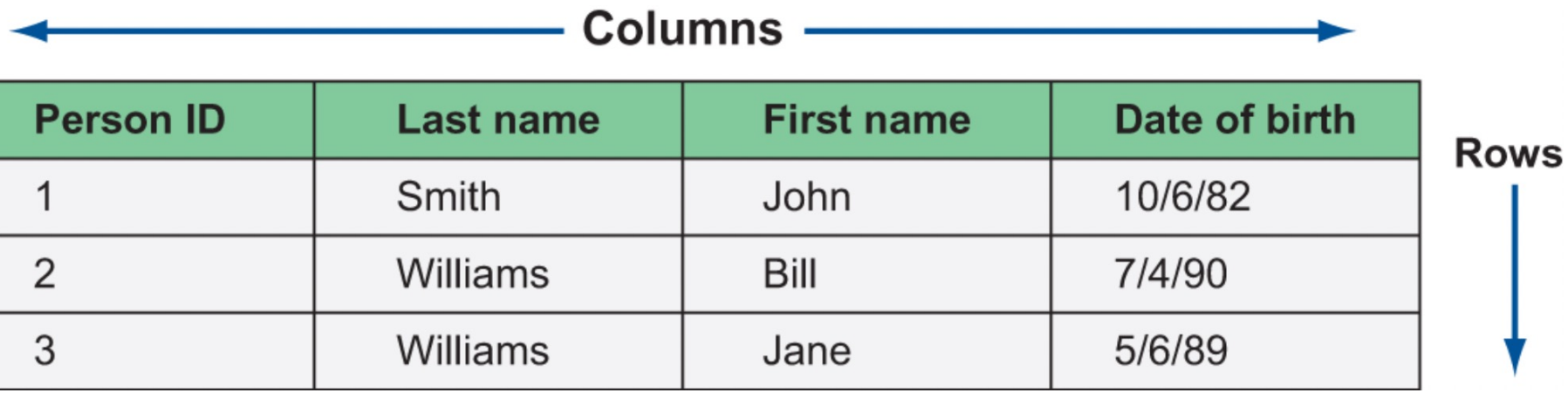
Goals

- To discuss and practice the basics of data processing and analysis tools (Pandas)
- To practice data processing/analysis concepts using real-world datasets

3.3 PANDAS DATA STRUCTURES

Structured Data

- Structured data is made up of rows and columns: from the humble spreadsheet to complex relational database systems, structured data is an intuitive way to store information



The Pandas Series

- **A Series** is a one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its index. The simplest Series is formed from only an array of data

```
>>> obj = pd.Series([4, 7, -5, 3])

# Results in the following:
0  4
1  7
2 -5
3  3
dtype: int64
```

The Pandas Series (Cont'd)

```
>>> obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

d  4
b  7
a -5
c  3

dtype: int64


>>> obj2['a']

# Results in the following:
-5
```

The Pandas Series (Cont'd)

- You can create a Series from a Python dictionary

```
>>> sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
>>> obj3 = pd.Series(sdata)

# Results in the following:
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
```

The Pandas Series (Cont'd)

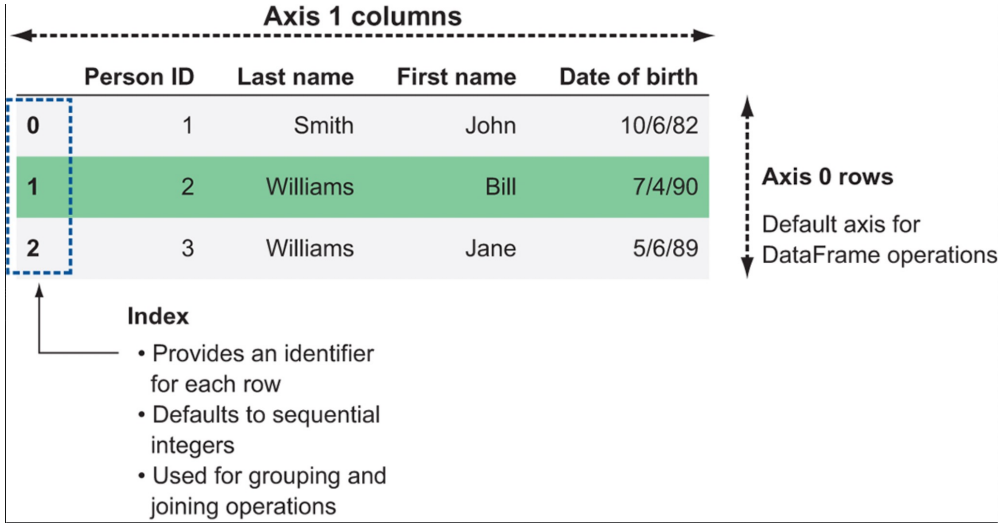
- The index in the resulting Series will have the dictionary's keys in sorted order
- You can override this by passing the dictionary keys in the order you want them to appear in the resulting Series.
- Since 'Utah' was not included in states, it is excluded from the resulting object

```
>>> sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
>>> states = ['California', 'Ohio', 'Oregon', 'Texas']
>>> obj3 = pd.Series(sdata, index=states)


# Results in the following:
California      NaN
Ohio        35000.0
Oregon      16000.0
Texas       71000.0
```

The Pandas DataFrame

- **DataFrame** represents a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.).
- The DataFrame has both a row and column index; it can be thought of as a dictionary of Series all sharing the same index

```
>>> data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
'year': [2000, 2001, 2002, 2001, 2002, 2003],
'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}

>>> df = pd.DataFrame(data)

# Results in the following:
   state  year  pop
0   Ohio  2000  1.5
1   Ohio  2001  1.7
2   Ohio  2002  3.6
3 Nevada  2001  2.4
4 Nevada  2002  2.9
5 Nevada  2003  3.2
```



The Pandas DataFrame (Cont'd)

- If you specify a sequence of columns, the DataFrame's columns will be arranged in that order

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
'year': [2000, 2001, 2002, 2001, 2002, 2003],
'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}

>>> pd.DataFrame(data, columns=['year', 'state', 'pop'])

# Results in the following:
   year   state  pop
0  2000   Ohio   1.5
1  2001   Ohio   1.7
2  2002   Ohio   3.6
3  2001 Nevada   2.4
4  2002 Nevada   2.9
5  2003 Nevada   3.2
```

The Pandas DataFrame (Cont'd)

- Possible data inputs to DataFrame constructor

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame; all sequences must be the same length
NumPy structured/record array	Treated as the "dict of arrays" case
dict of Series	Each value becomes a column; indexes from each Series are unioned together to form the result's row index if no explicit index is passed
dict of dicts	Each inner dict becomes a column; keys are unioned to form the row index as in the "dict of Series" case
List of dicts or Series	Each item becomes a row in the DataFrame; union of dict keys or Series indexes become the DataFrame's column labels
List of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed
NumPy MaskedArray	Like the "2D ndarray" case except masked values become NA/missing in the DataFrame result

The Pandas DataFrame (Cont'd)

- Some Index methods and properties

Method	Description
`append`	Concatenate with additional Index objects, producing a new Index
`difference`	Compute set difference as an Index
`intersection`	Compute set intersection
`union`	Compute set union
`isin`	Compute boolean array indicating whether each value is contained in the passed collection
`delete`	Compute new Index with element at index `i` deleted
`drop`	Compute new Index by deleting passed values
`insert`	Compute new Index by inserting element at index `i`
`is_monotonic`	Returns `True` if each element is greater than or equal to the previous element
`is_unique`	Returns `True` if the Index has no duplicate values
`unique`	Compute the array of unique values in the Index

The Pandas DataFrame (Cont'd)

- Indexing options with DataFrame

Type	Notes
`df[val]`	Select single column or sequence of columns from the DataFrame; special case conveniences: boolean array (filter rows), slice (slice rows), or boolean DataFrame (set values based on some criterion)
`df.loc[val]`	Selects single row or subset of rows from the DataFrame by label
`df.loc[:, val]`	Selects single column or subset of columns by label
`df.loc[val1, val2]`	Select both rows and columns by label
`df.iloc[where]`	Selects single row or subset of rows from the DataFrame by integer position
`df.iloc[:, where]`	Selects single column or subset of columns by integer position
`df.iloc[where_i, where_j]`	Select both rows and columns by integer position
`df.at[label_i, label_j]`	Select a single scalar value by row and column label
`df.iat[i, j]`	Select a single scalar value by row and column position (integers)
`reindex` method	Select either rows or columns by labels
`get_value, set_value` methods	Select single value by row and column label

Pandas Descriptive and Summary Statistics

- Descriptive and summary statistics

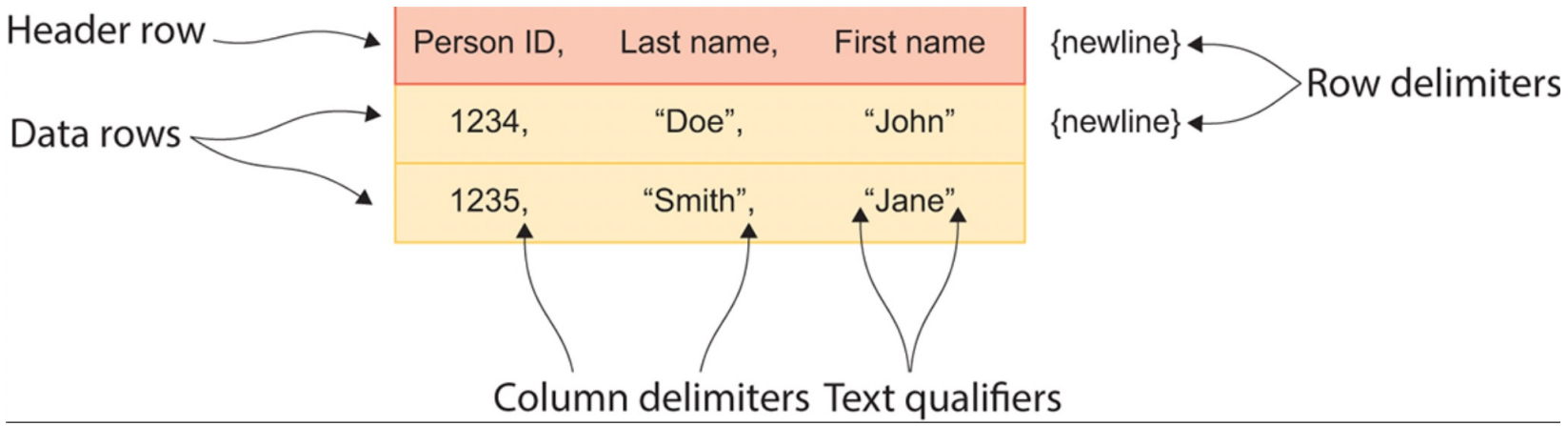
Method	Description
`count`	Number of non-NA values
`describe`	Compute set of summary statistics for Series or each DataFrame column
`min, max`	Compute minimum and maximum values
`argmin, argmax`	Compute index locations (integers) at which minimum or maximum value obtained, respectively
`idxmin, idxmax`	Compute index labels at which minimum or maximum value obtained, respectively
`quantile`	Compute sample quantile ranging from 0 to 1
`sum`	Sum of values
`mean`	Mean of values
`median`	Arithmetic median (50% quantile) of values
`mad`	Mean absolute deviation from mean value
`prod`	Product of all values
`var`	Sample variance of values
`std`	Sample standard deviation of values
`skew`	Sample skewness (third moment) of values

Parsing Functions in Pandas

Function	Description
`read_csv`	Load delimited data from a file, URL, or file-like object; use comma as default delimiter
`read_fwf`	Read data in fixed-width column format (i.e., no delimiters)
`read_clipboard`	Version of `read_csv` that reads data from the clipboard; useful for converting tables from web pages
`read_excel`	Read tabular data from an Excel XLS or XLSX file
`read_hdf`	Read HDF5 files written by pandas
`read_html`	Read all tables found in the given HTML document
`read_json`	Read data from a JSON (JavaScript Object Notation) string representation
`read_msgpack`	Read pandas data encoded using the MessagePack binary format
`read_pickle`	Read an arbitrary object stored in Python pickle format
`read_sas`	Read a SAS dataset stored in one of the SAS system's custom storage formats
`read_sql`	Read the results of a SQL query (using SQLAlchemy) as a pandas DataFrame
`read_stata`	Read a dataset from Stata file format
`read_feather`	Read the Feather binary file format
`read_sql_query`	Read SQL query into a DataFrame

Working With CSVs

- Every delimited text file format has two types of delimiters: row delimiters and column delimiters



Some *read_csv* Function Arguments in Pandas

Argument	Description
`path`	String indicating filesystem location, URL, or file-like object
`sep` or `delimiter`	Character sequence or regular expression to use to split fields in each row
`header`	Row number to use as column names; defaults to 0 (first row), but should be `None` if there is no header row
`index_col`	Column numbers or names to use as the row index in the result; can be a single name/number or a list of them for a hierarchical index
`names`	List of column names for result, combine with `header=None`
`skiprows`	Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip.
`na_values`	Sequence of values to replace with NA.
`comment`	Character(s) to split comments off the end of lines.
`parse_dates`	Attempt to parse data to `datetime`; `False` by default. If `True`, will attempt to parse all columns. Otherwise can specify a list of column numbers or name to parse. If element of list is tuple or list, will combine multiple columns together and parse to date (e.g., if date/time split across two columns).
`keep_date_col`	If joining columns to parse date, keep the joined columns; `False` by default.
`converters`	Dict containing column number of name mapping to functions (e.g., `{'foo': f}` would apply the function `f` to all values in the `'foo'` column).

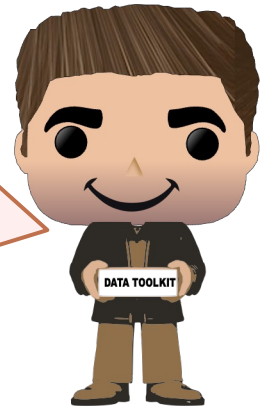
Some *read_csv* Function Arguments in Pandas (Cont'd)

Argument	Description
`dayfirst`	When parsing potentially ambiguous dates, treat as international format (e.g., 7/6/2012 -> June 7, 2012); `False` by default.
`date_parser`	Function to use to parse dates.
`nrows`	Number of rows to read from beginning of file.
`iterator`	Return a `TextParser` object for reading file piecemeal.
`chunksize`	For iteration, size of file chunks.
`skip_footer`	Number of lines to ignore at end of file.
`verbose`	Print various parser output information, like the number of missing values placed in non-numeric columns.
`encoding`	Text encoding for Unicode (e.g., `'utf-8'` for UTF-8 encoded text).
`squeeze`	If the parsed data only contains one column, return a Series.
`thousands`	Separator for thousands (e.g., `','` or `'.'`).

Some *read_csv* Examples

```
# You may parse a column(s) (eg. a column named 'TIME') as date
# while reading a CSV file.
# You may also choose the date_parser. Here we use a lambda function
pd.read_csv('my_dataset.csv',
                  parse_dates=['TIME'],
                  date_parser=lambda x: pd.to_datetime(x, format='%m/%d/%Y %I:%M:%S %p'))

# You may skip a certain number of rows such as header and notes rows
pd.read_csv('my_dataset.csv', skiprows=3)

# To allow reading a reasonably-sized file
pd.read_csv('my_dataset.csv', low_memory=False)

# Skip bad lines while reading a CSV
pd.read_csv('my_dataset.csv', error_bad_lines=False)

# Show a warning if bad lines encountered when reading a CSV
pd.read_csv('my_dataset.csv', warn_bad_lines=True)
```

Recall that a lambda function is an inline anonymous function. It can take any number of arguments, but can only have one expression



Time Series in Pandas

- For ***time stamps***, Pandas provides the ***Timestamp*** type - a replacement for Python's native datetime, but is based on the more efficient ***numpy.datetime64***
- For ***time periods***, Pandas provides the ***Period*** type – also based on ***numpy.datetime64***
- For ***time deltas*** or durations, Pandas provides the ***Timedelta*** type - replacement for Python's native datetime.timedelta type, and is based on **numpy.timedelta64**

```
>>> dates = pd.to_datetime([datetime(2015, 7, 3), '4th of July, 2015', '2015-Jul-6',
'07-07-2015', '20150708'])

# Results in the following:
 DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07', '2015-07-08'],
                              dtype='datetime64[ns]', freq=None)
```

Time Series in Pandas (Cont'd)

- Any **DatetimeIndex** can be converted to a **PeriodIndex** with the to_period() function with the addition of a frequency code; here we'll use 'D' to indicate daily frequency

```
>>> dates = pd.to_datetime([datetime(2015, 7, 3), '4th
of July, 2015', '2015-Jul-6', '07-07-2015',
'20150708'])
>>> dates.to_period('D')

# Results in the following:
PeriodIndex(['2015-07-03', '2015-07-04', '2015-07-06',
'2015-07-07', '2015-07-08'],
dtype='int64', freq='D')
```

Pandas frequency codes

Code	Description	Code	Description
D	Calendar day	B	Business day
W	Weekly		
M	Month end	BM	Business month end
Q	Quarter end	BQ	Business quarter end
A	Year end	BA	Business year end
H	Hours	BH	Business hours
T	Minutes		
S	Seconds		
L	Milliseonds		
U	Microseconds		
N	Nanoseconds		

Time Series in Pandas (Cont'd)

- A **TimedeltaIndex** can be created when one date is subtracted from another

```
>>> dates = pd.to_datetime([datetime(2015, 7, 3), '4th of
July, 2015', '2015-Jul-6', '07-07-2015', '20150708'])
>>> dates - dates[0]

# Results in the following:
TimedeltaIndex(['0 days', '1 days', '3 days', '4 days', '5
days'], dtype='timedelta64[ns]', freq=None)
```

Time Series in Pandas (Cont'd)

- **pd.date_range()** accepts a start date, an end date, and an optional frequency code to create a regular sequence of dates.
- By default, the frequency is one day.

```
>>> pd.date_range('2020-03-28', '2020-04-01')

# Results in the following:
DatetimeIndex(['2020-03-28', '2020-03-29', '2020-03-30', '2020-03-31',
               '2020-04-01'],
              dtype='datetime64[ns]', freq='D')
```

Time Series in Pandas (Cont'd)

- The date range can be specified not only with a start- and endpoint, but with a startpoint and a number of periods.

```
>>> dates = pd.date_range('2020-04-01', periods=10)
>>> dates
# Results in the following:
DatetimeIndex(['2020-04-01', '2020-04-02', '2020-04-03', '2020-04-04',
               '2020-04-05', '2020-04-06', '2020-04-07', '2020-04-08',
               '2020-04-09', '2020-04-10'],
              dtype='datetime64[ns]', freq='D')


>>> dates[0]
# Results in the following:
Timestamp('2020-04-01 00:00:00', freq='D')

>>> dates[0].strftime('%B %d, %Y')
# Results in the following:
'April 01, 2020'
```

Resampling Time Series Data

- You may resample a DataFrame of time series data according to a frequency code. See frequency code examples on Slide 65. In this example, daily data is being resampled to weekly data.

```
data={"dates": pd.date_range('2020-11-1', '2020-11-30'),
      "values": np.linspace(0, 29, 30)}

df = pd.DataFrame(data)

df = df.set_index("dates")

my_dates2 = df.resample('W').last()

# Results in the following >>
```

Other use cases:

```
resample('d').sum()
resample('BM').mean()
resample('d', how='sum')
You may also resample a Series like this: my_series.asfreq('BA')
```

dates	values
2020-11-01	0.0
2020-11-02	1.0
2020-11-03	2.0
2020-11-04	3.0
2020-11-05	4.0
...	
2020-11-28	27.0
2020-11-29	28.0
2020-11-30	29.0

dates	values
2020-11-01	0.0
2020-11-08	7.0
2020-11-15	14.0
2020-11-22	21.0
2020-11-29	28.0
2020-12-06	29.0

3.4 PRACTICING PANDAS

1. Practicing Pandas

- **Example: Creating a Dataframe and Answering Questions**
- Create a Dataframe using the following data

```
data = { 'Animal': ['cat', 'cat', 'snake', 'dog', 'dog', 'cat', 'snake', 'cat', 'dog', 'dog'],
'Age': [2.5, 3, 0.5, np.nan, 5, 2, 4.5, np.nan, 7, 3],
'Visits': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
'Priority': ['yes', 'yes', 'no', 'yes', 'no', 'no', 'no', 'yes', 'no', 'no']}
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

# Expected output
  Animal  Age Priority  Visits
a    cat  2.5      yes       1
b    cat  3.0      yes       3
c  snake  0.5       no       2
d    dog  NaN      yes       3
e    dog  5.0       no       2
f    cat  2.0       no       3
g  snake  4.5       no       1
h    cat  NaN      yes       1
i    dog  7.0       no       2
j    dog  3.0       no       1
```

1. Practicing Pandas (Cont'd)

- **Example: Creating a Dataframe and Answering Questions**
- Display a summary of the Dataframe's basic information (df.info())
- Return the first three rows of the Dataframe df
- Select just the animal and age columns from the Dataframe df
- Count the visit priority per animal
- Find the mean of the animals' ages
- Display a statistical summary of the dataset

2. Practicing Pandas

- **Example: Assigning an Index to a DataFrame**
- Given the following DataFrame:

```
df = pd.DataFrame({'month': [1, 4, 7, 10],
'year': [2012, 2014, 2013, 2014], 'sale': [55, 40, 84, 31]})
```

- By default, the indices are counting numbers starting at 0.
- We may assign a new index as follows:

```
df.set_index('month')
```

3. Practicing Pandas

- **Example: Re-Indexing a DataFrame**
- Given the following DataFrame containing Tesla sales for Q1 2020:

```
date_index = pd.date_range('1/1/2020', periods=3, freq='M')

df = pd.DataFrame({"tesla_sales": [8000, 7275, 14625]},
index=date_index)
```

	tesla_sales
2020-01-31	8000
2020-02-29	7275
2020-03-31	14625

- **We may assign a new index and use the reindex method as follows.**
- This method can be used to fill in data if it introduces new records
- date_index2 = pd.date_range('1/1/2020', periods=6, freq='M')

```
df.reindex(date_index2)
```

	tesla_sales
2020-01-31	8000.0
2020-02-29	7275.0
2020-03-31	14625.0
2020-04-30	NaN
2020-05-31	NaN
2020-06-30	NaN

* If you do want to fill in the NaN values present in the original DataFrame, use the fillna() method

The End