What is Pandas?

Pandas is a Python library used for working with data sets.

It has functions for analyzing, cleaning, exploring, and manipulating data.

The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Why Use Pandas?

Pandas allows us to analyze big data and make conclusions based on statistical theories.

Pandas can clean messy data sets, and make them readable and relevant.

Relevant data is very important in data science.

What Can Pandas Do?

Pandas gives you answers about the data. Like:

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?

Installation of Pandas

If you have Python and PIP already installed on a system, then installation of Pandas is very easy.

Install it using this command:

```
pip install pandas
```

Import Pandas

```
import pandas

import pandas

mydataset = {

: ["BMW", "Volvo", "Ford"],
  'passings': [3, 7, 2]
}

myvar = pandas.DataFrame(mydataset)

print(myvar)
```

Create an alias with the as keyword while importing:

Now the Pandas package can be referred to as pd instead of pandas.

```
import pandas as pd

mydataset = {
  'cars': ["BMW", "Volvo", "Ford"],
  'passings': [3, 7, 2]
}

myvar = pd.DataFrame(mydataset)

print(myvar)
print(pd.__version__)
```

Pandas Series

What is a Series?

A Pandas Series is like a column in a table.

It is a one-dimensional array holding data of any type.

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a)

print(myvar)
```

Labels

If nothing else is specified, the values are labeled with their index number. First value has index 0, second value has index 1 etc.

This label can be used to access a specified value.

```
print(myvar[0])
```

Create Labels

With the index argument, you can name your own labels.

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a, index = ["x", "y", "z"])

print(myvar)
```

When you have created labels, you can access an item by referring to the label.

```
print(myvar["y"])
```

Key/Value Objects as Series

```
import pandas as pd
```

```
calories = {"day1": 420, "day2": 380, "day3": 390}

myvar = pd.Series(calories)

print(myvar)
```

DataFrames

Data sets in Pandas are usually multi-dimensional tables, called DataFrames.

Series is like a column, a DataFrame is the whole table.

Create a DataFrame from two Series:

```
import pandas as pd

data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}

myvar = pd.DataFrame(data)

print(myvar)
```

What is a DataFrame?

A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.

```
import pandas as pd

data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}

#load data into a DataFrame object:
df = pd.DataFrame(data)
```

```
print(df)
```

Locate Row

As you can see from the result above, the DataFrame is like a table with rows and columns.

Pandas use the loc attribute to return one or more specified row(s)

```
#refer to the row index:

print(df.loc[0])
```

Named Indexes

With the index argument, you can name your own indexes.

```
import pandas as pd

data = {

  "calories": [420, 380, 390],

  "duration": [50, 40, 45]

}

df = pd.DataFrame(data, index = ["day1", "day2", "day3"])

print(df)
```

Load Files Into a DataFrame

If your data sets are stored in a file, Pandas can load them into a DataFrame.

```
import pandas as pd

df = pd.read_csv('data.csv')
```

```
print(df)
```

Data.csv

```
Duration,Pulse,Maxpulse,Calories
60,110,130,409.1
60,117,145,479.0
60,103,135,340.0
45,109,175,282.4
45,117,148,406.0
60,102,127,300.5
60,110,136,374.0
45,104,134,253.3
30,109,133,195.1
60,98,124,269.0
60,103,147,329.3
60,100,120,250.7
60,106,128,345.3
60,104,132,379.3
60,98,123,275.0
60,98,120,215.2
60,100,120,300.0
45,90,112,
60,103,123,323.0
45,97,125,243.0
60,108,131,364.2
45,100,119,282.0
60,130,101,300.0
45,105,132,246.0
60,102,126,334.5
60,100,120,250.0
60,92,118,241.0
60,103,132
60,100,132,280.0
60,102,129,380.3
60,92,115,243.0
45,90,112,180.1
60,101,124,299.0
60,93,113,223.0
60,107,136,361.0
60,114,140,415.0
60,102,127,300.5
60,100,120,300.1
```

60,100,120,300.0
45,104,129,266.0
45,90,112,180.1
60,98,126,286.0
60,100,122,329.4
60,111,138,400.0
60,111,131,397.0
60,99,119,273.0
60,109,153,387.6
45,111,136,300.0
45,108,129,298.0
60,111,139,397.6
60,107,136,380.2
80,123,146,643.1
60,106,130,263.0
60,118,151,486.0
30,136,175,238.0
60,121,146,450.7
60,118,121,413.0
45,115,144,305.0
20,153,172,226.4
45,123,152,321.0
210,108,160,1376.0
160,110,137,1034.4
160,109,135,853.0
45,118,141,341.0
20,110,130,131.4
180,90,130,800.4
150,105,135,873.4
150,107,130,816.0
20,106,136,110.4
300,108,143,1500.2
150,97,129,1115.0
60,109,153,387.6
90,100,127,700.0
150,97,127,953.2
45,114,146,304.0
90,98,125,563.2
45,105,134,251.0
45,110,141,300.0
120,100,130,500.4
270,100,131,1729.0
30,159,182,319.2

```
45,149,169,344.0
30,103,139,151.1
120,100,130,500.0
45,100,120,225.3
30,151,170,300.1
45,102,136,234.0
120,100,157,1000.1
45,129,103,242.0
20,83,107,50.3
180,101,127,600.1
45,107,137,
30,90,107,105.3
15,80,100,50.5
20,150,171,127.4
20,151,168,229.4
30,95,128,128.2
25,152,168,244.2
30,109,131,188.2
90,93,124,604.1
20,95,112,77.7
90,90,110,500.0
90,90,100,500.0
90,90,100,500.4
30,92,108,92.7
30,93,128,124.0
180,90,120,800.3
30,90,120,86.2
90,90,120,500.3
210,137,184,1860.4
60,102,124,325.2
45,107,124,275.0
15,124,139,124.2
45,100,120,225.3
60,108,131,367.6
60,108,151,351.7
60,116,141,443.0
60,97,122,277.4
60,105,125,
60,103,124,332.7
30,112,137,193.9
45,100,120,100.7
60,119,169,336.7
60,107,127,344.9
```

60,111,151,368.5
60,98,122,271.0
60,97,124,275.3
60,109,127,382.0
90,99,125,466.4
60,114,151,384.0
60,104,134,342.5
60,107,138,357.5
60,103,133,335.0
60,106,132,327.5
60,103,136,339.0
20,136,156,189.0
45,117,143,317.7
45,115,137,318.0
45,113,138,308.0
20,141,162,222.4
60,108,135,390.0
60,97,127,
45,100,120,250.4
45,122,149,335.4
60,136,170,470.2
45,106,126,270.8
60,107,136,400.0
60,112,146,361.9
30,103,127,185.0
60,110,150,409.4
60,106,134,343.0
60,109,129,353.2
60,109,138,374.0
30,150,167,275.8
60,105,128,328.0
60,111,151,368.5
60,97,131,270.4
60,100,120,270.4
60,114,150,382.8
30,80,120,240.9
30,85,120,250.4
45,90,130,260.4
45,95,130,270.0
45,100,140,280.9
60,105,140,290.8
60,110,145,300.4
60,115,145,310.2

```
75,120,150,320.4
75,125,150,330.4
```

Read CSV

Read CSV Files

A simple way to store big data sets is to use CSV files (comma separated files).

CSV files contains plain text and is a well know format that can be read by everyone including Pandas.

In our examples we will be using a CSV file called 'data.csv'.

Download data.csv. or Open data.csv

Load the CSV into a DataFrame:

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.to_string())
```

use `to_string()` to print the entire DataFrame.
If you have a large DataFrame with many rows, Pandas will only return the first 5 rows, and the last 5 rows:

Print the DataFrame without the to_string() method:

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df)
```

max_rows

The number of rows returned is defined in Pandas option settings.

You can check your system's maximum rows with the pd.options.display.max_rows statement.

```
import pandas as pd

print(pd.options.display.max_rows)
```

Pandas Read JSON

Read JSON

Big data sets are often stored, or extracted as JSON.

JSON is plain text, but has the format of an object, and is well known in the world of programming, including Pandas.

In our examples we will be using a JSON file called 'data.json'

https://www.w3schools.com/python/pandas/data.js

```
import pandas as pd

df = pd.read_json('data.json')

print(df.to_string())
```

JSON = Python Dictionary

Analyzing DataFrames

Viewing the Data

One of the most used method for getting a quick overview of the DataFrame, is the `head()` method.

The `head()` method returns the headers and a specified number of rows, starting from the top.

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.head(10))
```

Note: if the number of rows is not specified, the `head()` method will return the top 5 rows.

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.head())
```

There is also a `tail()` method for viewing the *last* rows of the DataFrame.

The `tail()` method returns the headers and a specified number of rows, starting from the bottom.

Print the last 5 rows of the DataFrame:

```
print(df.tail())
```

Info About the Data

The DataFrames object has a method called `info()`, that gives you more information about the data set.

```
print(df.info())
```

NumPy

What is NumPy?

NumPy is a Python library used for working with arrays.

It also has functions for working in domain of linear algebra, fourier transform, and matrices.

NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.

NumPy stands for Numerical Python.

Why Use NumPy?

In Python we have lists that serve the purpose of arrays, but they are slow to process.

NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

The array object in NumPy is called `ndarray`, it provides a lot of supporting functions that make working with `ndarray` very easy.

Arrays are very frequently used in data science, where speed and resources are very important.

Data Science: is a branch of computer science where we study how to store, use and analyze data for deriving information from it.

Why is NumPy Faster Than Lists?

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

This behavior is called locality of reference in computer science.

This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

Which Language is NumPy written in?

NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

Installation of NumPy

```
pip install numpy
```

Import NumPy

```
import numpy

arr = numpy.array([1, 2, 3, 4, 5])

print(arr)
```

NumPy as np

NumPy is usually imported under the np alias.

Create an alias with the as keyword while importing:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)

print(np.__version__)
```

NumPy Creating Arrays

Create a NumPy ndarray Object

NumPy is used to work with arrays. The array object in NumPy is called ndarray.

We can create a NumPy ndarray object by using the array() function.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)

print(type(arr))
```

type(): This built-in Python function tells us the type of the object passed to it. Like in above code it shows that arr is numpy.ndarray type.

To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:

```
import numpy as np

arr = np.array((1, 2, 3, 4, 5))

print(arr)
```

Dimensions in Arrays

A dimension in arrays is one level of array depth (nested arrays).
nested array: are arrays that have arrays as their elements.

0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

```
import numpy as np

arr = np.array(42)

print(arr)
```

1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

These are the most common and basic arrays.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)
```

2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.

These are often used to represent matrix or 2nd order tensors.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
```

3-D arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array.

These are often used to represent a 3rd order tensor.

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(arr)
```

Check Number of Dimensions?

NumPy Arrays provides the `ndim` attribute that returns an integer that tells us how many dimensions the array have.

```
import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

NumPy Array Indexing

Access Array Elements

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr[0])
```

Get third and fourth elements from the following array and add them.

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr[2] + arr[3])
```

Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('2nd element on 1st row: ', arr[0, 1])
```

Access the element on the 2nd row, 5th column:

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('5th element on 2nd row: ', arr[1, 4])
```

Access 3-D Arrays

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

print(arr[0, 1, 2])
```

Negative Indexing

Use negative indexing to access an array from the end.

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('Last element from 2nd dim: ', arr[1, -1])
```

Slicing arrays

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: [start:end].

We can also define the step, like this: [start:end:step].

If we don't pass start its considered 0

If we don't pass end its considered length of array in that dimension

If we don't pass step its considered 1

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
```

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[4:])
```

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[:4])
```

Negative Slicing

Use the minus operator to refer to an index from the end:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[-3:-1])
```

STEP

Use the `step` value to determine the step of the slicing:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5:2])
```

Slicing 2-D Arrays

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[1, 1:4])
```

From both elements, return index 2:

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[0:2, 2])
```

From both elements, slice index 1 to index 4 (not included), this will return a 2-D array:

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[0:2, 1:4])
```

NumPy Data Types

Data Types in Python

By default Python have these data types:

- strings - used to represent text data, the text is given under quote marks. e.g. "ABCD"
- integer - used to represent integer numbers. e.g. -1, -2, -3
- float - used to represent real numbers. e.g. 1.2, 42.42
- `boolean` - used to represent True or False.
- `complex` - used to represent complex numbers. e.g. 1.0 + 2.0j, 1.5 + 2.5j

Data Types in NumPy

- `i` - integer
- `b` - boolean
- `u` - unsigned integer
- `f` - float
- `c` - complex float

- `m` - timedelta
- `M` - datetime
- `O` - object
- `S` - string
- `U` - unicode string
- `V` - fixed chunk of memory for other type (void)

Checking the Data Type of an Array

The NumPy array object has a property called `dtype` that returns the data type of the array:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr.dtype)
```

The Difference Between Copy and View

The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.

The copy *owns* the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.

The view *does not own* the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.

COPY:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 42
print(arr)
print(x)
```

The copy SHOULD NOT be affected by the changes made to the original array.

VIEW

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
arr[0] = 42
print(arr)
print(x)
```

Make a view, change the view, and display both arrays:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
x[0] = 31
print(arr)
print(x)
```

Check if Array Owns its Data

As mentioned above, copies *owns* the data, and views *does not own* the data, but how can we check this?

Every NumPy array has the attribute `base` that returns `None` if the array owns the data.

Otherwise, the `base` attribute refers to the original object.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
y = arr.view()
print(x.base)
print(y.base)
```