What is pandas?

pandas is a data manipulation package in Python for tabular data. That is, data in the form of rows and columns, also known as DataFrames. Intuitively, you can think of a DataFrame as an Excel sheet.

pandas' functionality includes data transformations, like **sorting rows** and taking subsets, to calculating summary statistics such as the mean, reshaping DataFrames, and joining DataFrames together. pandas works well with other popular Python data science packages, often called the PyData ecosystem, including

- **NumPy** for numerical computing
- **Matplotlib**, **Seaborn**, **Plotly**, and other data visualization packages
- **scikit-learn** for machine learning

What is pandas used for?

pandas is used throughout the data analysis workflow. With pandas, you can:

- Import datasets from databases, spreadsheets, comma-separated values (CSV) files, and more.
- Clean datasets, for example, by dealing with missing values.
- Tidy datasets by reshaping their structure into a suitable format for analysis.
- Aggregate data by calculating summary statistics such as the mean of columns, correlation between them, and more.
- Visualize datasets and uncover insights.

pandas also contains functionality for time series analysis and analyzing text data.

Key benefits of the pandas package

Undoubtedly, pandas is a powerful data manipulation tool packaged with several benefits, including:

- **Made for Python:** Python is the world's most popular language for machine learning and data science.

- **Less verbose per unit operations:** Code written in pandas is less verbose, requiring fewer lines of code to get the desired output.
- **Intuitive view of data:** pandas offers exceptionally intuitive data representation that facilitates easier data understanding and analysis.
- **Extensive feature set:** It supports an extensive set of operations from exploratory data analysis, dealing with missing values, calculating statistics, visualizing univariate and bivariate data, and much more.
- **Works with large data:** pandas handles large data sets with ease. It offers speed and efficiency while working with datasets of the order of millions of records and hundreds of columns, depending on the machine.

How to install pandas?

Before delving into its functionality, let us first install pandas. You can avoid this step by **registering for a free DataCamp account** and using **DataLab**, DataLab cloud-based IDE that comes with pandas (alongside the top python data science packages) pre-installed.

Run and edit the code from this tutorial online

Run code

Install pandas

Installing pandas is straightforward; just use the `pip install` command in your terminal.

```
pip install pandas
```

Run code

POWERED BY

Importing data in pandas

To begin working with pandas, import the pandas Python package as shown below. When importing pandas, the most common alias for pandas is `pd`.

```
import pandas as pd
```

Importing CSV files

Use `read_csv()` with the path to the CSV file to read a comma-separated values file (see our **tutorial on importing data with read_csv()** for more detail).

```
df = pd.read_csv("diabetes.csv")
```

This read operation loads the CSV file `diabetes.csv` to generate a pandas Dataframe object `df`. Throughout this tutorial, you'll see how to manipulate such DataFrame objects.

Importing text files

Reading text files is similar to CSV files. The only nuance is that you need to specify a separator with the `sep` argument, as shown below. The separator argument refers to the symbol used to separate rows in a DataFrame. Comma (`sep = ","`), whitespace(`sep = "\s"`), tab (`sep = "\t"`), and colon(`sep = ":"`) are the commonly used separators. Here `\s` represents a single white space character.

```
df = pd.read_csv("diabetes.txt", sep="\s")
```

Importing Excel files (single sheet)

Reading excel files (both XLS and XLSX) is as easy as the `read_excel()` function, using the file path as an input.

```
df = pd.read_excel('diabetes.xlsx')
```

You can also specify other arguments, such as `header` for to specify which row becomes the DataFrame's header. It has a default value of `0`, which denotes the first row as headers or column names. You can also specify column names as a list in the `names` argument. The `index_col` (default is `None`) argument can be used if the file contains a row index.

Note: *In a pandas DataFrame or Series, the index is an identifier that points to the location of a row or column in a pandas DataFrame. In a nutshell, the index labels the row or column of a DataFrame and lets you access a specific row or column by using its index (you will see this later on). A DataFrame's row index can be a range (e.g., 0 to 303), a time series (dates or timestamps), a unique identifier (e.g.,* `employee_ID` *in an* `employees` *table), or other types of data. For columns, it's usually a string (denoting the column name).*

Importing Excel files (multiple sheets)

Reading Excel files with multiple sheets is not that different. You just need to specify one additional argument, `sheet_name`, where you can either pass a string for the sheet name or an integer for the sheet position (note that Python uses 0-indexing, where the first sheet can be accessed with `sheet_name = 0`)

```
# Extracting the second sheet since Python uses 0-indexing

df = pd.read_excel('diabetes_multi.xlsx', sheet_name=1)
```

Importing JSON file

Similar to the `read_csv()` function, you can use `read_json()` for JSON file types with the JSON file name as the argument (for more detail read **this tutorial on importing JSON and HTML data into pandas**). The below code reads a JSON file from disk and creates a DataFrame object `df`.

```
df = pd.read_json("diabetes.json")
```

Outputting data in pandas

Just as pandas can import data from various file types, it also allows you to export data into various formats. This happens especially when data is transformed using pandas and needs to be saved locally on your machine. Below is how to output pandas DataFrames into various formats.

Outputting a DataFrame into a CSV file

A pandas DataFrame (here we are using `df`) is saved as a CSV file using the `.to_csv()` method. The arguments include the filename with path and `index` – where `index = True` implies writing the DataFrame's index.

```
df.to_csv("diabetes_out.csv", index=False)
```

This code saves a pandas DataFrame df to a CSV file named "diabetes_out.csv" in the current working directory. The to_csv() method is used to write the DataFrame to a CSV file. The index=False argument specifies that the index column should not be included in the output file. This is useful when the index is not meaningful or when it is already included as a separate column in the DataFrame.

Outputting a DataFrame into a JSON file

Export DataFrame object into a JSON file by calling the `.to_json()` method.

```
df.to_json("diabetes_out.json")
```

This code saves a pandas DataFrame object df as a JSON file named "diabetes_out.json". The to_json() method is a built-in function in pandas that converts a DataFrame to a JSON string. By passing the file name as an argument, the method saves the JSON string to a file with the specified name. This can be useful for storing data in a format that can be easily shared or used by other programs.

Note: *A JSON file stores a tabular object like a DataFrame as a key-value pair. Thus you would observe repeating column headers in a JSON file.*

Outputting a DataFrame into a text file

As with writing DataFrames to CSV files, you can call .to_csv(). The only differences are that the output file format is in .txt, and you need to specify a separator using the sep argument.

```
df.to_csv('diabetes_out.txt', header=df.columns, index=None, sep=' ')
```

This code exports a pandas DataFrame df to a text file named "diabetes_out.txt" using the to_csv() method.

The header parameter is set to df.columns, which means that the column names of the DataFrame will be included as the first row in the output file.

The index parameter is set to None, which means that the row index of the DataFrame will not be included in the output file.

The sep parameter is set to a space character, which means that the values in each row will be separated by a space in the output file.

Overall, this code exports the DataFrame df to a text file with column names as the first row and values separated by spaces.

Outputting a DataFrame into an Excel file

Call .to_excel() from the DataFrame object to save it as a ".xls" or ".xlsx" file.

```
df.to_excel("diabetes_out.xlsx", index=False)
```

This code uses the pandas library in Python to export a DataFrame object df to an Excel file named "diabetes_out.xlsx". The to_excel() method is called on the DataFrame object and takes two arguments: the file name to save the Excel file as, and a boolean value index which is set to False to exclude the index column from being exported to the Excel file. This code will create a new Excel file in the current working directory with the data from the DataFrame object.

Viewing and understanding DataFrames using pandas

After reading tabular data as a DataFrame, you would need to have a glimpse of the data. You can either view a small sample of the dataset or a summary of the data in the form of summary statistics.

How to view data using .head() and .tail()

You can view the first few or last few rows of a DataFrame using the .head() or .tail() methods, respectively. You can specify the number of rows through the n argument (the default value is 5).

```
df.head()
```

This code is written in Python and it calls the head() method on a Pandas DataFrame object named df. The head() method is used to display the first few rows of the DataFrame. By default, it displays the first 5 rows, but you can pass an integer argument to display a different number of rows. This code is useful for quickly inspecting the contents of a DataFrame and getting a sense of what kind of data it contains.

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

First five rows of the DataFrame

```
df.tail(n = 10)
```

This code is written in Python and it uses the tail() method to display the last 10 rows of a DataFrame df. The n parameter is set to 10 to specify the number of rows to display. The tail() method is commonly used to quickly check the last few rows of a DataFrame to ensure that the data has been loaded correctly or to get a quick overview of the data.

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outco
758	1	106	76	0	0	37.5	0.197	26	
759	6	190	92	0	0	35.5	0.278	66	
760	2	88	58	26	16	28.4	0.766	22	
761	9	170	74	31	0	44.0	0.403	43	
762	9	89	62	0	0	22.5	0.142	33	
763	10	101	76	48	180	32.9	0.171	63	
764	2	122	70	27	0	36.8	0.340	27	
765	5	121	72	23	112	26.2	0.245	30	
766	1	126	60	0	0	30.1	0.349	47	
767	1	93	70	31	0	30.4	0.315	23	

First 10 rows of the DataFrame

Understanding data using `.describe()`

The .describe() method prints the summary statistics of all numeric columns, such as count, mean, standard deviation, range, and quartiles of numeric columns.

```
df.describe()
```

This code is written in Python and it calls the describe() method on a Pandas DataFrame object named df. The describe() method generates descriptive statistics of the DataFrame, including count, mean, standard deviation, minimum, maximum, and quartile values for each column. This method is useful for quickly understanding the distribution of data in a DataFrame.

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunctio
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.00000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.47187
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.33132
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.07800
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.24375
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.37250
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.62625
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.42000

Get summary statistics with .describe()

It gives a quick look at the scale, skew, and range of numeric data.

You can also modify the quartiles using the percentiles argument. Here, for example, we're looking at the 30%, 50%, and 70% percentiles of the numeric columns in DataFrame df.

```
df.describe(percentiles=[0.3, 0.5, 0.7])
```

This code is written in Python and it uses the describe() method to generate descriptive statistics of a DataFrame. The percentiles parameter is used to specify the percentiles to include in the output. In this case, the percentiles 0.3, 0.5, and 0.7 are specified. The output will include the count, mean, standard deviation, minimum, 30th percentile, 50th percentile (median), 70th percentile, and maximum values for each column in the DataFrame.

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunctio
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.00000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.47187
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.33132
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.07800
30%	1.000000	102.000000	64.000000	8.200000	0.000000	28.200000	0.25900
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.37250
70%	5.000000	134.000000	78.000000	31.000000	106.000000	35.490000	0.56370
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.42000

Get summary statistics with specific percentiles

You can also isolate specific data types in your summary output by using the `include` argument. Here, for example, we're only summarizing the columns with the `integer` data type.

```
df.describe(include=[int])
```

This code is written in Python and it uses the describe() method to generate descriptive statistics of a DataFrame. The include parameter is used to specify the data types to be included in the output. In this case, it includes only integer columns in the output.

So, df.describe(include=[int]) will generate descriptive statistics of only the integer columns in the DataFrame df. This includes the count, mean, standard deviation, minimum, maximum, and quartile values of the integer columns.

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	81.000000	1.000000

Get summary statistics of integer columns only

Similarly, you might want to exclude certain data types using `exclude` argument.

```
df.describe(exclude=[int])
```

This code is written in Python and it uses the describe() method of a Pandas DataFrame object to generate descriptive statistics of the data in the DataFrame. The exclude parameter is used to exclude certain data types from the analysis. In this case, the exclude=[int] parameter is used to exclude integer columns from the analysis. This means that the describe() method will only generate statistics for non-integer columns in the DataFrame.

	BMI	DiabetesPedigreeFunction
count	768.000000	768.000000
mean	31.992578	0.471876
std	7.884160	0.331329
min	0.000000	0.078000
25%	27.300000	0.243750
50%	32.000000	0.372500
75%	36.600000	0.626250
max	67.100000	2.420000

Get summary statistics of non-integer columns only

Often, practitioners find it easy to view such statistics by transposing them with the `.T` attribute.

```
df.describe().T
```

This code uses the describe() method to generate summary statistics of a pandas DataFrame df. The T attribute is then used to transpose the resulting summary statistics table, so that the rows become columns and vice versa. This makes it easier to read and compare the statistics for different columns.

For example, if df has columns for "age", "income", and "education", the resulting table will have rows for "count", "mean", "std", "min", "25%", "50%", "75%", and "max", and columns for "age", "income", and "education".

Overall, this code is useful for quickly getting an overview of the distribution and range of values in a DataFrame.

	count	mean	std	min	25%	50%	75%	max
Pregnancies	768.0	3.845052	3.369578	0.000	1.00000	3.0000	6.00000	17.00
Glucose	768.0	120.894531	31.972618	0.000	99.00000	117.0000	140.25000	199.00
BloodPressure	768.0	69.105469	19.355807	0.000	62.00000	72.0000	80.00000	122.00
SkinThickness	768.0	20.536458	15.952218	0.000	0.00000	23.0000	32.00000	99.00
Insulin	768.0	79.799479	115.244002	0.000	0.00000	30.5000	127.25000	846.00
BMI	768.0	31.992578	7.884160	0.000	27.30000	32.0000	36.60000	67.10
DiabetesPedigreeFunction	768.0	0.471876	0.331329	0.078	0.24375	0.3725	0.62625	2.42
Age	768.0	33.240885	11.760232	21.000	24.00000	29.0000	41.00000	81.00
Outcome	768.0	0.348958	0.476951	0.000	0.00000	0.0000	1.00000	1.00

Transpose summary statistics with `.T`

Understanding data using `.info()`

The .info() method is a quick way to look at the data types, missing values, and data size of a DataFrame. Here, we're setting the show_counts argument to True, which gives a few over the total non-missing values in each column. We're also setting memory_usage to True, which shows the total memory usage of the DataFrame elements. When verbose is set to True, it prints the full summary from .info().

```
df.info(show_counts=True, memory_usage=True, verbose=True)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               768 non-null    int64
 1   Glucose                   768 non-null    int64
 2   BloodPressure             768 non-null    int64
 3   SkinThickness             768 non-null    int64
 4   Insulin                   768 non-null    int64
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64
 8   Outcome                   768 non-null    int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

Understanding your data using `.shape`

The number of rows and columns of a DataFrame can be identified using the .shape attribute of the DataFrame. It returns a tuple (row, column) and can be indexed to get only rows, and only columns count as output.

```
df.shape # Get the number of rows and columns

df.shape[0] # Get the number of rows only

df.shape[1] # Get the number of columns only
```

POWERED BY

```
(768,9)

768
```

```
9
```

Get all columns and column names

Calling the `.columns` attribute of a DataFrame object returns the column names in the form of an `Index` object. As a reminder, a pandas index is the address/label of the row or column.

```
df.columns
```

```
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
      dtype='object')
```

It can be converted to a list using a `list()` function.

```
list(df.columns)
```

```
['Pregnancies',
 'Glucose',
 'BloodPressure',
 'SkinThickness',
 'Insulin',
 'BMI',
 'DiabetesPedigreeFunction',
 'Age',
 'Outcome']
```

Checking for missing values in pandas with `.isnull()`

The sample DataFrame does not have any missing values. Let's introduce a few to make things interesting. The `.copy()` method makes a copy of the original DataFrame. This is done to ensure that any changes to the copy don't reflect in the original DataFrame. Using `.loc` (to be discussed later), you can set rows two to five of the `Pregnancies` column to `NaN` values, which denote missing values.

```
df2 = df.copy()

df2.loc[2:5,'Pregnancies'] = None

df2.head(7)
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6.0	148	72	35	0	33.6	0.627	50	1
1	1.0	85	66	29	0	26.6	0.351	31	0
2	NaN	183	64	0	0	23.3	0.672	32	1
3	NaN	89	66	23	94	28.1	0.167	21	0
4	NaN	137	40	35	168	43.1	2.288	33	1
5	NaN	116	74	0	0	25.6	0.201	30	0
6	3.0	78	50	32	88	31.0	0.248	26	1

You can see, that now rows 2 to 5 are NaN

You can check whether each element in a DataFrame is missing using the .isnull() method.

```
df2.isnull().head(7)
```

Given it's often more useful to know how much missing data you have, you can combine .isnull() with .sum() to count the number of nulls in each column.

```
df2.isnull().sum()
```

POWERED BY

```
Pregnancies              4
Glucose                  0
BloodPressure            0
SkinThickness            0
Insulin                  0
BMI                      0
DiabetesPedigreeFunction 0
Age                      0
Outcome                  0
dtype: int64
```

You can also do a double sum to get the total number of nulls in the DataFrame.

```
df2.isnull().sum().sum()

4
```

Slicing and Extracting Data in pandas

The pandas package offers several ways to subset, filter, and isolate data in your DataFrames. Here, we'll see the most common ways.

Isolating one column using `[ ]`

You can isolate a single column using a square bracket `[]` with a column name in it. The output is a pandas `Series` object. A pandas Series is a one-dimensional array containing data of any type, including integer, float, string, boolean, python objects, etc. A DataFrame is comprised of many series that act as columns.

```
df['Outcome']
```

```
0      1
1      0
2      1
3      0
4      1
      ..
763    0
764    0
765    0
766    1
767    0
Name: Outcome, Length: 768, dtype: int64
```

Isolating one column in pandas

Isolating two or more columns using `[[ ]]`

You can also provide a list of column names inside the square brackets to fetch more than one column. Here, square brackets are used in two different ways. We use the outer square brackets to indicate a subset of a DataFrame, and the inner square brackets to create a list.

```
df[['Pregnancies', 'Outcome']]
```

	Pregnancies	Outcome
0	6	1
1	1	0
2	8	1
3	1	0
4	0	1
...	...	...
763	10	0
764	2	0
765	5	0
766	1	1
767	1	0

768 rows × 2 columns

Isolating two columns in pandas

Isolating one row using `[ ]`

A single row can be fetched by passing in a boolean series with one `True` value. In the example below, the second row with `index = 1` is returned.
Here, `.index` returns the row labels of the DataFrame, and the comparison turns that into a Boolean one-dimensional array.

```
df[df.index==1]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
1	1	85	66	29	0	26.6	0.351	31	0

Isolating one row in pandas

Isolating two or more rows using `[ ]`

Similarly, two or more rows can be returned using the `.isin()` method instead of a `==` operator.

```
df[df.index.isin(range(2,10))]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
5	5	116	74	0	0	25.6	0.201	30	0
6	3	78	50	32	88	31.0	0.248	26	1
7	10	115	0	0	0	35.3	0.134	29	0
8	2	197	70	45	543	30.5	0.158	53	1
9	8	125	96	0	0	0.0	0.232	54	1

Isolating specific rows in pandas

Using `.loc[]` and `.iloc[]` to fetch rows

You can fetch specific rows by labels or conditions using `.loc[]` and `.iloc[]` ("location" and "integer location"). `.loc[]` uses a label to point to a row, column or cell, whereas `.iloc[]` uses the numeric position. To understand the difference between the two, let's modify the index of `df2` created earlier.

```
df2.index = range(1,769)
```

The below example returns a pandas `Series` instead of a DataFrame. The `1` represents the row index (label), whereas the `1` in `.iloc[]` is the row position (first row).

```
df2.loc[1]

Pregnancies              6.000

Glucose                148.000

BloodPressure             72.000

SkinThickness             35.000

Insulin                  0.000

BMI                     33.600

DiabetesPedigreeFunction     0.627
```

```
Age                         50.000

Outcome                      1.000

Name: 1, dtype: float64

df2.iloc[1]

Pregnancies                  1.000

Glucose                     85.000

BloodPressure               66.000

SkinThickness               29.000

Insulin                      0.000

BMI                         26.600

DiabetesPedigreeFunction     0.351

Age                         31.000

Outcome                      0.000

Name: 2, dtype: float64
```

You can also fetch multiple rows by providing a range in square brackets.

```
df2.loc[100:110]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcon
100	1.0	122	90	51	220	49.7	0.325	31	
101	1.0	163	72	0	0	39.0	1.222	33	
102	1.0	151	60	0	0	26.1	0.179	22	
103	0.0	125	96	0	0	22.5	0.262	21	
104	1.0	81	72	18	40	26.6	0.283	24	
105	2.0	85	65	0	0	39.6	0.930	27	
106	1.0	126	56	29	152	28.7	0.801	21	
107	1.0	96	122	0	0	22.4	0.207	27	
108	4.0	144	58	28	140	29.5	0.287	37	
109	3.0	83	58	31	18	34.3	0.336	25	
110	0.0	95	85	25	36	37.4	0.247	24	

Isolating rows in pandas with `.loc[]`

```
df2.iloc[100:110]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcon
101	1.0	163	72	0	0	39.0	1.222	33	
102	1.0	151	60	0	0	26.1	0.179	22	
103	0.0	125	96	0	0	22.5	0.262	21	
104	1.0	81	72	18	40	26.6	0.283	24	
105	2.0	85	65	0	0	39.6	0.930	27	
106	1.0	126	56	29	152	28.7	0.801	21	
107	1.0	96	122	0	0	22.4	0.207	27	
108	4.0	144	58	28	140	29.5	0.287	37	
109	3.0	83	58	31	18	34.3	0.336	25	
110	0.0	95	85	25	36	37.4	0.247	24	

Isolating rows in pandas with `.iloc[]`

You can also subset with `.loc[]` and `.iloc[]` by using a list instead of a range.

```
df2.loc[[100, 200, 300]]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcor
100	1.0	122	90	51	220	49.7	0.325	31	
200	4.0	148	60	27	318	30.9	0.150	29	
300	8.0	112	72	0	0	23.6	0.840	58	

Isolating rows using a list in pandas with `.loc[]`

```
df2.iloc[[100, 200, 300]]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcon
101	1.0	163	72	0	0	39.0	1.222	33	
201	0.0	113	80	16	0	31.0	0.874	21	
301	0.0	167	0	0	0	32.3	0.839	30	

Isolating rows using a list in pandas with `.iloc[]`

You can also select specific columns along with rows. This is where `.iloc[]` is different from `.loc[]` – it requires column location and not column labels.

```
df2.loc[100:110, ['Pregnancies', 'Glucose', 'BloodPressure']]
```

	Pregnancies	Glucose	BloodPressure
100	1.0	122	90
101	1.0	163	72
102	1.0	151	60
103	0.0	125	96
104	1.0	81	72
105	2.0	85	65
106	1.0	126	56
107	1.0	96	122
108	4.0	144	58
109	3.0	83	58
110	0.0	95	85

Isolating columns in pandas with `.loc[]`

```
df2.iloc[100:110, :3]
```

	Pregnancies	Glucose	BloodPressure
101	1.0	163	72
102	1.0	151	60
103	0.0	125	96
104	1.0	81	72
105	2.0	85	65
106	1.0	126	56
107	1.0	96	122
108	4.0	144	58
109	3.0	83	58
110	0.0	95	85

Isolating columns with `.iloc[]`

For faster workflows, you can pass in the starting index of a row as a range.

```
df2.loc[760:, ['Pregnancies', 'Glucose', 'BloodPressure']]
```

	Pregnancies	Glucose	BloodPressure
760	6.0	190	92
761	2.0	88	58
762	9.0	170	74
763	9.0	89	62
764	10.0	101	76
765	2.0	122	70
766	5.0	121	72
767	1.0	126	60
768	1.0	93	70

Isolating columns and rows in pandas with `.loc[]`

```
df2.iloc[760:, :3]
```

	Pregnancies	Glucose	BloodPressure
761	2.0	88	58
762	9.0	170	74
763	9.0	89	62
764	10.0	101	76
765	2.0	122	70
766	5.0	121	72
767	1.0	126	60
768	1.0	93	70

Isolating columns and rows in pandas with `.iloc[]`

You can update/modify certain values by using the assignment operator `=`

```
df2.loc[df['Age']==81, ['Age']] = 80
```

Conditional slicing (that fits certain conditions)

pandas lets you filter data by conditions over row/column values. For example, the below code selects the row where Blood Pressure is exactly 122. Here, we are isolating rows using the brackets `[]` as seen in previous sections. However, instead of inputting row indices or column names, we are inputting a condition where the column `BloodPressure` is equal to 122. We denote this condition using `df.BloodPressure == 122`.

```
df[df.BloodPressure == 122]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outco
106	1	96	122	0	0	22.4	0.207	27	

Isolating rows based on a condition in pandas

The below example fetched all rows where `Outcome` is 1. Here `df.Outcome` selects that column, `df.Outcome == 1` returns a Series of Boolean values determining which `Outcomes` are equal to 1, then `[]` takes a subset of `df` where that Boolean Series is `True`.

```
df[df.Outcome == 1]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outco
0	6	148	72	35	0	33.6	0.627	50	
2	8	183	64	0	0	23.3	0.672	32	
4	0	137	40	35	168	43.1	2.288	33	
6	3	78	50	32	88	31.0	0.248	26	
8	2	197	70	45	543	30.5	0.158	53	
...	...	...	...	...	...	...	...	...	
755	1	128	88	39	110	36.5	1.057	37	
757	0	123	72	0	0	36.3	0.258	52	
759	6	190	92	0	0	35.5	0.278	66	
761	9	170	74	31	0	44.0	0.403	43	
766	1	126	60	0	0	30.1	0.349	47	

268 rows × 9 columns

Isolating rows based on a condition in pandas

You can use a `>` operator to draw comparisons. The below code fetches `Pregnancies`, `Glucose`, and `BloodPressure` for all records with `BloodPressure` greater than 100.

```
df.loc[df['BloodPressure'] > 100, ['Pregnancies', 'Glucose', 'BloodPressure']]
```

	Pregnancies	Glucose	BloodPressure
43	9	171	110
84	5	137	108
106	1	96	122
177	0	129	110
207	5	162	104
362	5	103	108
369	1	133	102
440	0	189	104
549	4	189	110
658	11	127	106
662	8	167	106
672	10	68	106
691	13	158	114

Cleaning data using pandas

Data cleaning is one of the most common tasks in data science. pandas lets you preprocess data for any use, including but not limited to training machine learning and deep learning models. Let's use the DataFrame `df2` from earlier, having four missing values, to illustrate a few data cleaning use cases. As a reminder, here's how you can see how many missing values are in a DataFrame.

```
df2.isnull().sum()

Pregnancies              4
Glucose                  0
BloodPressure            0
SkinThickness            0
Insulin                  0
BMI                      0
DiabetesPedigreeFunction 0
Age                      0
Outcome                  0
dtype: int64
```

Dealing with missing data technique #1: Dropping missing values

One way to deal with missing data is to drop it. This is particularly useful in cases where you have plenty of data and losing a small portion won't impact the downstream analysis. You can use a `.dropna()` method as shown below. Here, we are saving the results from `.dropna()` into a DataFrame `df3`.

```
df3 = df2.copy()

df3 = df3.dropna()
```

```
df3.shape

(764, 9) # this is 4 rows less than df2
```

The `axis` argument lets you specify whether you are dropping rows, or **[columns](#)**, with missing values. The default `axis` removes the rows containing NaNs. Use `axis = 1` to remove the columns with one or more NaN values. Also, notice how we are using the argument `inplace=True` which lets you skip saving the output of `.dropna()` into a new DataFrame.

```
df3 = df2.copy()

df3.dropna(inplace=True, axis=1)

df3.head()
```

	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DPF	Age	Outcome	STF
0	148	72	35	0	33.6	0.627	50	1	0.700000
1	85	66	29	0	26.6	0.351	31	0	0.935484
2	183	64	0	0	23.3	0.672	32	1	0.000000
3	89	66	23	94	28.1	0.167	21	0	1.095238
4	137	40	35	168	43.1	2.288	33	1	1.060606

Dropping missing data in pandas

You can also drop both rows and columns with missing values by setting the `how` argument to `'all'`

```
df3 = df2.copy()

df3.dropna(inplace=True, how='all')
```

Dealing with missing data technique #2: Replacing missing values

Instead of dropping, replacing missing values with a summary statistic or a specific value (depending on the use case) maybe the best way to go. For example, if there is one missing row from a temperature column denoting temperatures throughout the days of the week, replacing that missing value with the average temperature of that week may be more effective than dropping values completely. You can replace the missing data with the row, or column mean using the code below.

```
df3 = df2.copy()
```

```
# Get the mean of Pregnancies

mean_value = df3['Pregnancies'].mean()

# Fill missing values using .fillna()

df3 = df3.fillna(mean_value)
```

Dealing with Duplicate Data

Let's add some duplicates to the original data to learn how to eliminate duplicates in a DataFrame. Here, we are using the `.concat()` method to concatenate the rows of the `df2` DataFrame to the `df2` DataFrame, adding perfect duplicates of every row in `df2`.

```
df3 = pd.concat([df2, df2])

df3.shape

(1536, 9)
```

You can remove all duplicate rows (default) from the DataFrame using **.drop_duplicates()** method.

```
df3 = df3.drop_duplicates()

df3.shape

(768, 9)
```

Renaming columns

A common data cleaning task is renaming columns. With the `.rename()` method, you can use `columns` as an argument to rename specific columns. The below code shows the dictionary for mapping old and new column names.

```
df3.rename(columns = {'DiabetesPedigreeFunction':'DPF'}, inplace = True)

df3.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DPF	Age	Outcome
0	6.0	148	72	35	0	33.6	0.627	50	1
1	1.0	85	66	29	0	26.6	0.351	31	0
2	8.0	183	64	0	0	23.3	0.672	32	1
3	1.0	89	66	23	94	28.1	0.167	21	0
4	0.0	137	40	35	168	43.1	2.288	33	1

Renaming columns in pandas

You can also directly assign column names as a list to the DataFrame.

```
df3.columns = ['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI', 'DPF', 'Age', 'Outcome', 'STF']

df3.head()
```

	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	148	72	35	0	33.6	0.627	50	1
1	85	66	29	0	26.6	0.351	31	0
2	183	64	0	0	23.3	0.672	32	1
3	89	66	23	94	28.1	0.167	21	0
4	137	40	35	168	43.1	2.288	33	1

Renaming columns in pandas

For more on data cleaning, and for easier, more predictable data cleaning workflows, check out the following checklist, which provides you with a comprehensive set of common **data cleaning tasks**.

Data analysis in pandas

The main value proposition of pandas lies in its quick data analysis functionality. In this section, we'll focus on a set of analysis techniques you can use in pandas.

Summary operators (mean, mode, median)

As you saw earlier, you can get the mean of each column value using the `.mean()` method.

```
df.mean()
```

```
Pregnancies                   3.845052
Glucose                     120.894531
BloodPressure                69.105469
SkinThickness                20.536458
Insulin                      79.799479
BMI                          31.992578
DiabetesPedigreeFunction      0.471876
Age                          33.240885
Outcome                       0.348958
dtype: float64
```

Printing the mean of columns in pandas

A mode can be computed similarly using the `.mode()` method.

```
df.mode()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcom
0	1.0	99	70.0	0.0	0.0	32.0	0.254	22.0	0.(
1	NaN	100	NaN	NaN	NaN	NaN	0.258	NaN	Nal

Printing the mode of columns in pandas

Similarly, the median of each column is computed with the `.median()` method

```
df.median()
```

```
Pregnancies                   3.0000
Glucose                     117.0000
BloodPressure                72.0000
SkinThickness                23.0000
Insulin                      30.5000
BMI                          32.0000
DiabetesPedigreeFunction      0.3725
Age                          29.0000
Outcome                       0.0000
dtype: float64
```

Printing the median of columns in pandas

Create new columns based on existing columns

pandas provides fast and efficient computation by combining two or more columns like scalar variables. The below code divides each value in the column `Glucose` with the corresponding value in the `Insulin` column to compute a new column named `Glucose_Insulin_Ratio`.

```
df2['Glucose_Insulin_Ratio'] = df2['Glucose']/df2['Insulin']
```

```
df2.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
1	6.0	148	72	35	0	33.6	0.627	50	1
2	1.0	85	66	29	0	26.6	0.351	31	0
3	NaN	183	64	0	0	23.3	0.672	32	1
4	NaN	89	66	23	94	28.1	0.167	21	0
5	NaN	137	40	35	168	43.1	2.288	33	1

Create a new column from existing columns in pandas

Counting using `.value_counts()`

Often times you'll work with categorical values, and you'll want to count the number of observations each category has in a column. Category values can be counted using the `.value_counts()` methods. Here, for example, we are counting the number of observations where `Outcome` is diabetic (1) and the number of observations where the `Outcome` is non-diabetic (0).

```
df['Outcome'].value_counts()
```

```
0    500
1    268
Name: Outcome, dtype: int64
```

Using `.value_counts()` *in pandas*

Adding the `normalize` argument returns proportions instead of absolute counts.

```
df['Outcome'].value_counts(normalize=True)
```

```
0    0.651042
1    0.348958
Name: Outcome, dtype: float64
```

Using `.value_counts()` *in pandas with normalization*

Turn off automatic sorting of results using `sort` argument (`True` by default). The default sorting is based on the counts in descending order.

```
df['Outcome'].value_counts(sort=False)
```

```
1    268
0    500
Name: Outcome, dtype: int64
```

Using .value_counts() *in pandas with sorting*

You can also apply .value_counts() to a DataFrame object and specific columns within it instead of just a column. Here, for example, we are applying value_counts() on df with the subset argument, which takes in a list of columns.

```
df.value_counts(subset=['Pregnancies', 'Outcome'])
```

POWERED BY

```
Pregnancies  Outcome
1            0          106
2            0           84
0            0           73
3            0           48
4            0           45
0            1           38
5            0           36
6            0           34
1            1           29
3            1           27
7            1           25
4            1           23
8            1           22
5            1           21
7            0           20
2            1           19
9            1           18
6            1           16
8            0           16
10           0           14
             1           10
9            0           10
11           1            7
12           0            5
13           0            5
             1            5
11           0            4
12           1            4
14           1            2
15           1            1
17           1            1
dtype: int64
```

Using .value_counts() *in pandas while subsetting columns*

Aggregating data with `.groupby()` in pandas

pandas lets you aggregate values by grouping them by specific column values. You can do that by combining the `.groupby()` method with a summary method of your choice. The below code displays the mean of each of the numeric columns grouped by `Outcome`.

```
df.groupby('Outcome').mean()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunc
Outcome							
0	3.298000	109.980000	68.184000	19.664000	68.792000	30.304200	0.429
1	4.865672	141.257463	70.824627	22.164179	100.335821	35.142537	0.550

Aggregating data by one column in pandas

`.groupby()` enables grouping by more than one column by passing a list of column names, as shown below.

```
df.groupby(['Pregnancies', 'Outcome']).mean()
```

		Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunc
Pregnancies	**Outcome**						
0	**0**	111.945205	69.205479	21.054795	77.561644	31.727397	0.457
	1	144.236842	63.210526	24.605263	89.578947	39.213158	0.643
1	**0**	104.254717	66.830189	23.047170	84.320755	29.616038	0.451
	1	143.793103	71.310345	29.517241	151.137931	37.793103	0.613
2	**0**	105.214286	61.940476	20.107143	72.619048	29.679762	0.479
	1	135.473684	69.052632	28.210526	144.315789	34.578947	0.543
3	**0**	109.604167	65.708333	17.520833	62.020833	29.231250	0.358
	1	148.444444	68.148148	24.629630	132.666667	32.548148	0.563
4	**0**	117.555556	71.577778	18.422222	78.466667	31.255556	0.410
	1	139.913043	67.000000	10.913043	51.782609	33.873913	0.516
5	**0**	111.666667	74.666667	17.166667	46.861111	31.100000	0.359
	1	131.190476	78.857143	17.761905	75.190476	36.780952	0.460
6	**0**	115.352941	66.382353	18.705882	69.029412	29.591176	0.433
	1	132.375000	72.750000	15.375000	52.000000	31.775000	0.421
7	**0**	121.000000	70.350000	19.350000	72.500000	29.975000	0.405
	1	148.800000	71.120000	21.040000	94.040000	34.756000	0.474
8	**0**	106.625000	75.312500	12.937500	14.500000	30.693750	0.526
	1	150.000000	75.090909	20.500000	149.772727	32.204545	0.488
9	**0**	107.000000	70.400000	22.400000	71.200000	28.840000	0.311
	1	144.944444	82.055556	20.055556	57.555556	33.300000	0.683
10	**0**	117.571429	72.857143	10.571429	25.071429	30.114286	0.411
	1	125.600000	66.500000	22.900000	48.400000	31.380000	0.514
11	**0**	113.250000	81.000000	10.000000	0.000000	37.125000	0.259
	1	134.000000	70.285714	28.428571	102.857143	39.385714	0.673
12	**0**	111.000000	80.200000	24.600000	31.800000	30.560000	0.301
	1	116.750000	71.500000	30.250000	213.500000	34.575000	0.623
13	**0**	117.200000	74.400000	22.000000	50.000000	33.280000	0.405
	1	133.800000	73.200000	12.600000	5.800000	36.720000	0.521
14	**1**	137.500000	70.000000	27.500000	92.000000	35.100000	0.312
15	**1**	136.000000	70.000000	32.000000	110.000000	37.100000	0.153
17	**1**	163.000000	72.000000	41.000000	114.000000	40.900000	0.817

Aggregating data by two columns in pandas

Any summary method can be used alongside `.groupby()`, including `.min()`, `.max()`, `.mean()`, `.median()`, `.sum()`, `.mode()`, and more.

Pivot tables

pandas also enables you to calculate summary statistics as pivot tables. This makes it easy to draw conclusions based on a combination of variables. The below code picks the rows as unique values of `Pregnancies`, the column values are the unique values of `Outcome`, and the cells contain the average value of `BMI` in the corresponding group.

For example, for `Pregnancies = 5` and `Outcome = 0`, the average BMI turns out to be 31.1.

```
pd.pivot_table(df, values="BMI", index='Pregnancies',

          columns=['Outcome'], aggfunc=np.mean)
```

Outcome	0	1
Pregnancies		
0	31.727397	39.213158
1	29.616038	37.793103
2	29.679762	34.578947
3	29.231250	32.548148
4	31.255556	33.873913
5	31.100000	36.780952
6	29.591176	31.775000
7	29.975000	34.756000
8	30.693750	32.204545
9	28.840000	33.300000
10	30.114286	31.380000
11	37.125000	39.385714
12	30.560000	34.575000
13	33.280000	36.720000
14	NaN	35.100000
15	NaN	37.100000
17	NaN	40.900000

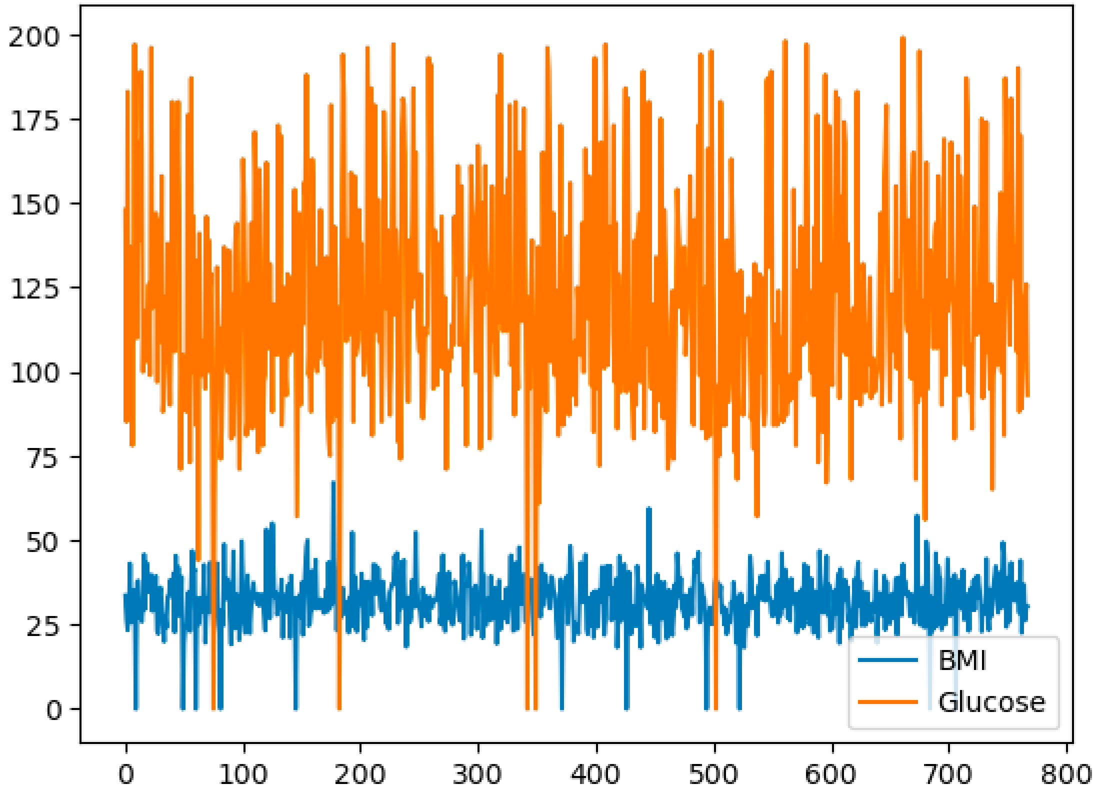
Data visualization in pandas

pandas provides convenience wrappers to `Matplotlib` plotting functions to make it easy to visualize your DataFrames. Below, you'll see how to do common data visualizations using pandas.

Line plots in pandas

pandas enables you to chart out the relationships among variables using line plots. Below is a line plot of BMI and Glucose versus the row index.

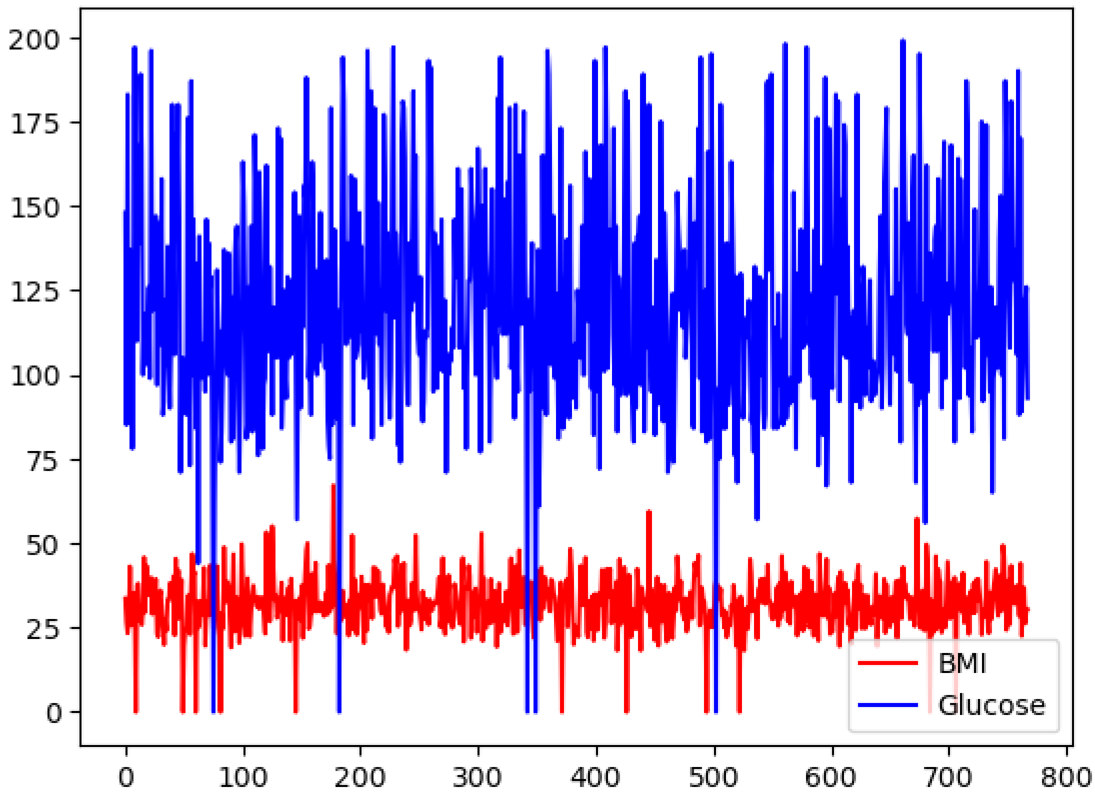
```
df[['BMI', 'Glucose']].plot.line()
```



Basic line plot with pandas

You can select the choice of colors by using the color argument.

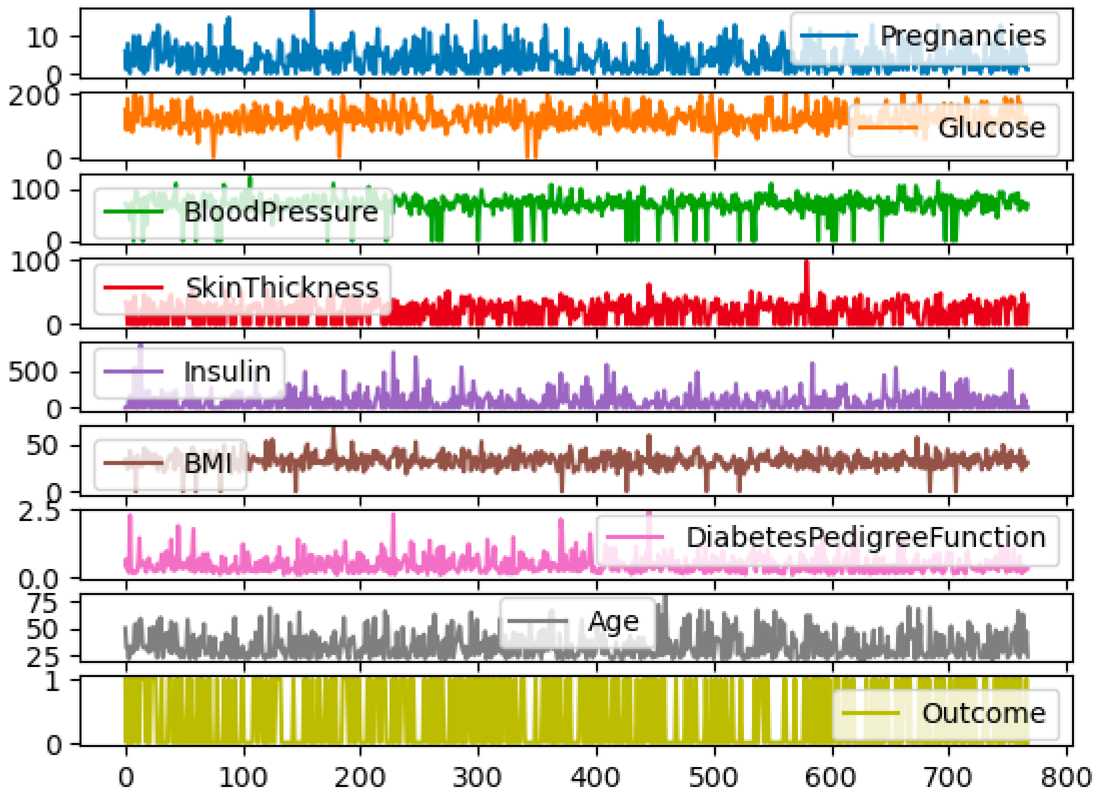
```
df[['BMI', 'Glucose']].plot.line(figsize=(20, 10),

                   color={"BMI": "red", "Glucose": "blue"})
```



Basic line plot with pandas, with custom colors

All the columns of `df` can also be plotted on different scales and axes by using the `subplots` argument.

```
df.plot.line(subplots=True)
```

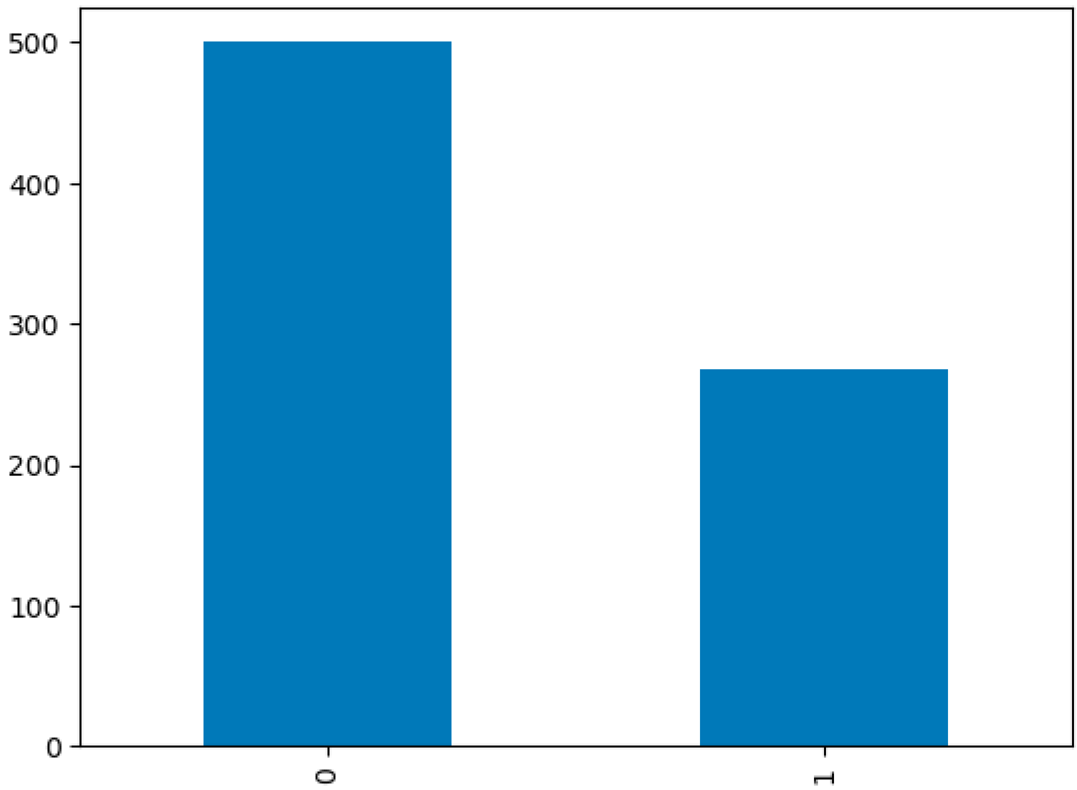


Subplots for line plots with pandas

Bar plots in pandas

For discrete columns, you can use a bar plot over the category counts to visualize their distribution. The variable `Outcome` with binary values is visualized below.

```
df['Outcome'].value_counts().plot.bar()
```

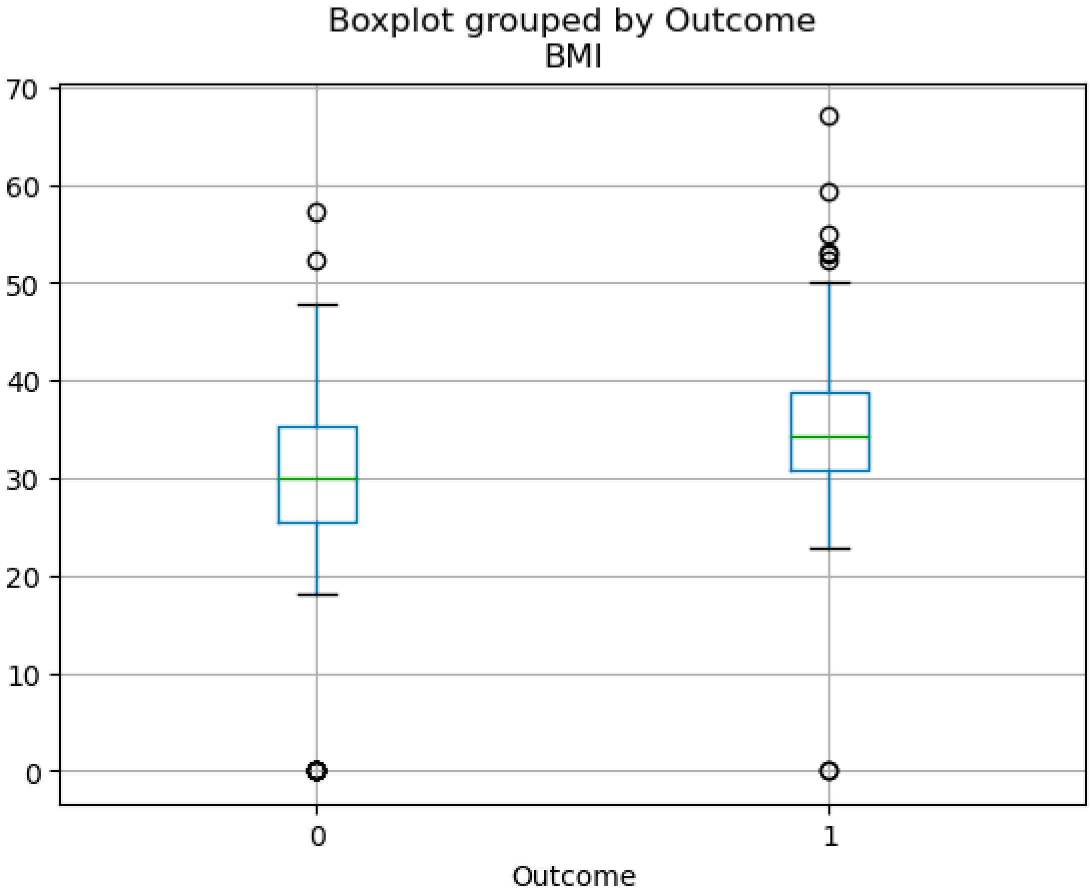


Barplots in pandas

Box plots in pandas

The quartile distribution of continuous variables can be visualized using a boxplot. The code below lets you create a boxplot with pandas.

```
df.boxplot(column=['BMI'], by='Outcome')
```



Boxplots in pandas