Explore the different object attributes of the dataframe

- Pandas is the high-powered Python library used for data analysis.
  - It is built on top of NumPy and is capable of handling large datasets efficiently.
  - Pandas is also used for data visualization as it can plot data in a tabular or graphical form.
- It will be the basis of GeoPandas, which we will use to look at vector data
  - For now, we will start with an ordinary table of data.

```
# Do our imports
import pandas
import os

# Set our file paths (YOU SHOULD HAVE DOWNLOADED THIS FROM GOOGLE DRIVE!)
data_directory = '../data'
food_prices_filename = 'world_monthly_food_prices.csv'
food_prices_path = os.path.join(data_directory, food_prices_filename)

# Pandas provides a read_csv function that will read a CSV file into a dataframe
food_prices = pandas.read_csv(food_prices_path)

# Print the dataframe using default printing options
print(food_prices)
```

```
                                        Domain Code                  Domain  \
0                                                CP  Consumer Price Indices
1                                                CP  Consumer Price Indices
2                                                CP  Consumer Price Indices
3                                                CP  Consumer Price Indices
4                                                CP  Consumer Price Indices
```

```
..                                       ...                     ...
176                                       CP  Consumer Price Indices
177                                       CP  Consumer Price Indices
178                                       CP  Consumer Price Indices
179                                       CP  Consumer Price Indices
180  FAOSTAT Date: Wed Aug 17 16:12:56 CEST 2016                     NaN

     AreaCode AreaName  ElementCode ElementName  ItemCode  \
0      5000.0    World       7001.0     January   23013.0
1      5000.0    World       7001.0     January   23013.0
2      5000.0    World       7001.0     January   23013.0
3      5000.0    World       7001.0     January   23013.0
4      5000.0    World       7001.0     January   23013.0
..        ...      ...          ...         ...       ...
176    5000.0    World       7012.0    December   23013.0
177    5000.0    World       7012.0    December   23013.0
178    5000.0    World       7012.0    December   23013.0
179    5000.0    World       7012.0    December   23013.0
180       NaN      NaN          NaN         NaN       NaN

                                  ItemName    Year  Value  Flag  \
0    Consumer Prices, Food Indices (2000 = 100)  2000.0   99.5   NaN
1    Consumer Prices, Food Indices (2000 = 100)  2001.0  101.5   NaN
2    Consumer Prices, Food Indices (2000 = 100)  2002.0  106.5   NaN
3    Consumer Prices, Food Indices (2000 = 100)  2003.0  112.5   NaN
4    Consumer Prices, Food Indices (2000 = 100)  2004.0  119.2   NaN
..                                         ...     ...    ...   ...
176  Consumer Prices, Food Indices (2000 = 100)  2011.0  210.3   NaN
177  Consumer Prices, Food Indices (2000 = 100)  2012.0  224.2   NaN
178  Consumer Prices, Food Indices (2000 = 100)  2013.0  239.0   NaN
179  Consumer Prices, Food Indices (2000 = 100)  2014.0  250.9   NaN
180                                        NaN     NaN    NaN   NaN

             FlagD
0    Official data
1    Official data
2    Official data
3    Official data
4    Official data
..             ...
176  Official data
177  Official data
178  Official data
```

```
179  Official data
180            NaN

[181 rows x 12 columns]
```

- For instance look at the column names

```
print('List of column names:', food_prices.columns)
```

```
List of column names: Index(['Domain Code', 'Domain', 'AreaCode', 'AreaName', 'ElementCode',
       'ElementName', 'ItemCode', 'ItemName', 'Year', 'Value', 'Flag',
       'FlagD'],
      dtype='object')
```

Get a specific column

- To get a column, use square braces with the name of the column.

```
food_prices_value_column = food_prices['Value']

print('Specific column:', food_prices_value_column)
```

```
Specific column: 0      99.5
1      101.5
2      106.5
3      112.5
4      119.2
       ...
176    210.3
177    224.2
178    239.0
179    250.9
180      NaN
Name: Value, Length: 181, dtype: float64
```

```
print('Specific value in that column:', food_prices_value_column[6])
```

```
Specific value in that column: 132.2
```

Under the hood

- All parts of the pandas dataframe are represented via numpy arrays
    - If you want to access them as numpy arrays instead of Pandas dataframes or columns, you can use the .values attribute

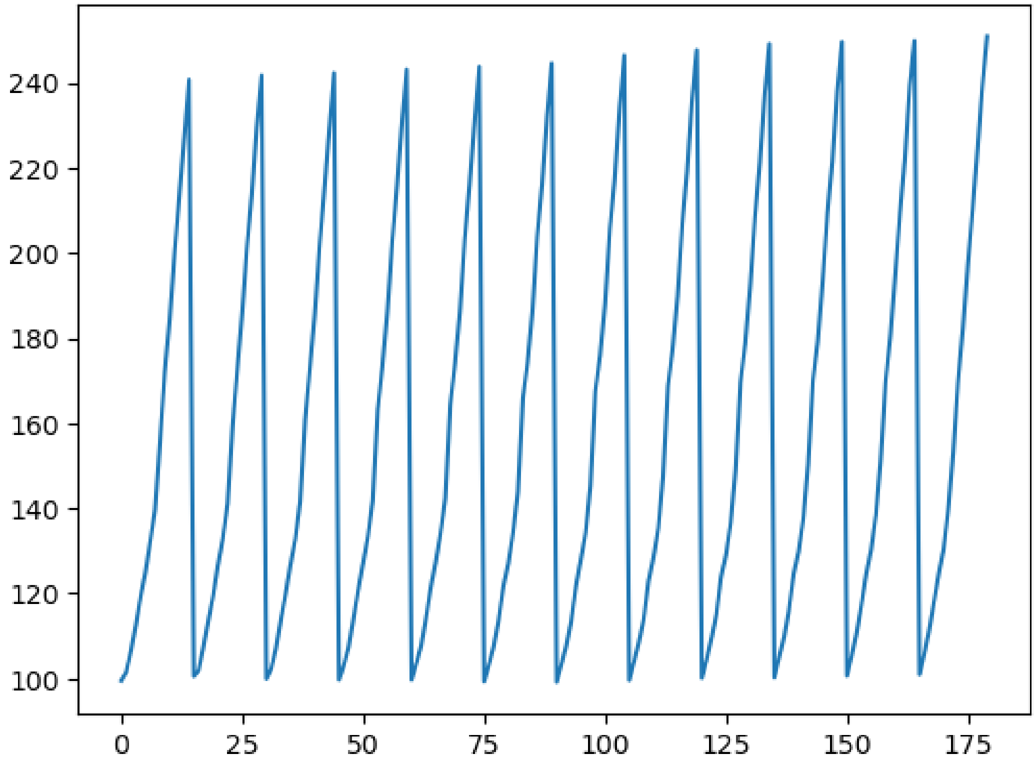
```
numpy_array = food_prices_value_column.values
print(numpy_array)
```

```
[ 99.5 101.5 106.5 112.5 119.2 125.  132.2 139.7 155.5 172.  184.2 199.9
 213.2 227.2 240.7 100.5 101.9 107.5 113.7 119.5 126.8 133.  141.2 159.4
 172.6 185.9 201.6 213.7 229.7 241.8  99.9 102.3 107.2 113.9 120.  126.9
 132.9 141.6 161.  173.  185.6 201.9 214.9 229.2 242.3  99.7 103.1 107.4
 114.2 120.9 127.3 133.5 141.9 163.  173.4 186.1 202.6 215.6 230.6 243.1
  99.7 103.5 107.7 114.2 121.4 127.4 134.2 142.4 164.3 174.  186.1 203.2
 215.9 230.6 243.8  99.3 103.2 107.6 113.7 121.6 127.  134.3 143.7 165.8
 174.3 186.4 204.2 216.  232.1 244.6  99.2 103.3 107.7 113.5 121.7 128.
 134.3 145.3 167.5 175.9 188.  205.7 217.3 234.  246.4  99.6 103.9 108.5
 113.9 122.8 128.4 135.2 147.4 168.6 177.5 189.9 207.1 219.5 236.2 247.7
 100.1 104.4 109.4 114.9 123.9 129.  136.7 149.  169.7 178.8 192.1 208.5
 221.  237.4 249.1 100.2 105.  109.6 116.  124.5 129.8 137.6 150.4 170.1
 179.3 193.5 209.4 221.4 238.4 249.6 100.6 105.3 110.6 117.2 124.5 130.3
 138.5 151.5 169.8 180.4 195.3 209.6 222.1 239.4 249.9 100.9 105.7 111.3
 118.  124.6 130.5 140.1 153.2 170.  182.2 197.3 210.3 224.2 239.  250.9
   nan]
```

Default plotting

- By default, pandas has a plot function
    - Let's try it

```
import matplotlib
from matplotlib import pyplot as plt
plt.plot(food_prices['Value'])
plt.show()
```



Discussion point

- Why is the data like this?

Pandas Functionality

- Let's start by importing everything afresh and setting a random seed.
  - Setting the random seed means we will get the same random numbers each time we run the code.

```
import os

import numpy as np
import pandas as pd

# Set a seed value for the random generator
np.random.seed(48151623)
```

Create an individual column

- In pandas, they are called Series
- One very important thing to note is that in the output, we see TWO columns
  - Pandas always includes an index for every Series or Dataframe
  - The index is the leftmost column, and is used to identify each row

```
# Creating a Series by passing a list of values, letting pandas create a default integer i
s = pd.Series([1, 3, 5, np.nan, 6, 8])
print(s)
```

```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

There are many built-in ways of generating series

- For instance, you can automatically generate a series of dates

```
# Pandas is very detailed in dealing with dates and all the quirks (leap year?) that this
dates = pd.date_range('20130101', periods=6)
print(dates)
```

```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
```

```
# Creating a DataFrame by passing a NumPy array, with a datetime index and labeled columns
df = pd.DataFrame(np.random.randn(6, 4), columns=list('ABCD'))
print('df:\n', df)


df2 = pd.DataFrame({'A': 1.,
                    'B': pd.Timestamp('20130102'),
                    'C': pd.Series(1, index=list(range(4)), dtype='float32'),
```

```
                    'D': np.array([3] * 4, dtype='int32'),
                    'E': pd.Categorical(["test", "train", "test", "train"]),
                    'F': 'foo'})

# df.head()
# print(df.index)
# print(df.columns)
df.describe()

# Also note that a dataframe is really just a numpy array dressed up with extra trappings.
# can get back the raw array (though this might lose a lot of functionality).
a = df.to_numpy()
print('a\n', a)

# Sorting Values:

# Also, I want to illustrate THE MOST COMMON MISTAKE people make with Pandas.

# The sort_values method (a method is just a function attached to an object) returns a NEW
# Thus, in the line below, if you just printed df, it would not be sorted because we didn'
df.sort_values(by='B')
# print('Not sorted:\n', df)

# Easy way to get around this is just to assign the returned dataframe to a variable (even
df = df.sort_values(by='B')
# print('Sorted with return:\n', df)

# Alternatively, if you hate returning things, there is the inplace=True command, which wi
df.sort_values(by='B', inplace=True)
# print('Sorted inplace:\n', df)

## Selection/subsetting of data

# Selecting a single column, which yields a Series, equivalent to df.A
df['A']
df.A

# Selecting via [], which slices the rows.
df[0:3] # CAN BE SLOW

# Note, slicing above, which uses the
```

```
# standard Python / Numpy expressions for selecting and setting are intuitiveits best to u
# the optimized pandas data access methods, .at, .iat, .loc and .iloc.

## Selecting by LABELS, loc and iloc

r = df.loc[0] # 0-th row.

# print('r', r)

# Discuss difference between df['A'] and df.loc[0]
r = df.loc[0, 'A']

r = df.loc[:, 'A'] # Colon is a slice, an empty colon means ALL the values.

# OPTIMIZATION:
# for faster single point access, use:
r = df.at[0, 'A']

# SELECTING BY POSITION
r = df.iloc[3]

# Selecting with slices
r = df.iloc[3:5, 0:2]

# Slices again with an empty slice.
r = df.iloc[1:3, :]

r = df.iloc[:, 1:3]

# SIMILAR OPTIMIZATION:
r = df.iat[1, 1]

# Boolean indexing
# Using a single column's values to select data.
r = df[df['A'] > 0]

# Make a copy (why?) and add a column
df2 = df.copy()
df2['E'] = ['one', 'one', 'two', 'three', 'four', 'three']
r = df2[df2['E'].isin(['two', 'four'])]
```

```
# Setting by assigning with a NumPy array:
df.loc[:, 'D'] = np.array([5] * len(df))

# Missing data

# First we're going to create a new df by "reindexing" the old one, which will shuffle the
# order according to the index provided. At the same time, we're going to add on a new, em
# EE, which we set as 1 for the first two obs.

df1 = df.reindex(index=[2, 0, 1, 3], columns=list(df.columns) + ['EE'])
df1.loc[0:1, 'EE'] = 1
# print(df1)

# Apply: Similar to R. Applies a function across many cells (fast because it's vectorized)
df.apply(np.cumsum)
df.apply(lambda x: x.max() - x.min())

# Concat
s = pd.Series(range(0, 6))
# print('s', s)

r = pd.concat([df, s]) # Concatenate it, default is by row, which just puts it on the bott

r = pd.concat([df, s], axis=1) # Concatenate as a new column

# print(r) # Result when concatenating a series of the same size.

s = pd.Series(range(0, 7))
r = pd.concat([df, s], axis=1) # Concatenate as a new column

s = pd.Series(range(0, 2))
r = pd.concat([df, s], axis=1) # Concatenate as a new column

# Join
# SQL style merges. See the Database style joining section.

left = pd.DataFrame({'key': ['foo', 'bar'], 'lval': [1, 2]})
right = pd.DataFrame({'key': ['foo', 'bar'], 'rval': [4, 5]})

# print(left)
# print(right)
```

```
df = pd.merge(left, right, on='key')

# print('df:\n', df)

# Stacking
stacked = df.stack()
# print('stacked:\n', stacked)


# Pivot Tables
df = pd.DataFrame({'A': ['one', 'one', 'two', 'three'] * 3,
                   'B': ['A', 'B', 'C'] * 4,
                   'C': ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 2,
                   'D': np.random.randn(12),
                   'E': np.random.randn(12)})

# print(df) # SPREADSHEET VIEW
df = pd.pivot_table(df, values='D', index=['A', 'B'], columns=['C'])
# print(df) # Multiindexed (Pivot table) view.

# NOTICE that a pivot table is just the above date but where specific things have been mad
# indices.

# PLOTTING
ts = pd.Series(np.random.randn(1000),
            index=pd.date_range('1/1/2000', periods=1000))

ts = ts.cumsum()
ts.plot()
import matplotlib.pyplot as plt
# plt.show()


# Writing to files

df.to_csv('foo.csv')

# Reading files:

# FIRST NOTE, here we are using relative paths (which you should almost always do too). th
# this path works if you organized your data into the folder structure I suggested.
```

```python
wdi_filename = "WDI_CO2_data.csv"
wdi_path = os.path.join(data_directory, wdi_filename)
df = pd.read_csv(wdi_path)

print('csv read as a df\n', df)

# For reference, here's the Excel version
# df = pd.read_excel('foo.xlsx', 'Sheet1', index_col=None, na_values=['NA'])

cols = list(df.columns)

# Make a subset of only 2 cols
r = df[['Country Code', '1970 [YR1970]']]
# print(r)

r = df.loc[df['Country Code'] == 'CAN']
# print('r', r)

rr = r.loc[df['Series Name'] == 'Total greenhouse gas emissions (kt of CO2 equivalent)']
print(rr)

# Class exercise: Plot the emissions of CO2 for Canada (or whereever I don't care).
```

```
df:
          A         B         C         D
0  0.336581  0.923754 -0.277124  0.388604
1 -1.295428  3.296657 -0.698246  0.245552
2 -1.086536  1.187113 -0.153344 -1.264476
3  0.798694  1.330577 -0.113975 -0.060949
4 -0.076321 -0.240310  0.728421 -0.384309
5  0.634212  1.605129  1.415844  0.385849
a
 [[ 0.33658127  0.92375415 -0.27712413  0.38860406]
 [-1.2954285   3.29665716 -0.69824576  0.24555238]
 [-1.0865361   1.18711302 -0.15334368 -1.26447611]
 [ 0.79869365  1.33057707 -0.11397528 -0.06094946]
 [-0.07632075 -0.24031028  0.72842085 -0.38430931]
 [ 0.6342119   1.60512881  1.41584369  0.38584939]]
csv read as a df
                                            Country Name Country Code  \
0                                           Afghanistan          AFG
1                                           Afghanistan          AFG
```

```
2                                         Afghanistan          AFG
3                                         Afghanistan          AFG
4                                         Afghanistan          AFG
...                                               ...          ...
2640                                              NaN          NaN
2641                                              NaN          NaN
2642                                              NaN          NaN
2643  Data from database: World Development Indicators          NaN
2644                        Last Updated: 10/15/2020          NaN

                                            Series Name          Series Code  \
0     Access to clean fuels and technologies for coo...       EG.CFT.ACCS.ZS
1               Access to electricity (% of population)       EG.ELC.ACCS.ZS
2     Adjusted net enrollment rate, primary (% of pr...          SE.PRM.TENR
3                          Arable land (% of land area)       AG.LND.ARBL.ZS
4     Agricultural methane emissions (thousand metri...  EN.ATM.METH.AG.KT.CE
...                                                 ...                  ...
2640                                                NaN                  NaN
2641                                                NaN                  NaN
2642                                                NaN                  NaN
2643                                                NaN                  NaN
2644                                                NaN                  NaN

     1960 [YR1960]     1961 [YR1961]     1962 [YR1962]     1963 [YR1963]  \
0               ..                ..                ..                ..
1               ..                ..                ..                ..
2               ..                ..                ..                ..
3               ..  11.7176730079956  11.7942591060871  11.8708452041785
4               ..                ..                ..                ..
...            ...               ...               ...               ...
2640           NaN               NaN               NaN               NaN
2641           NaN               NaN               NaN               NaN
2642           NaN               NaN               NaN               NaN
2643           NaN               NaN               NaN               NaN
2644           NaN               NaN               NaN               NaN

       1964 [YR1964]   1965 [YR1965]  ...     2011 [YR2011]     2012 [YR2012]  \
0                 ..              ..  ...             22.33             24.08
1                 ..              ..  ...  43.2220189082037              69.1
2                 ..              ..  ...                ..                ..
3     11.94743130227  11.94743130227  ...  11.9336458046135  11.9321140826517
4                 ..              ..  ...                ..                ..
...              ...             ...  ...               ...               ...
```

```
2640            NaN             NaN  ...             NaN             NaN
2641            NaN             NaN  ...             NaN             NaN
2642            NaN             NaN  ...             NaN             NaN
2643            NaN             NaN  ...             NaN             NaN
2644            NaN             NaN  ...             NaN             NaN

         2013 [YR2013]    2014 [YR2014]    2015 [YR2015]     2016 [YR2016]  \
0                26.17            27.99             30.1             32.44
1     68.9332656860352             89.5             71.5              97.7
2                   ..               ..               ..                ..
3     11.9244554728426  11.903011365377  11.893821033606  11.8386790429801
4                   ..               ..               ..                ..
...                ...              ...              ...               ...
2640               NaN              NaN              NaN               NaN
2641               NaN              NaN              NaN               NaN
2642               NaN              NaN              NaN               NaN
2643               NaN              NaN              NaN               NaN
2644               NaN              NaN              NaN               NaN

     2017 [YR2017]     2018 [YR2018] 2019 [YR2019] 2020 [YR2020]
0               ..                ..            ..            ..
1             97.7  98.7132034301758            ..            ..
2               ..                ..            ..            ..
3               ..                ..            ..            ..
4               ..                ..            ..            ..
...            ...               ...           ...           ...
2640           NaN               NaN           NaN           NaN
2641           NaN               NaN           NaN           NaN
2642           NaN               NaN           NaN           NaN
2643           NaN               NaN           NaN           NaN
2644           NaN               NaN           NaN           NaN

[2645 rows x 65 columns]
    Country Name Country Code  \
369       Canada          CAN

                                           Series Name        Series Code  \
369  Total greenhouse gas emissions (kt of CO2 equi...  EN.ATM.GHGT.KT.CE

    1960 [YR1960] 1961 [YR1961] 1962 [YR1962] 1963 [YR1963] 1964 [YR1964]  \
369            ..            ..            ..            ..            ..

    1965 [YR1965]  ...     2011 [YR2011]     2012 [YR2012] 2013 [YR2013]  \
```

```
369             ..  ...  1033481.98200961  1027063.85487082            ..

    2014 [YR2014] 2015 [YR2015] 2016 [YR2016] 2017 [YR2017] 2018 [YR2018]  \
369             ..            ..            ..            ..            ..

    2019 [YR2019] 2020 [YR2020]
369             ..            ..

[1 rows x 65 columns]
```