Chapter 2

The Pandas essentials for data analysis

Objectives (part 1)

Applied

1. Create a DataFrame by reading data from a file or by using the DataFrame constructor.
2. Save a DataFrame to disk as a pickle file, and restore the DataFrame by reading the pickle file.
3. Examine the data in a DataFrame by displaying the data and its attributes.
4. Examine the data in a DataFrame by using the info(), nunique(), and describe() methods.
5. Access columns, rows, or a subset of columns and rows by using some combination of dot notation, brackets, the query() method, and the loc[] or iloc[] accessor.
6. Use Pandas methods to get statistics for the columns of a DataFrame.

Objectives (part 2)

7. Use Python to do calculations on the data in the columns of a DataFrame.
8. Use the Pandas or Python replace() method to replace the data in a DataFrame or Series object.
9. Use Pandas methods to do the following DataFrame operations:

   Sort the rows

   Set and reset an index

   Pivot the data

   Melt the data

   Group and aggregate the data

   Plot the data based on the index that has been set

Objectives (part 3)

Knowledge

1. Distinguish between a Series and a DataFrame.
2. Describe these attributes of a DataFrame: values, index, columns, shape.

The first five rows of the Child Mortality data in a DataFrame

	Year	Age Group	Death Rate
0	1900	1-4 Years	1983.8
1	1901	1-4 Years	1695.0
2	1902	1-4 Years	1655.7
3	1903	1-4 Years	1542.1
4	1904	1-4 Years	1591.5

The components of a DataFrame

Component	Description
Column labels	The names at the tops of the columns.
Column data	The data in the columns. All of the data in a column typically has the same data type with one entry in each row.
Column data types	Each column has a defined data type. If all of the elements in a column don't have the same data type, the elements are stored with the object data type.
Index	Also known as a row label. If an index isn't defined, it is generated as a sequence of integers starting with zero.
Metadata	Attributes of the DataFrame that are generated by Pandas when the DataFrame is constructed or changed.

Terms related to DataFrame objects

- DataFrame
- Column
- Row
- Series object
- Element
- Datapoint

Three of the Pandas read() methods for importing data into a DataFrame

```
read_csv()
read_excel()
read_sql_query()
```

How to import a CSV file from a website

```
import pandas as pd
url = "https://data.cdc.gov/api/views/v6ab-adf5/" +
      "rows.csv?accessType=DOWNLOAD"
mortality_data = pd.read_csv(url)
```

The DataFrame() constructor

```
DataFrame(params)
```

Parameters

data

columns

index

The data and columns arrays for a DataFrame

```
df_data=[[1900, '1-4 Years', 1983.8],[1901, '1-4 Years', 1695.0]]
df_columns=['Year', 'Age Group', 'Death Rate']
```

The code that creates the DataFrame

```
import pandas as pd
mortality_df = pd.DataFrame(
    data=df_data,
    columns=df_columns)
```

The DataFrame that's created

	Year	Age Group	Death Rate
0	1900	1-4 Years	1983.8
1	1901	1-4 Years	1695.0

Three of the Pandas methods for saving a DataFrame to disk

```
to_pickle()
to_csv()
to_excel()
```

How to save a DataFrame as a pickle file

```
mortality_data.to_pickle('mortality_data.pkl')
```

The Pandas read_pickle() method

```
read_pickle(filename)
```

How to read a pickle file to restore a DataFrame

```
mortality_data = pd.read_pickle('mortality_data.pkl')
```

The DataFrame that's restored

	Year	Age Group	Death Rate
0	1900	1-4 Years	1983.8
1	1901	1-4 Years	1695.0
2	1902	1-4 Years	1655.7
3	1903	1-4 Years	1542.1
4	1904	1-4 Years	1591.5

Two methods for displaying the data in a DataFrame

```
head(rows)
tail(rows)
```

JupyterLab automatically displays the contents of a named DataFrame

```
mortality_data
```

	Year	Age Group	Death Rate
0	1900	1-4 Years	1983.8
1	1901	1-4 Years	1695.0
2	1902	1-4 Years	1655.7
3	1903	1-4 Years	1542.1
4	1904	1-4 Years	1591.5
...	...	...	...
471	2014	15-19 Years	45.5
472	2015	15-19 Years	48.3
473	2016	15-19 Years	51.2
474	2017	15-19 Years	51.5
475	2018	15-19 Years	49.2

476 rows × 3 columns

How to use the head() and tail() methods

```
mortality_data.head()        # displays the first 5 rows
mortality_data.tail(3)       # displays the last 3 rows
```

How to display the data in 5 rows and all columns

```
with pd.option_context(
        'display.max_rows', 5,
        'display.max_columns', None):
    display(mortality_data)
```

Some of the attributes of a DataFrame object

Attribute	Description
`values`	The values of the DataFrame in an array format
`index`	The row index
`columns`	The column names
`size`	The total number of elements
`shape`	The number of rows and columns

The values attribute

```
mortality_data.values
================================================
array([[1900, '1-4 Years', 1983.8],
       [1901, '1-4 Years', 1695.0],
       [1902, '1-4 Years', 1655.7],
       ...,
       [2016, '15-19 Years', 51.2],
       [2017, '15-19 Years', 51.5],
       [2018, '15-19 Years', 49.2]], dtype=object)
```

The other four attributes

```
print("Index:  ", mortality_data.index)
print("Columns:", mortality_data.columns)
print("Size:   ", mortality_data.size)
print("Shape:  ", mortality_data.shape)
================================================================
Index:   RangeIndex(start=0, stop=476, step=1)
Columns: Index(['Year', 'Age Group', 'Death Rate'], dtype='object')
Size:    1428
Shape:   (476, 3)
```

How to use the columns attribute to replace spaces with nothing

```
mortality_data.columns =
    mortality_data.columns.str.replace(' ', '')
```

The new column names

```
Index(['Year', 'AgeGroup', 'DeathRate'], dtype='object')
```

The info(), nunique(), and describe() methods

Method	Description
`info(params)`	Returns information about the DataFrame and its columns.
`nunique()`	Returns the number of unique data items in each column.
`describe()`	Returns statistical information for each numeric column.

How to use the info() method

```
mortality_data.info()
=====================================
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 476 entries, 0 to 475
Data columns (total 3 columns):
Year          476 non-null int64
AgeGroup      476 non-null object
DeathRate     476 non-null float64
dtypes: float64(1), int64(1), object(1)
memory usage: 11.3+ KB
```

The memory_usage parameter ensures a more accurate usage result

```
mortality_data.info(memory_usage='deep')
=========================================
<class 'pandas.core.frame.DataFrame'>
...
memory usage: 38.7 KB
```

How to use the nunique() method

```
mortality_data.nunique()
========================
Year           119
AgeGroup         4
DeathRate      430
dtype: int64
```

How to use the describe() method

With the T property, the statistics are displayed in the columns

```
mortality_data.describe().T
```

	count	mean	std	min	25%	50%	75%	max
Year	476.0	1959.00000	34.387268	1900.0	1929.000	1959.0	1989.000	2018.0
DeathRate	476.0	192.92416	293.224216	11.4	40.575	89.5	222.575	1983.8

Without the T property, the statistics are displayed in the rows

```
mortality_data.describe()
```

The syntax for accessing columns

To access	With brackets	With dot notation
One column	`df[column_name]` `df[[column_name]]`	`df.column_name`
Two or more columns	`df[[col1,col2,...]]`	

Two ways to access one column

With dot notation

```
mortality_data.DeathRate.head(2)
================================
0    1983.8
1    1695.0
Name: DeathRate, dtype: float64
```

With brackets

```
mortality_data['DeathRate'].head(2)
===================================
0    1983.8
1    1695.0
Name: DeathRate, dtype: float64
```

How to use a list to access two or more columns

```
mortality_data[['Year','DeathRate']].head(2)
```

	Year	DeathRate
0	1900	1983.8
1	1901	1695.0

The query() method

```
query(condition)
```

How to use the query() method to access rows (part 1)

Based on a single column

```
mortality_data.query('Year == 1900')
```

	Year	AgeGroup	DeathRate
0	1900	1-4 Years	1983.8
119	1900	5-9 Years	466.1
238	1900	10-14 Years	298.3
357	1900	15-19 Years	484.8

How to use the query() method to access rows (part 2)

Based on multiple columns with the *and* operator

```
mortality_data.query('Year == 2000 and AgeGroup != "1-4 Years"')
```

	Year	AgeGroup	DeathRate
219	2000	5-9 Years	15.8
338	2000	10-14 Years	20.3
457	2000	15-19 Years	67.1

How to use the query() method to access rows (part 3)

Based on multiple columns with the *or* operator

```
mortality_data.query('Year == 1900 or Year == 2000').head()
```

	Year	AgeGroup	DeathRate
0	1900	1-4 Years	1983.8
100	2000	1-4 Years	32.4
119	1900	5-9 Years	466.1
219	2000	5-9 Years	15.8
238	1900	10-14 Years	298.3

With backticks for column names that contain spaces

```
mortality_data.query('Year == 2000 and `Age Group` != "1-4 Years"')
```

How to access one column from a subset of rows using dot notation

```
mortality_data.query('Year == 1900').DeathRate
==============================================
0      1983.8
119     466.1
238     298.3
357     484.8
Name: DeathRate, dtype: float64
```

How to access one column from a subset of rows using brackets

```
mortality_data.query('Year == 1900')['DeathRate']
================================================
0      1983.8
119     466.1
238     298.3
357     484.8
Name: DeathRate, dtype: float64
```

How to access one column from a subset of rows using a list

```
mortality_data.query('Year == 1900')[['DeathRate']]
```

	DeathRate
0	1983.8
119	466.1
238	298.3
357	484.8

How to access two or more columns from a subset of rows using a list

```
mortality_data.query('Year == 1900')[['AgeGroup','DeathRate']]
```

	AgeGroup	DeathRate
0	1-4 Years	1983.8
119	5-9 Years	466.1
238	10-14 Years	298.3
357	15-19 Years	484.8

The loc[ ] and iloc[ ] accessors for accessing rows and columns

Accessor	Accesses rows and columns
`loc[rows,columns]`	by their labels.
`iloc[rows,columns]`	by their positions.

How to access rows with the loc[ ] accessor

With a list to access the rows with labels 0, 5, and 10

```
mortality_data.loc[[0,5,10]]
```

With a slice to access the rows with labels 4 through 6

```
mortality_data.loc[4:6]
```

With a slice to access every 5th row from 0 through 20

```
mortality_data.loc[0:20:5]
```

With a conditional expression to access the rows for the year 1917

```
mortality_data.loc[mortality_data.Year == 1917]
```

How to access rows and columns with the loc[ ] accessor

With lists of row and column labels

```
mortality_data.loc[[0,5,10],['AgeGroup','DeathRate']]
```

With slices of row and column labels

```
mortality_data.loc[4:6,'AgeGroup':'DeathRate']
```

How to access rows and columns with the iloc[ ] accessor

With lists of row and column positions

```
mortality_data.iloc[[4,5,6],[1,2]]
```

With slices of row and column positions

```
mortality_data.iloc[4:7,1:3]
```

With a negative row position in a slice to access the last 10 rows

```
mortality_data.iloc[-10:]
```

The sort_values() method

```
sort_values(columns, ascending)
```

How to sort by one column in descending order

```
mortality_data.sort_values('DeathRate',
                           ascending=False).head(3)
```

	Year	AgeGroup	DeathRate
0	1900	1-4 Years	1983.8
1	1901	1-4 Years	1695.0
2	1902	1-4 Years	1655.7

How to sort by multiple columns in ascending order

```
mortality_data.sort_values(['Year','DeathRate']).head(3)
```

	Year	AgeGroup	DeathRate
238	1900	10-14 Years	298.3
119	1900	5-9 Years	466.1
357	1900	15-19 Years	484.8

How to sort by multiple columns in mixed orders

```
mortality_data.sort_values(['Year','DeathRate'],
                           ascending=[True,False]).head()
```

	Year	AgeGroup	DeathRate
0	1900	1-4 Years	1983.8
357	1900	15-19 Years	484.8
119	1900	5-9 Years	466.1
238	1900	10-14 Years	298.3
1	1901	1-4 Years	1695.0

Some of the Pandas methods for both Series and DataFrame objects

```
count()
mean()
median()
min()
max()
std()
sum()
cumsum()
quantile(q)
```

How to apply a method to one column

```
mortality_data.DeathRate.mean()
===============================
192.92415966386568
```

How to apply a method to two columns

```
mortality_data[['AgeGroup','DeathRate']].max()
==============================================
AgeGroup     5-9 Years
DeathRate       1983.8
dtype: object
```

How to apply a method to all columns

```
mortality_data.count()
======================
Year          476
AgeGroup      476
DeathRate     476
dtype: int64
```

How to apply the quantile() method to two different quantiles

```
mortality_data.quantile([.1,.9])
```

	Year	DeathRate
0.1	1911.5	21.50
0.9	2006.5	430.85

The Python operators for column arithmetic

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Integer division
%	Modulo/Remainder
**	Exponentiation

How to add a column to a DataFrame

```
mortality_data['MeanCentered'] = \
    mortality_data.DeathRate - mortality_data.DeathRate.mean()
```

	Year	AgeGroup	DeathRate	MeanCentered
0	1900	1-4 Years	1983.8	1790.87584
1	1901	1-4 Years	1695.0	1502.07584
2	1902	1-4 Years	1655.7	1462.77584
3	1903	1-4 Years	1542.1	1349.17584

How to modify the data in an existing column

```
mortality_data['DeathRate'] = \
    mortality_data.DeathRate / 100000
```

	Year	AgeGroup	DeathRate	MeanCentered
0	1900	01-04 Years	0.019838	1790.87584
1	1901	01-04 Years	0.016950	1502.07584
2	1902	01-04 Years	0.016557	1462.77584
3	1903	01-04 Years	0.015421	1349.17584

The Pandas replace() method

```
replace(to_replace,value, inplace)
```

The Python replace() method

```
replace(old,new)
```

How to modify the string data in a column

With the Pandas replace() method with three parameters

```
mortality_data.AgeGroup.replace(
    to_replace = ['1-4 Years','5-9 Years'],
    value = ['01-04 Years','05-09 Years'],
    inplace = True)
```

With the Pandas replace() method and a dictionary of old and new values

```
mortality_data.AgeGroup.replace(
    {'1-4 Years':'01-04 Years','5-9 Years':'05-09 Years'},
    inplace = True)
```

With the Python replace() method

```
mortality_data['AgeGroup'] =
    mortality_data.AgeGroup.str.replace('1-4 Years', '01-04 Years')
mortality_data['AgeGroup'] =
    mortality_data.AgeGroup.str.replace('5-9 Years', '05-09 Years')
```

The result for all three of the examples

	Year	AgeGroup	DeathRate	MeanCentered
0	1900	01-04 Years	0.019838	1790.87584
1	1901	01-04 Years	0.016950	1502.07584
2	1902	01-04 Years	0.016557	1462.77584
3	1903	01-04 Years	0.015421	1349.17584

The set_index() method

```
set_index(columns, verify_integrity)
```

How to set a one-column index

```
mortality_data = mortality_data.set_index('Year')
mortality_data.head(2)
```

Year	AgeGroup	DeathRate	MeanCentered
1900	01-04 Years	0.019838	1790.87584
1901	01-04 Years	0.016950	1502.07584

How to verify the integrity of a one-column index

```
mortality_data = mortality_data.set_index('Year', verify_integrity=True)
mortality_data.head(2)
```

```
==========================================================================
ValueError: Index has duplicate keys: Int64Index([1900, 1901, ....
```

How to set a two-column index

```
mortality_data.set_index(['Year','AgeGroup'],
verify_integrity=True)
mortality_data.head(2)
```

		DeathRate	MeanCentered
Year	AgeGroup		
1900	01-04 Years	0.019838	1790.87584
1901	01-04 Years	0.016950	1502.07584

The reset_index() method

```
reset_index(inplace)
```

How to reset an index

```
mortality_data.reset_index(inplace=True)
mortality_data.head(2)
```

	Year	AgeGroup	DeathRate	MeanCentered
0	1900	01-04 Years	0.019838	1790.87584
1	1901	01-04 Years	0.016950	1502.07584

The pivot() method of a DataFrame

```
pivot(index,columns,values)
```

How to pivot the AgeGroup column for the death rate

```
mortality_wide = mortality_data.pivot(
    index='Year', columns='AgeGroup', values='DeathRate')
mortality_wide.head(3)
```

AgeGroup	01-04 Years	05-09 Years	10-14 Years	15-19 Years
Year				
1900	0.019838	0.004661	0.002983	0.004848
1901	0.016950	0.004276	0.002736	0.004544
1902	0.016557	0.004033	0.002525	0.004215

How to pivot the AgeGroup column for all other data

```
mortality_wide = mortality_data.pivot(
    index='Year', columns='AgeGroup')
mortality_wide.head(3)
```

	DeathRate				MeanCentered			
AgeGroup	01-04 Years	05-09 Years	10-14 Years	15-19 Years	01-04 Years	05-09 Years	10-14 Years	15-19 Years
Year								
1900	0.019838	0.004661	0.002983	0.004848	1790.87584	273.17584	105.37584	291.87584
1901	0.016950	0.004276	0.002736	0.004544	1502.07584	234.67584	80.67584	261.47584
1902	0.016557	0.004033	0.002525	0.004215	1462.77584	210.37584	59.57584	228.57584

The melt() method of a DataFrame

Method	Description
`melt(params)`	Melts the data in two or more columns into a single column.

Parameters of the melt() method

Parameters	Description
`id_vars`	The column or columns that identify each row.
`value_vars`	The columns to melt. If none are specified, all other columns will be melted.
`var_name`	The name of the column that will contain the melted column names, or “variable” by default.
`value_name`	The name of the column that will contain the values, or “value” by default.

The starting DataFrame in wide form

	Year	01-04 Years	05-09 Years	10-14 Years	15-19 Years
0	1900	0.019838	0.004661	0.002983	0.004848
1	1901	0.016950	0.004276	0.002736	0.004544
2	1902	0.016557	0.004033	0.002525	0.004215
3	1903	0.015421	0.004147	0.002682	0.004341

How to melt the data for just the 01-04 and 05-09 columns

```
mortality_long = mortality_wide.melt(
    id_vars = 'Year',
    value_vars=['01-04 Years','05-09 Years'],
    var_name ='AgeGroup',
    value_name='DeathRate')
```

	Year	AgeGroup	DeathRate
0	1900	01-04 Years	0.019838
1	1901	01-04 Years	0.016950
...	...	...	...
236	2017	05-09 Years	0.000116
237	2018	05-09 Years	0.000115

The groupby() method

```
groupby(columns)
```

How to group the data on the AgeGroup column

```
mortality_data.groupby('AgeGroup').mean().head(4)
```

AgeGroup	Year	DeathRate	MeanCentered
01-04 Years	1959	0.003832	190.301891
05-09 Years	1959	0.001173	-75.598109
10-14 Years	1959	0.000938	-99.154412
15-19 Years	1959	0.001774	-15.549370

How to group the data on the Year column

```
mortality_data.groupby('Year').median().head(4)
```

Year	DeathRate	MeanCentered
1900	0.004755	282.52584
1901	0.004410	248.07584
1902	0.004124	219.47584
1903	0.004244	231.47584

How to group the data on multiple columns

```
mortality_data.groupby(['Year','AgeGroup']).count().head()
```

Year	AgeGroup	DeathRate	MeanCentered
1900	01-04 Years	1	1
	05-09 Years	1	1
	10-14 Years	1	1
	15-19 Years	1	1
1901	01-04 Years	1	1

The agg() method

```
agg(methods)
```

How to aggregate the data for all columns in each age group

```
mortality_data.groupby('AgeGroup').agg(['mean','median'])
```

	Year		DeathRate		MeanCentered	
	mean	median	mean	median	mean	median
AgeGroup						
01-04 Years	1959	1959	0.003832	0.001091	190.301891	-83.82416
05-09 Years	1959	1959	0.001173	0.000484	-75.598109	-144.52416
10-14 Years	1959	1959	0.000938	0.000446	-99.154412	-148.32416
15-19 Years	1959	1959	0.001774	0.001069	-15.549370	-86.02416

How to aggregate the data for just the death rate in each age group

```
mortality_data.groupby('AgeGroup')['DeathRate'] \
    .agg(['mean','median','std','nunique'])
```

AgeGroup	mean	median	std	nunique
01-04 Years	0.003832	0.001091	0.005005	117
05-09 Years	0.001173	0.000484	0.001275	115
10-14 Years	0.000938	0.000446	0.000884	115
15-19 Years	0.001774	0.001069	0.001384	117

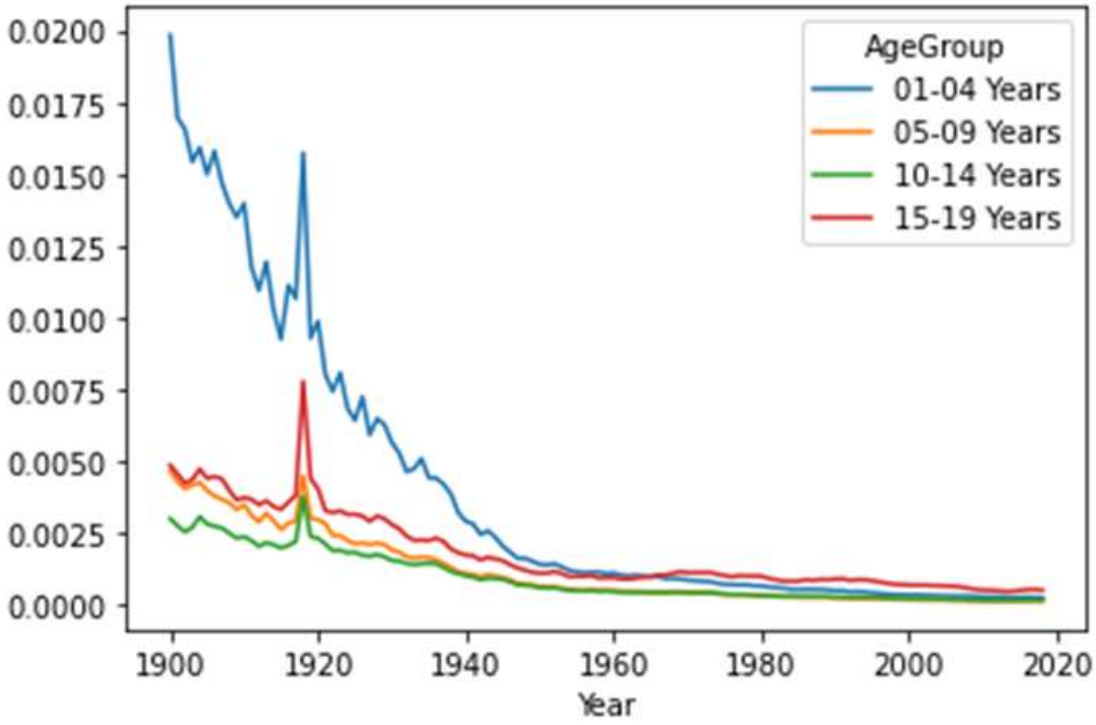
How to aggregate the data for just the death rate in each year group

```
mortality_data.groupby('Year')['DeathRate'] \
    .agg(['mean','median','std','min','max','var','nunique'])
```

Year	mean	median	std	min	max	var	nunique
1900	0.008082	0.004755	0.007882	0.002983	0.019838	6.212178e-05	4
1901	0.007127	0.004410	0.006597	0.002736	0.016950	4.352410e-05	4
1902	0.006832	0.004124	0.006527	0.002525	0.016557	4.260299e-05	4

How to chain the pivot() and plot() methods

```
mortality_data.pivot(index='Year',
columns='AgeGroup')['DeathRate'].plot()
```



How to chain the groupby(), agg(), and plot() methods

```
mortality_data.groupby('AgeGroup')['DeathRate'] \
    .agg(['mean','median','std']).plot.barh()
```