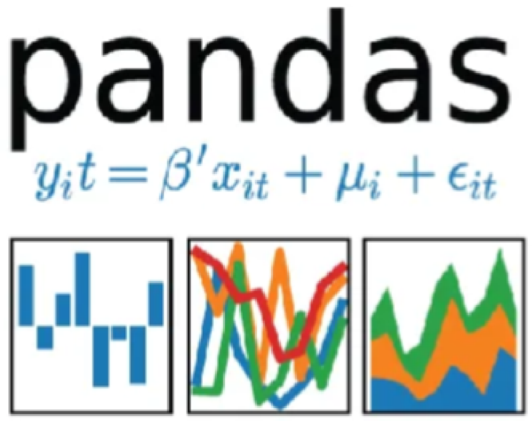
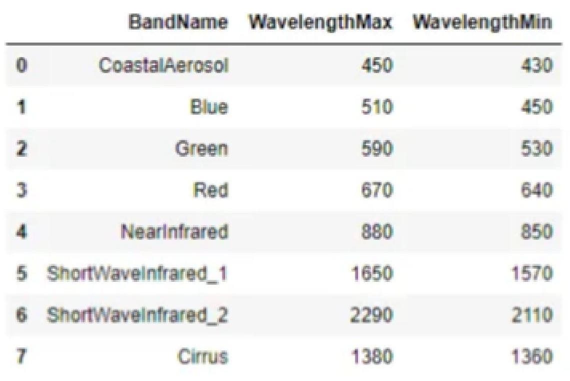
pandas: Series and DataFrames
Mathematical Programming with Python

https://people.sc.fsu.edu/~jburkardt/classes/math1800_2023/pandas1/pandas1.pdf





	BandName	WavelengthMax	WavelengthMin
0	CoastalAerosol	450	430
1	Blue	510	450
2	Green	590	530
3	Red	670	640
4	NearInfrared	880	850
5	ShortWaveInfrared_1	1650	1570
6	ShortWaveInfrared_2	2290	2110
7	Cirrus	1380	1360

"The pandas library"

- `pandas` *offers tools for data analysis;*
- *The basic structures are the* `Series` *and* `DataFrame`*;*
- *A* `Series` *is simply a one-dimensional sequence of values, similar to a* `numpy` *array;*
- *A* `DataFrame` *stores a set of* `Series` *in a two dimensional table;*
- *A* `DataFrame` *can hold a combination of text, numbers, and other items;*
- *Rows and columns of a* `DataFrame` *can be accessed by labels rather than indices;*
- *Data cleaning is a preliminary task to identify data entries that are missing, implausible, or inappropriate;*
- *Statistical analysis looks for relations between columns of data;*
- *A variety of plotting options can be applied to the data;*
- *Many functions are available for analyzing and aggregating the information in a* `DataFrame`*;*

1 A Python library for data analysis

The `pandas` library offers tools for the collection, cleaning, analysis of large collections of data. A typical dataset is stored in a *dataframe*, which can be thought of as a generalization of an array. But the items in a `numpy` array are all of the same datatype, and are accessed only by row and column index. In contrast, each column of a dataframe can include data of any type, and can be accessed by specifying the corresponding column label. The rows of the dataframe are often simply indices, but can also involve labels.

Because `pandas` is designed for real world applications, it expects the data to be in a rougher format than the mathematical arrays handled by `numpy`. So the first important operation is `data input`, in which raw data files in a variety of inputs, such as Excel files `xls`, comma separated value files `csv`, Java Script Object `json`, or even simple text files `txt`. This data must be read into the uniform `pandas` dataframe format before any further work can be done.

Because `pandas` expects the data may be coming from business or medical applications, it has to deal with the likelihood that some of the data items are missing or nonsensical or have been given a special value indicating a problem.

The `pandas` homepage at `https://pandas/pydata/org` explains how to download and install the library.

By convention, the shortcut `pd` is used to refer to elements of the `pandas` library after it has been imported.

```
  import pandas as pd
```

2 The Series data structure

The most important `pandas` object is the `DataFrame`, which is a sort of table of values. Each column of this table is a list of values of the same datatype, known as a `Series`. We will start our discussion by looking at how these objects can be defined and manipulated.

A new `Series` can be created as a list of values fed to the `pd.Series()` function:

```
  river_lengths = pd.Series ( [ 6300, 6650, 6275, 6400 ] )
```

which will print out as

```
0  6300
1  6650
2  6275
3  6400
dtype: int64
```

The numbers that appear on the left are the index values associated with each item of data. All `Series` data will have a list of indices; they can be specified, but otherwise they default to a simple integer sequence is used. Individual entries of the `Series` can be referenced by index:

```
  for i in range ( 0, 4 ):
    print ( river_lengths[i] )
```

or by a generator:

```
for length in river_lengths:
  print ( length )
```

The default index is not an integer array, but an integer generator, something like the Python `range()` function. The index is an attribute of the `Series` and can be referenced.

```
  print ( river_lengths.index )
  RangeIndex(start=0, stop=4, step=1)
```

The user can replace the default index by more meaningful values:

```
  river_lengths = pd.Series ( [6300, 6650, 6275, 6400 ],
    index = [ 'Yangtze', 'Nile', 'Mississippi', 'Amazon' ] )
```

Now, we can reference our river lengths by name:

```
  print ( "  river_lengths['Nile'] = ", river_lengths['Nile'] )
```

The data can be given a descriptive name (aside from the variable name) and a data type:

```
  river_lengths = pd.Series ( [6300, 6650, 6275, 6400 ],
    name = 'Length (km)', dtype = float )
```

This extra information will show up when we print the `Series` or some of its entries:

```
  print ( river_lengths )

Yangtze        6300.0
Nile           6650.0
Mississippi    6275.0
Amazon         6400.0
Name: Length (km), dtype: float64
```

We can filter the data

```
  print ( river_lengths[river_lengths < 6500] )

Yangtze        6300.0
Mississippi    6275.0
Amazon         6400.0
Name: Length (km), dtype: float64
```

We can create a new `Series`, in this case, a copy of our river lengths converted to miles:

```
  river_lengths_miles = river_lengths * 0.621371
  river_lengths_miles.name = "Length (miles)"

Yangtze        3914.637300
Nile           4132.117150
Mississippi    3899.103025
Amazon         3976.774400
Name: Length (miles), dtype: float64
```

3 Creating a new Series from old ones

Computations can be done using the entries in a `Series`. For instance, suppose we have created two `Series`, containing the mass m and radius r of various astronomical bodies. We would like to create a new `Series` which stores the surface gravity of a body as $grav = G * m/r^2$, where G is the gravitational constant, stored in `scipy.constants.G`.

We carry out the computation almost as though we were working with `numpy` arrays. Some data is missing, and in those cases the result is reported as `NaN` (Not-a-Number). We can remove such results from our table before presenting it.

We created our two `Series` as Python `dicts`, so they each come with an index and name already:

```
  mass = pd.Series ( { \
    'Ganymede' : 1.482e23, \
    'Callisto' : 1.076323, \
    'Io'       : 8.932e22, \
    'Europa'   : 4.800e22, \
    'Moon'     : 7.342e22, \
    'Earth'    : 5.972e24 },
    name = 'mass (kg)' )

  radius = pd.Series ( { \
    'Ganymede' : 2.634e6, \
    'Io'       : 1.822e6, \
    'Moon'     : 1.737e6, \
    'Earth'    : 6.371e6 },
    name = 'radius (m)' )
```

Now we compute the surface gravity values, creating a new `Series`, for which we can also specify the name and the index name:

```
from scipy.constants import G
gravity = G * mass / radius**2
gravity.name = 'surface gravity m/s**2'
gravity.index.name = 'Body'
```

with the result:

```
Body
Callisto         NaN
Earth       9.819973
Europa           NaN
Ganymede    1.425681
Io          1.795799
Moon        1.624129
Name: surface gravity m/s**2, dtype: float64
```

We can report whether each entry has the value `NaN` with the command:

```
gravity.isnull()
```

We can make a copy of our `Series` with the `NaN` data: removed:

```
gravity = gravity.dropna()
```

and we can even copy out the computed values into a `numpy` array:

```
gravity_array = gravity.values
```

4 Creating a DataFrame Online

A `DataFrame` can be thought of as a two-dimensional table of an ordered and labeled set of `Series` columns, sharing the same index. One way to create a `DataFrame` is to use the command `pd.DataFrame()` to reorganize the information in a dictionary, possibly adding a new index:

```
moon_data = { \
  'mass'   : [ 1.482e23,   1.076e23, 8.932e23,  4.800e22,  7.342e22 ], \
  'radius' : [ 2.634e6,    None,     1.822e6,   None,      1.737e6  ], \
  'parent' : [ 'Jupiter', 'Jupiter', 'Jupiter', 'Jupiter', 'Earth'  ] }

moon_index = [ 'Ganymede', 'Callisto', 'Io', 'Europa', 'Moon' ]

moon_df = pd.DataFrame ( moon_data, index = moon_index )
```

As a `DataFrame`, the information will now print out as:

```
                  mass     radius   parent
Ganymede  1.482000e+23  2634000.0  Jupiter
Callisto  1.076000e+23        NaN  Jupiter
Io        8.932000e+23  1822000.0  Jupiter
Europa    4.800000e+22        NaN  Jupiter
Moon      7.342000e+22  1737000.0    Earth
```

Some of the differences between a `numpy` array and a `DataFrame` should be clear now. Each column of the `DataFrame` is a `Series` and hence consists of data of the same type, but different columns may contain different data types: character, float, integer, boolean. And while `numpy` uses numeric indexing, in a `DataFrame` each row has an index, each column has a header, these can be character strings, and can be used for indexing.

5 Accessing rows, columns, and cells

To refer to a value or the index of a value in a `DataFrame`, `pandas` supplies functions `loc[]` and `iloc[]`.

To print the values in a specific row, specify the row label:

```
print ( moon_df.loc["Europa"] )
```

resulting in:

```
mass      48000000000000000000000.0
radius                          NaN
parent                      Jupiter
Name: Europa, dtype: object
```

To print the values in a specific column, specify the row label as a colon, and follow it by the column label:

```
print ( moon_df.loc[:,"mass"] )
```

resulting in:

```
Ganymede    1.482000e+23
Callisto    1.076000e+23
Io          8.932000e+23
Europa      4.800000e+22
Moon        7.342000e+22
Name: mass, dtype: float64
```

To print a single value, specify the row and column labels:

```
    print ( moon_df.loc["Europa","mass" )
```

resulting in:

```
4.8e+22
```

We can use `loc()` to identify an entry we wish to change:

```
moon_df.loc['Callisto','radius'] = 2410300
```

The `iloc()` function uses numeric indexing to access rows, columns or specific entries.

```
print ( moon_df.iloc[3] )
print ( moon_df.iloc[:,0] )
print ( moon_df.iloc[3,0] )
```

6 Fitting a line to data

In our next example, we will find two variables which seem to be vary together. A simple model would be a linear relationship, of the form $y = ax + b$. Although our data is unlikely to lie exactly on any such line, we can try to determine values a and b that represent a good fit, as long as we can define what we mean by a good fit.

The linear least squares solution to our problem finds values such that we minimize the sum of the squared errors,

$$E(a, b) = \sum_{i=0}^{i<n} (ax_i + b - y_i)^2$$

The solution to this problem can be found by setting to zero the partial derivatives $\frac{\partial E}{\partial a}$ and $\frac{\partial E}{\partial b}$ and doing some manipulation to find

$$a = \frac{x \cdot (y - \bar{y})}{x \cdot (x - \bar{x})}$$
$$b = \bar{y} - a * \bar{x}$$

where the barred quanties are mean values, and the dot indicates a vector dot product. This can be programmed as:

```
def llsq ( x, y ):
  import numpy as np
  xbar = np.mean ( x )
  ybar = np.mean ( y )

  xy = np.dot ( x, y - ybar )
  xx = np.dot ( x, x - xbar )

  a = xy / xx
  b = ybar - a * xbar

  return a, b
```

In the coming example, we will suspect that our data is well described by a linear relationship, we will want to compute a and b, and then draw our linear formula along with a scatterplot of our data for comparison.

7 Creating a DataFrame from a File

Most of the time, `pandas` is used to handle large datasets which have been created by other programs, and stored in files. These files

- come in a variety of formats;
- may have column headers;
- may have row indices;
- may very likely have some missing data.

`pandas` supplies a number of special functions for reading a file in these formats, for finding the headers and indices or supplying default values if missing, and for treating missing data.

The most common file reading functions include:

- `pd.read_csv()`, for Comma Separated Values (csv) data;
- `pd.read_excel()`, for Microsoft Excel data;
- `pd.read_json()`, for JSON data;

The first arguments to the file reading functions is the name of the file to be read. Following that, there are many optional arguments that allow you to deal with the peculiarities of the file you are reading and how you want to handle it.

8 The India Dataset

The file *india_data.csv* is available on the Canvas website, and on the text website at `https://scipython.com/eg/bak`. We will use `pandas` to read, modify, and analyze the data in this file. The file contains columns of demographic data on the 36 states and union territories (UT) of India.

The data file can be read in with:

```
df = pd.read_csv ( 'india_data.csv', index_col = 0 )
```

The DataFrame contains an index of State names:

```
df.index

Index(['Uttar Pradesh', 'Maharashtra', 'Bihar', 'West Bengal',
      ...
      'Dadra and Nagar Haveli', 'Daman and Diu', 'Lakshadweep'],
     dtype='object', name='State/UT')
```

and column labels:

```
df.columns

Index(['Male Population', 'Female Population', 'Area (km2)',
        'Male Literacy (%)', 'Female Literacy (%)', 'Fertility Rate'],
       dtype='object')
```

We can quickly inspect the DataFrame with `df.head(n)`, which outputs the first `n` rows (or five rows if not specified):

```
df.head()

                Male Population  ...  Female Literacy (%)
State/UT                         ...
Uttar Pradesh         104480510  ...                59.26
Maharashtra            58243056  ...                75.48
Bihar                  54278157  ...                53.33
West Bengal            46809027  ...                71.16
Madhya Pradesh         37612306  ...                60.02

[5 rows x 5 columns]
```

`pandas` makes it straightforward to compute new columns for our DataFrame. We can add a column for the population of each State:

```
df['Population'] = df['Male Population'] + df['Female Population']
```

and then we can sum the `'Population'` column to get the total population:

```
total_pop = df['Population'].sum()
print('Total population:', total_pop )

Total population: 1,210,754,977
```

We can add a column for the population density, and as a check, print the density of West Bengal:

```
df['Population Density (km-2)'] = df['Population'] / df['Area (km2)']
df.loc['West Bengal', 'Population Density (km-2)']

1028.44009149089
```

and compute the mean population density over India:

```
total_pop / df['Area (km2)'].sum()

368.3195047153525
```

Maximum and minimum values are obtained in the same way as in `numpy`. For example:

```
df['Male Literacy (%)'].min()

73.39
```

Perhaps more usefully, `idxmin` and `idxmax` return the index labels of the minimum and maximum values, respectively. Here, we seek the largest state/UT by area:

```
df['Area (km2)'].idxmax()

'Rajasthan'
```

Naturally, the value returned can be passed to `df.loc[]` to obtain the entire row. For example, the row corresponding to the most densely populated State / UT:

```
df.loc[df['Population Density (km−2)'].idxmax()]

Male Population                8887326
Female Population              7800615
Area (km2)                     1484
Male Literacy (%)              91.03
Female Literacy (%)            80.93
Population                     16687940
Population Density (km−2)      1.124524e+04
Name: Delhi, dtype: float64
```

Correlation statistics between `DataFrames` or `Series` can be calculated with the `corr()` function:

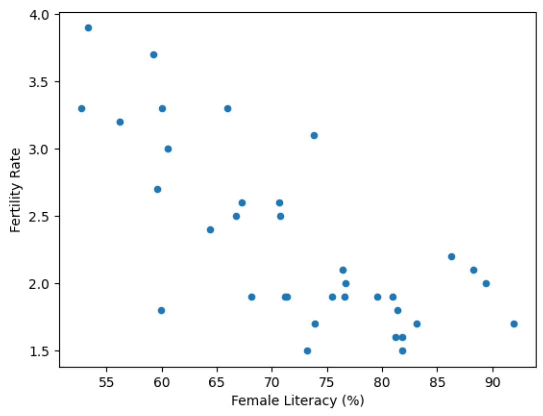
```
df['Female Literacy (%)'].corr(df['Fertility Rate'])

−0.7361949271996956
```

The correlation is relatively large and negative, suggesting that as Female Literacy rises, there is a strong tendency for the Fertility Rate to decline.

In this example, because just two columns of data are being compared, a single correlation coefficient is produced. If we seek the correlations of many columns of data, a full correlation matrix is returned as a new `DataFrame`.

`pandas` can produce simple, labeled plots and charts from a `DataFrame` using `matplotlib`. For instance, we can compute a scatter plot of literacy versus fertility:

```
df.plot.scatter ( 'Female Literacy (%)', 'Fertility Rate' )
```



We would like to use the method of linear least squares to get a linear fit to the data of the form

```
Fertility Rate = a * Female Literacy + b
```

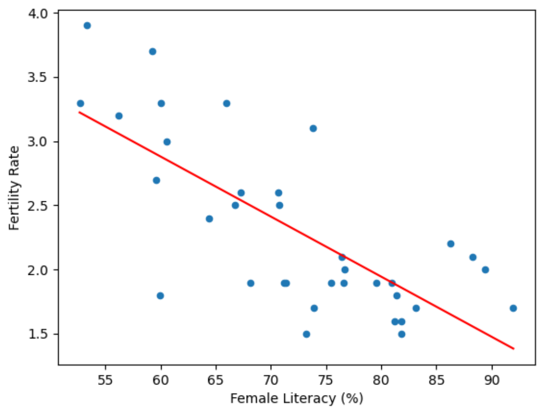
If we do this immediately, the code fails because one of the values of literacy is missing, and so all the numeric results become NaN's. To get around this problem, we need to drop records with missing data:

```
df = df.dropna ( )
```

Now we can use the method of linear least squares to come up with a linear fit to the data, by extracting the numerical values, and calling

```
x = df[ 'Female Literacy (%)'].values
y = df[ 'Fertility Rate' ].values
a, b = llsq ( x, y )
a =  −0.045
b =   5.627
```

and now we can replot our data with the addition of our linear least squares function $y = -0.045 * x + 5.627$:


Fertility Rate
Female Literacy (%)