Intro to Visualization in Python - Static Plots - 1

One should look for what is and not what he thinks should be. (Albert Einstein)

Static Plots: Topic introduction

In this part of the course, we will cover the following concepts:

- Data visualization basics and use cases
- Build plots with matplotlib

Chat question

- Think back to a specific time you encountered or used a chart or graph to make sense of information
- What specific type of chart or graph did you observe or create?
- How did it help you understand the story the data was telling?

Module completion checklist

Objective	Complete
Prepare data for visualization	
Create histograms, boxplots, and bar charts	

Visualizing data with matplotlib

- `matplotlib` is a popular plotting library among scientists and data analysts
- It is one of the older Python plotting libraries, so it has become quite flexible and ***well-documented (link)***
- Other plotting libraries include `Seaborn` (which is built on `matplotlib`), `ggplot` (the Python version of the popular R plotting library), `Plotly`, `Bokeh`, and many others
- `Pandas` also come with some plotting capabilities, and these are just based on `matplotlib`
- Explore the different types of plots that can be created with `matplotlib` by browsing the ***matplotlib gallery (link)***

Loading packages

- Load the packages:

```
import pandas as pd
import numpy as np
import pickle
import os
from pathlib import Path
```

Directory settings

- In order to maximize the efficiency of the workflow, encode the directory structure into `variables`
- Use the `pathlib` library
- Let the `main_dir` be the variable corresponding to the `course` folder
- Let `data_dir` be the variable corresponding to the `data` folder

```
# Set 'main_dir' to location of the project folder
home_dir = Path(".").resolve()
main_dir = home_dir.parent.parent
print(main_dir)
```

```
data_dir = str(main_dir) + "/data"
print(data_dir)
```

```
plot_dir = str(main_dir) + "/plots"
if not os.path.exists(plot_dir):
    os.makedirs(plot_dir)
print(plot_dir)
```

Importing matplotlib

- Import `pyplot` as `plt` so that `plt.[any_function]()` with appropriate arguments to create a plot
- The `pyplot` module of the `matplotlib` library has a large and diverse set of functions
- It allows the ability to create pretty much any conceivable visualization out there
- See the documentation on `pyplot` ***here(link)***

```
import matplotlib.pyplot as plt
```

matplotlib.pyplot

`matplotlib.pyplot` is a state-based interface to matplotlib. It provides a MATLAB-like way of plotting.

pyplot is mainly intended for interactive plots and simple cases of programmatic plot generation:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 5, 0.1)
y = np.sin(x)
plt.plot(x, y)
```

The object-oriented API is recommended for more complex plots.

Functions

`acorr`(x, *[, data])	Plot the autocorrelation of *x*.
`angle_spectrum`(x[, Fs, Fc, window, pad_to, ...])	Plot the angle spectrum.
`annotate`(s, xy, *args, **kwargs)	Annotate the point *xy* with text *s*.
`arrow`(x, y, dx, dy, **kwargs)	Add an arrow to the axes.
`autoscale`([enable, axis, tight])	Autoscale the axis view to the data (toggle).

Dataset for visualization

- Load the dataset and save it as `df`

```
# This dataset is of type dataframe. Let's assign this dataset to a variable, so that we can manipulate it freely.
df = pd.read_csv(str(data_dir)+"/"+ "diabetes.csv")
```

```
print(type(df))  #<- a Pandas DataFrame!
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
print(len(df))   #<- returns the number of rows
```

```
768
```

Subsetting data

- Create a subset of the data so that the variables needed are present
- Name this subset as `df_subset`

```
df_subset = df[['BMI', 'SkinThickness', 'Glucose', 'BloodPressure', 'DiabetesPedigreeFunction',
'Insulin', 'Age', 'Outcome', 'Pregnancies']]
print(df_subset.head())
```

```
    BMI  SkinThickness  Glucose  ...  Age  Outcome  Pregnancies
0  33.6             35      148  ...   50        1            6
1  26.6             29       85  ...   31        0            1
2  23.3              0      183  ...   32        1            8
3  28.1             23       89  ...   21        0            1
4  43.1             35      137  ...   33        1            0

[5 rows x 9 columns]
```

- These variables are choosen because they illustrate the concepts best
- However, you should be able to work with (and visualize) all of your data

Data reshaping: wide vs. long

- **Data reshaping** refers to converting between either **wide** or **long** data formats
  - **Wide** data is much more visually digestible, which is why it is more common when using data from some type of report
  - **Long** data is much easier to work with in Pandas, and generally speaking in most data analysis and plotting tools

Data reshaping: wide vs. long (cont'd)

- **Wide data** often appears when the values are some type of aggregate (we will use the `mean` of groups)

- Let's make a typical **wide dataframe** of two rows and eight columns that looks like this:

```
   Outcome        BMI  ...     Insulin        Age
0        0  30.304200  ...   68.792000  31.190000
1        1  35.142537  ...  100.335821  37.067164

[2 rows x 8 columns]
```

Prepare data: group and summarize

- After grouping and summarizing data, create a summary dataset that will include the following:
  - Grouped data by `Target` variable
  - Mean value computed on the grouped data that includes the following variables:
    - DiabetesPedigreeFunction
    - Age
    - Glucose
    - Insulin
    - SkinThickness
    - BloodPressure
    - BMI

Prepare data: group and summarize (cont'd)

- For demonstration, use the original dataframe `df` to identify the grouping column
- Then use this column to perform the groupby operation and find the mean of the columns present in `df_subset`

```
col_dict = df_subset.nunique().to_dict()
grouping_col = min(col_dict, key=col_dict.get)
# Group data by variable with min levels.
grouped = df_subset.groupby(grouping_col)
```

```
# Compute mean on the listed variables using the grouped data.
df_grouped_mean = grouped.mean()[['BMI', 'SkinThickness', 'Glucose', 'BloodPressure',
'DiabetesPedigreeFunction', 'Insulin', 'Age']]
print(df_grouped_mean)
```

```
                BMI  SkinThickness  ...     Insulin        Age
Outcome                             ...
0         30.304200      19.664000  ...   68.792000  31.190000
1         35.142537      22.164179  ...  100.335821  37.067164

[2 rows x 7 columns]
```

Prepare data: group and summarize (cont'd)

```
# Reset index of the dataset.
df_grouped_mean = df_grouped_mean.reset_index()
print(df_grouped_mean)
```

```
   Outcome         BMI  ...     Insulin        Age
0        0   30.304200  ...   68.792000  31.190000
1        1   35.142537  ...  100.335821  37.067164

[2 rows x 8 columns]
```

- This dataframe is considered **wide** because each variable has its own column
- It makes the table easier to present, but inconvenient to run analyses on or visualize

Why long?

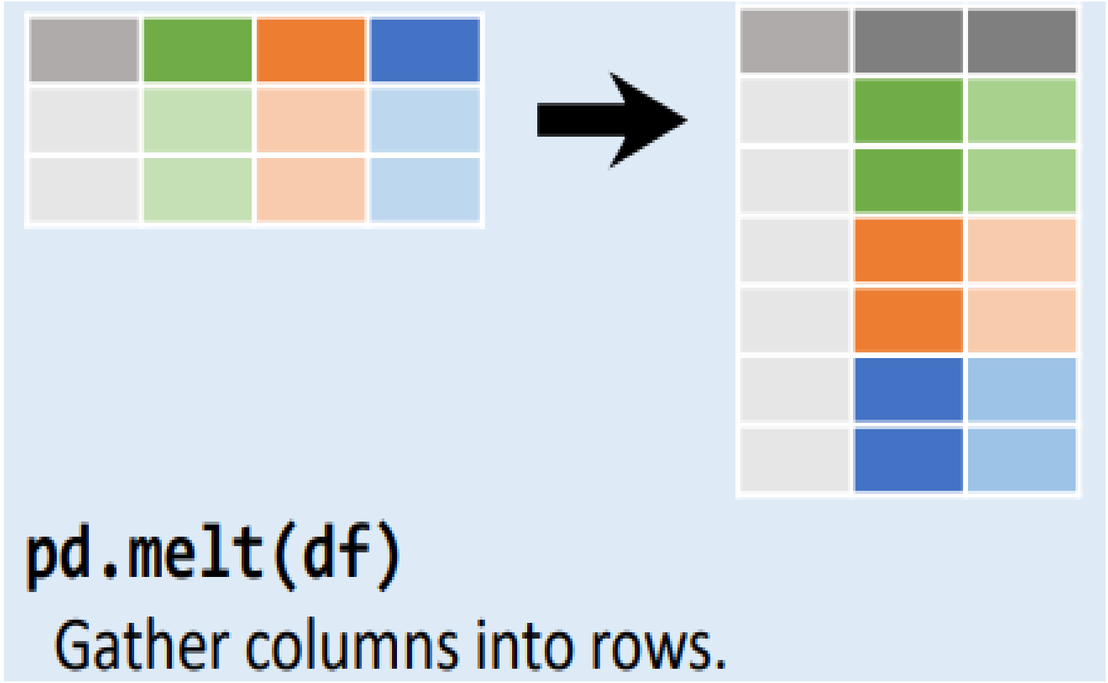
- Now convert this wide data to the **long format**
  - Leave the `categorical` variable and the mean values as is in their columns
  - All of other variables will appear as a single metric column

- This format is convenient to work with when running analysis and plot the data

	stroke	metric	mean
0	0	avg_glucose_level	104.795513
1	1	avg_glucose_level	132.544739
2	0	bmi	28.825388
3	1	bmi	30.217789

Wide to long format: melt

- To **convert from wide to long format**, use the Pandas `melt` function with the following arguments:
  i. Wide dataframe
  ii. Variable(s) that will be preserved as the `ids` of the data (like categorical variables)
  iii. Name of the variable that will now contain the column names from the wide data selected to melt together
  iv. Name of the column that will contain respective values corresponding to the melted columns



Wide to long format: melt (cont'd)

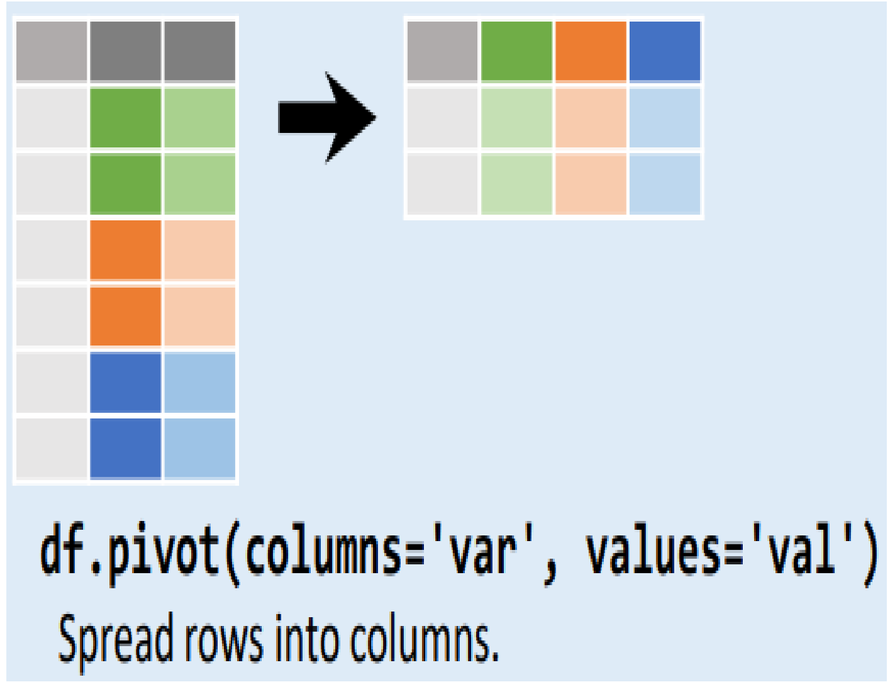
```
# Melt the wide data into long.
df_grouped_mean_long = pd.melt(df_grouped_mean,       #<- wide dataset
                               id_vars = [grouping_col],     #<- identifying variable
                               var_name = 'metric',      #<- contains col names of wide data
                               value_name = 'mean')      #<- contains values from above columns
print(df_grouped_mean_long)
```

```
    Outcome                    metric        mean
0         0                       BMI   30.304200
1         1                       BMI   35.142537
2         0             SkinThickness   19.664000
3         1             SkinThickness   22.164179
4         0                   Glucose  109.980000
5         1                   Glucose  141.257463
6         0             BloodPressure   68.184000
7         1             BloodPressure   70.824627
8         0  DiabetesPedigreeFunction    0.429734
9         1  DiabetesPedigreeFunction    0.550500
10        0                   Insulin   68.792000
11        1                   Insulin  100.335821
12        0                       Age   31.190000
13        1                       Age   37.067164
```

Long to wide format: pivot

- Convert the **long data back to wide** format with the `.pivot()` method

1. The `index` argument refers to what values will become the `ids` in the new dataframe
2. The `columns` argument refers to the column in which its values will be converted to column names
3. Lastly, supply the `values` argument to fill in the values of the wide data



Long to wide format: pivot (cont'd)

```
# Melt the long data into wide.
df_grouped_mean_wide = df_grouped_mean_long.pivot(
                                                    index = [grouping_col],   #<- identifying
variable

                                                    columns = 'metric', #<- col names of wide data
                                                    values = 'mean')    #<- values from above

columns
print(df_grouped_mean_wide)
```

```
metric          Age        BMI  ...     Insulin  SkinThickness
Outcome                         ...
0         31.190000  30.304200  ...   68.792000      19.664000
1         37.067164  35.142537  ...  100.335821      22.164179

[2 rows x 7 columns]
```

Module completion checklist

Objective	Complete
Prepare data for visualization	✔
Create histograms, boxplots, and bar charts	

Univariate plots

- **Univariate plots** are used to **visualize the distribution of a single variable**

- They are mainly used in the initial stages of EDA when we want to learn more about individual variables in our data

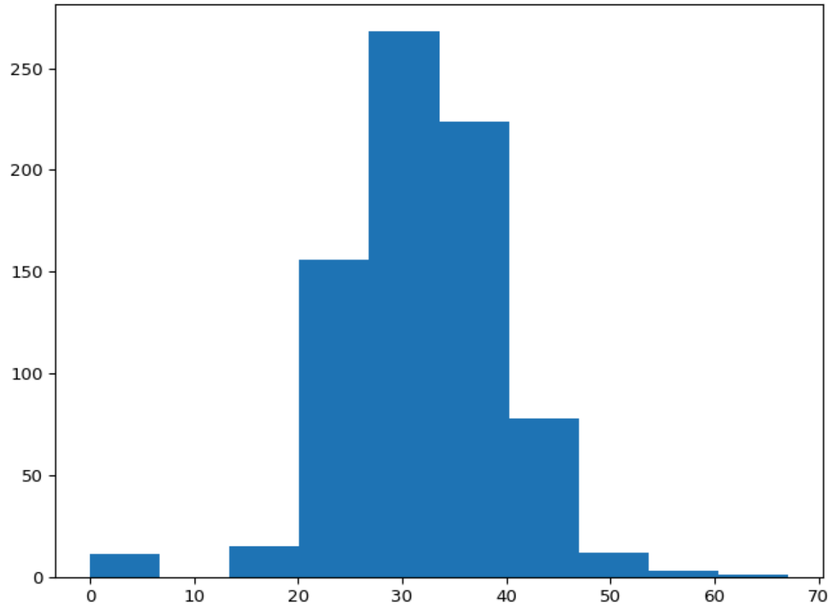
- They are also combined with other univariate plots to compare data distributions of different variables

- Univariate plots include the following popular graphs: `histogram`, `boxplot`, `density curve`, `dot plot`, `QQ plot`, and `bar plot`

Univariate plots: histogram

- A **histogram** represents the **distribution of numerical data**
- The height of each bar has been calculated as the number of observations in that range
- Use `plt.hist()` to produce a basic histogram of any numeric variable

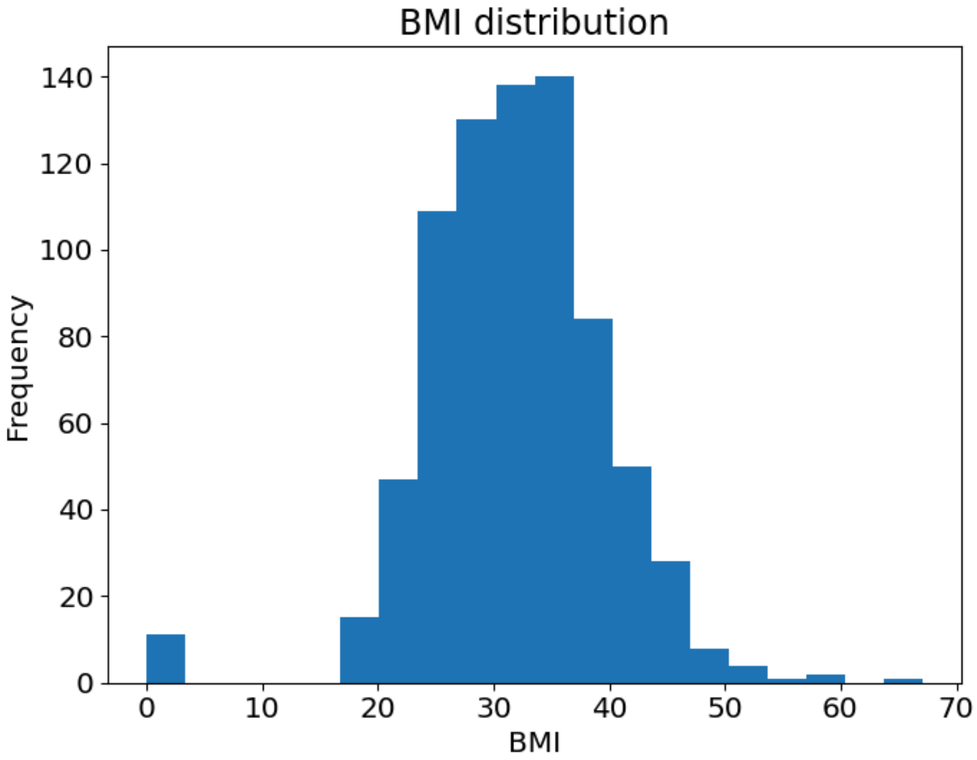
```
plt.rcParams.update({'font.size': 15})
plt.hist(df_subset['BMI'])
plt.show()
```



Univariate plots: histogram (cont'd)

- **Bins** represent the intervals in which to group the observations
- Control the number of bins with the `bins` parameter
- As the **number of bins increases**, the range of values each bin represents decreases, and so does the height of the bar

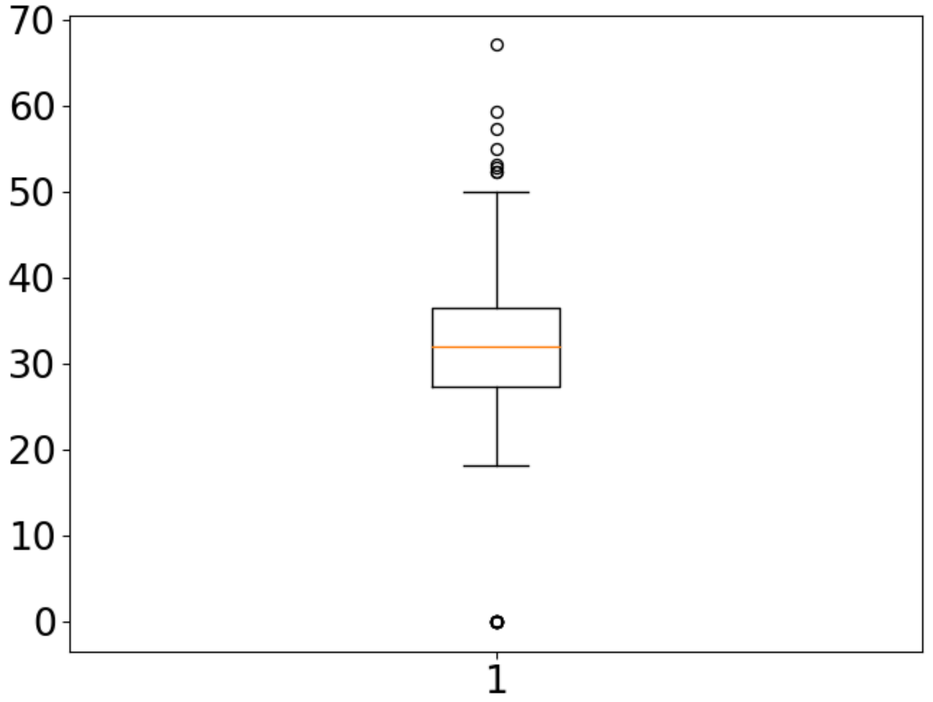
```
plt.hist(df_subset['BMI'], bins = 20)
plt.xlabel('BMI')       #<- label x-axis
plt.ylabel('Frequency')      #<- label y-axis
plt.title('BMI distribution')      #<- add plot
title
plt.show()
```



Univariate plots: boxplot

- A **boxplot** is a visual summary of the **25th, 50th, and 75th percentiles**
- The orange line shows the median of `ppl_total`
- The top and bottom of the box are the `25th` and `75th` percentile respectively
- The outermost lines are called the `whiskers`
- Values beyond whiskers are considered **outliers** - they are substantially outside the rest of the data

```
plt.boxplot(df_subset['BMI'])
plt.show()
```

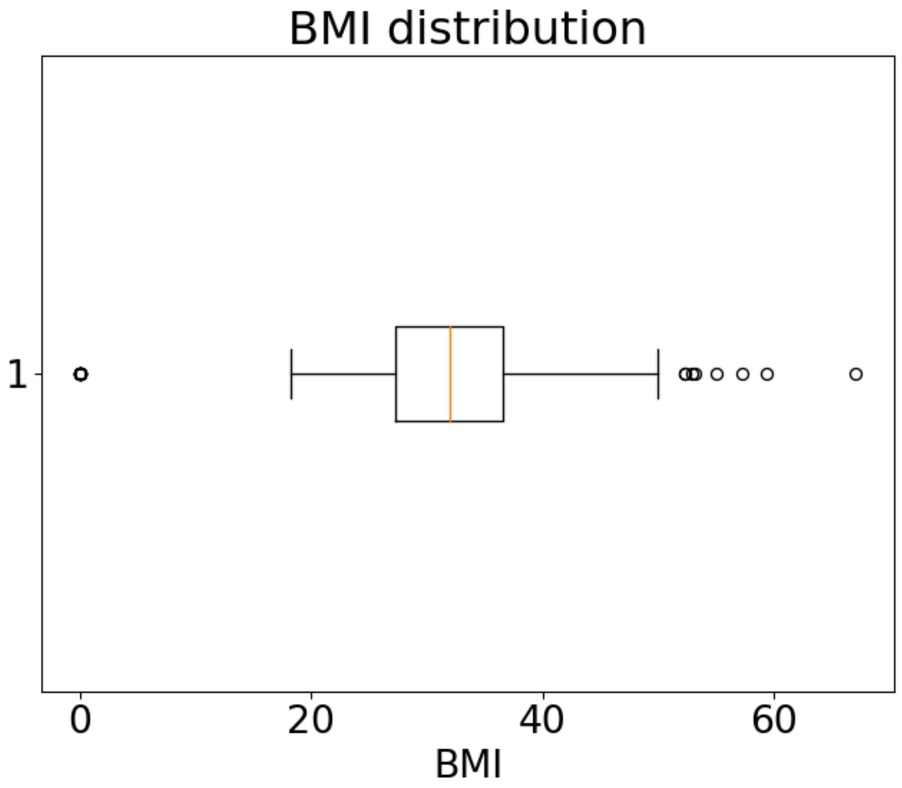


Univariate plots: boxplot (cont'd)

- The orientation of the plot can be set to horizontal by setting `vert = False`

```
plt.boxplot(df_subset['BMI'], vert = False)
plt.xlabel('BMI')        # label x-axis
plt.title('BMI distribution')      # add plot
title
plt.show()
```

- Answer in chat: By looking at this boxplot, what can be told about the **`'BMI'`** distribution in the data?



Univariate plots: bar chart

- A **bar chart** is a plot where the height of each bar represents **the numeric value of a category**

- Use `plt.bar()` to produce a basic histogram of **any categorical variable**

- Bar charts are most commonly used when visualizing survey data or summary data

- The general syntax for creating a bar chart consists of 3 main variables:
  - position of the bars on the `axis`
  - height of the bars
  - names of categories that are used to label the bars

Univariate plots: bar chart (cont'd)

- When plotting bar charts of any complexity, the best type of data to use is **long data**
- First create a simple bar chart of the variable means using the `df_grouped_mean_long` data we created earlier

```
print(df_grouped_mean_long)
```

Univariate plots: bar chart (cont'd)

- Next, filter `'Outcome'` by a category and only keep two columns: `metric` and `mean`

```
query = 'Outcome' + "==" + str('0')
df_false_means = df_grouped_mean_long.query(query)[['metric','mean']]
print(df_false_means)
```

Univariate plots: bar chart (cont'd)

- Now, get the data needed and assign it to the three variables for convenience and clarity:

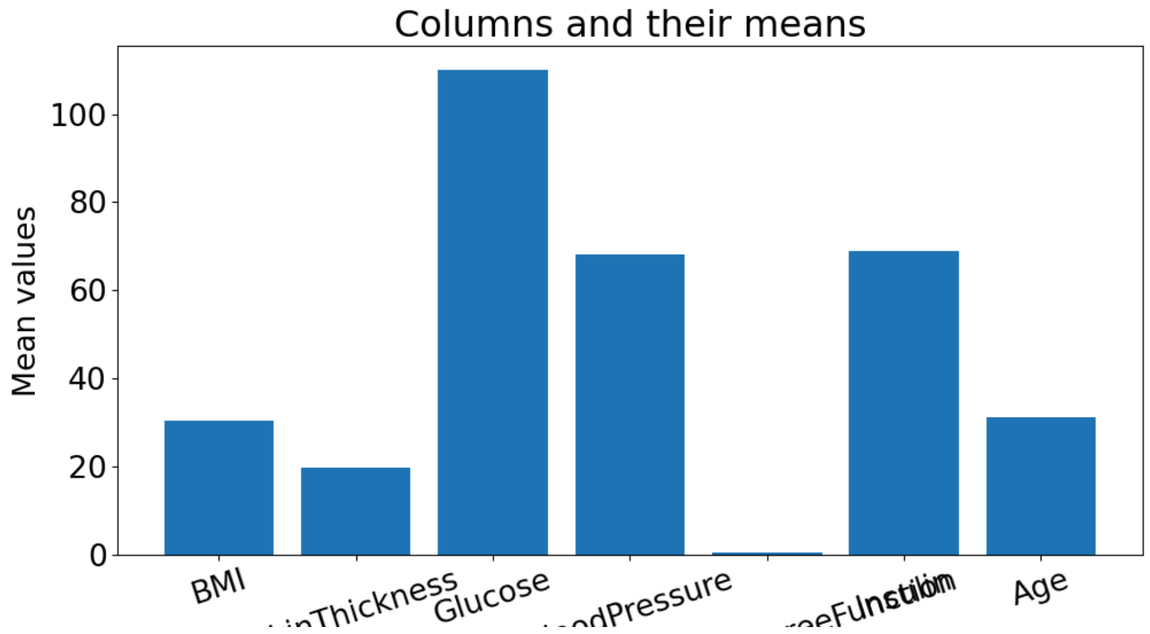
1. **Categories** (i.e., labels) that will represent each bar are all contained in the `metric` column
2. **Bar heights** are contained in the `mean` column for each of the 5 categories
3. **Bar positions** will be a range of numbers based on the number of categories (i.e., bars)

```
bar_labels = df_false_means['metric']      #<- 1
bar_heights = df_false_means['mean']       #<- 2
num_bars = len(bar_heights)
bar_positions = np.arange(num_bars)       #<- 3
```

Univariate plots: bar chart (cont'd)

- Labels are tricky to fit sometimes, so either **adjust** the figure size or label orientation

```
plt.figure(figsize = (12, 6))
plt.bar(bar_positions, bar_heights)
plt.xticks(bar_positions,bar_labels,rotation = 18)
plt.ylabel('Mean values')
plt.title('Columns and their means')  #<- add plot title
```



Customize anything

- All possible style customizations are available in a `matplotlibrc` file
- ***The customizing Matplotlib with style sheets and rcParams (link)*** contains all of them, and any of those parameters can be passed to `rcParams` variable like demonstrated earlier
- The example below contains a script of parameters and their default values
- Here is a part of that file with a sample of all parameters for modifying the style of the `axes`

```
## ***************************************************************************
## * AXES                                                                    *
## ***************************************************************************
## Following are default face and edge colors, default tick sizes,
## default font sizes for tick labels, and so on.  See
## https://matplotlib.org/api/axes_api.html#module-matplotlib.axes
#axes.facecolor:     white   # axes background color
#axes.edgecolor:     black   # axes edge color
#axes.linewidth:     0.8     # edge line width
#axes.grid:          False   # display grid or not
#axes.grid.axis:     both    # which axis the grid should apply to
#axes.grid.which:    major   # grid lines at {major, minor, both} ticks
#axes.titlelocation: center  # alignment of the title: {left, right, center}
```

Knowledge check

Module completion checklist

Objective	Complete
Prepare data for visualization	✔
Create histograms, boxplots, and bar charts	✔

Congratulations on completing this module!

You are now ready to try Tasks 1-13 in the Exercise for this topic