DATA SOCIETY:

Intro to Visualization in Python - Static Plots - 1

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

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***
- Other plotting libraries you may come across are `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 you can create with `matplotlib` by browsing their ***gallery***

Loading packages

- Let’s load the packages we will be using:

```
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 your workflow, you should encode your directory structure into `variables`
- We will use the `pathlib` library
- Let the `main_dir` be the variable corresponding to your `course` folder
- Let `data_dir` be the variable corresponding to your `data` folder

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

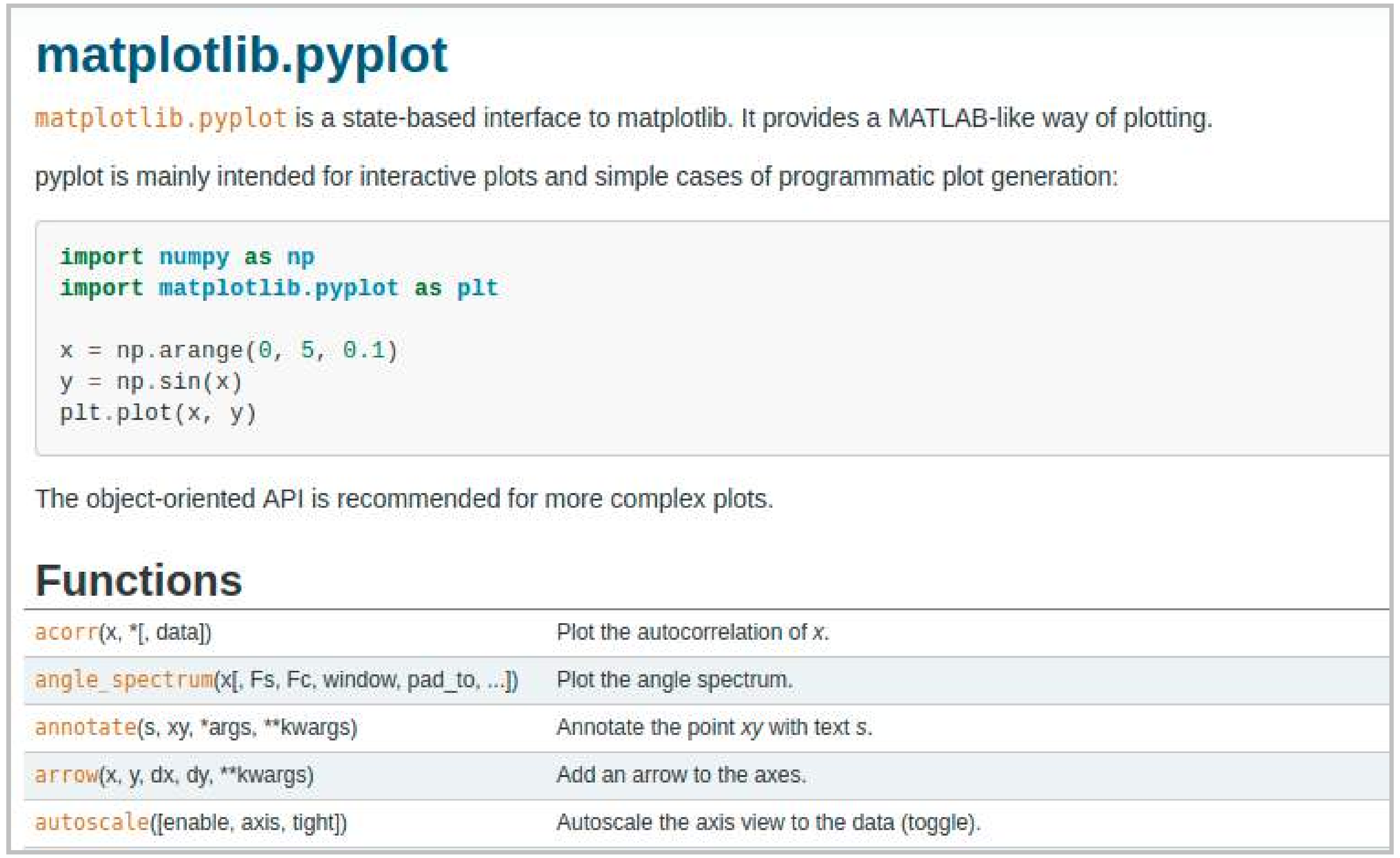
```
data_dir = str(main_dir) + "/data"
print(data_dir)
```

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

Importing matplotlib

- Let’s import `pyplot` as `plt` so that we can call `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 us to create pretty much any conceivable visualization out there
- See the documentation on `pyplot` ***here***

```
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

- We will now 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!
```

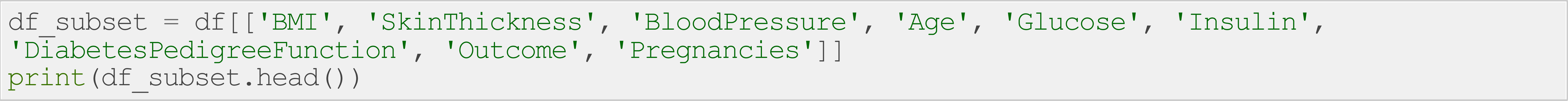
```
<class 'pandas.core.frame.DataFrame'>
```

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

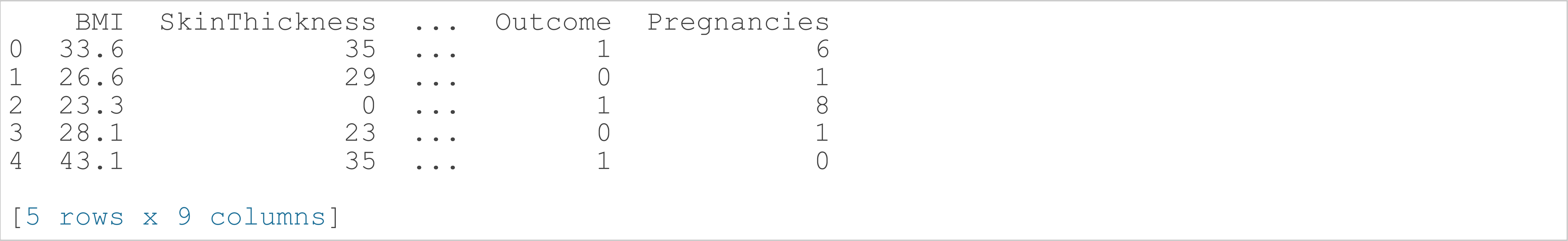
```
768
```

Subsetting data

- Let’s subset our data so that we have the variables we need
- Let’s name this subset `df_subset`



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



```
    BMI  SkinThickness  ...  Outcome  Pregnancies
0  33.6             35  ...        1            6
1  26.6             29  ...        0            1
2  23.3              0  ...        1            8
3  28.1             23  ...        0            1
4  43.1             35  ...        1            0

[5 rows x 9 columns]
```

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

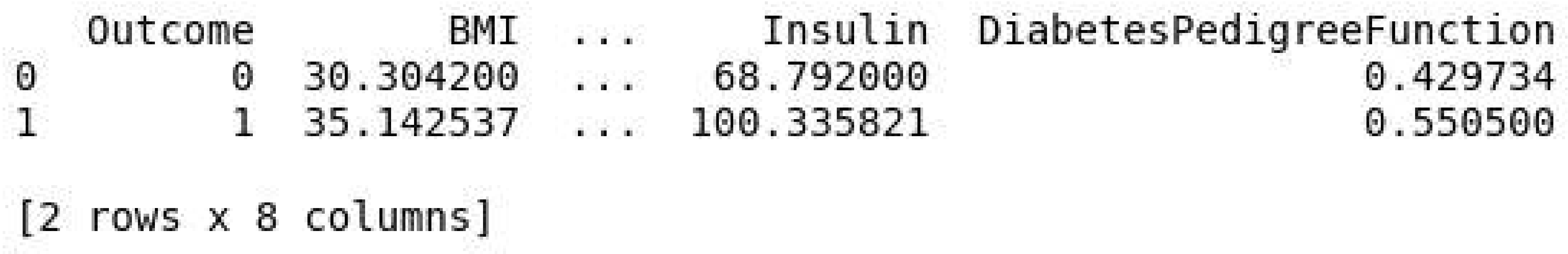
Data Reshaping: wide vs. long

- Talking about data reshaping usually refers to converting between what is called either **wide** or **long** data formats
  - **Wide** data is much more visually digestible, which is why you're likely to come across it if you are 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 six columns that looks like this:



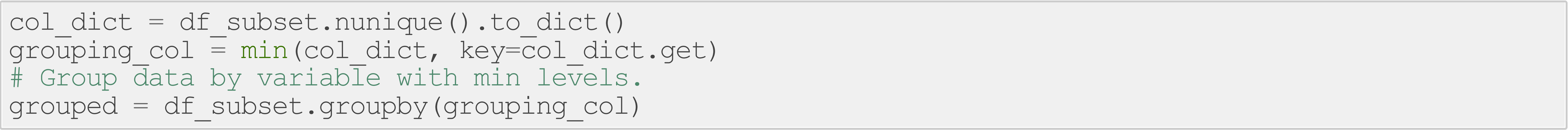
```
   Outcome        BMI  ...     Insulin  DiabetesPedigreeFunction
0        0  30.304200  ...   68.792000                  0.429734
1        1  35.142537  ...  100.335821                  0.550500

[2 rows x 8 columns]
```

Prepare data: group and summarize

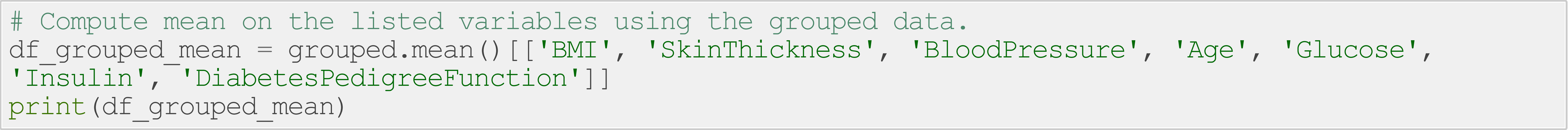
- Now that we know how to group and summarize data, let’s create a summary dataset that would include the following:
  - Grouped data by `Target` variable
  - Mean value computed on the grouped data that includes the following variables:
    - `BMI`
    - `SkinThickness`
    - `BloodPressure`
    - `Age`
    - `Glucose`
    - `Insulin`
    - `DiabetesPedigreeFunction`

Prepare data: group and summarize (cont’d)

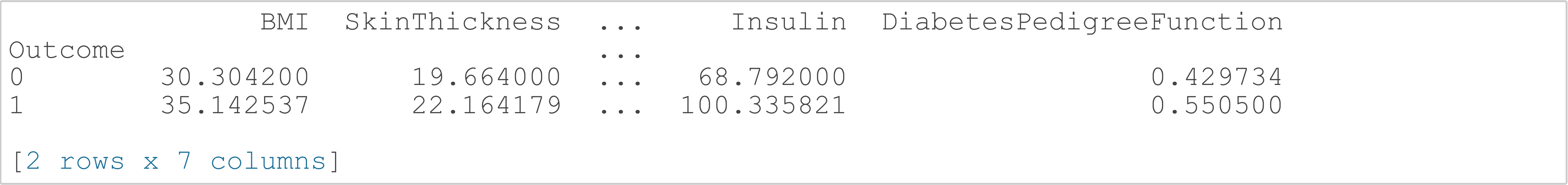
- For demonstration, we use the original dataframe `df` to identify the grouping column
- Then we use this column to perform the groupby operation and find the mean of the columns present in `df_subset`



```
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', 'BloodPressure', 'Age', 'Glucose',
'Insulin', 'DiabetesPedigreeFunction']]
print(df_grouped_mean)
```



```
                BMI  SkinThickness  ...     Insulin  DiabetesPedigreeFunction
Outcome                             ...
0         30.304200      19.664000  ...   68.792000                  0.429734
1         35.142537      22.164179  ...  100.335821                  0.550500

[2 rows x 7 columns]
```

Prepare data: group and summarize (cont'd)

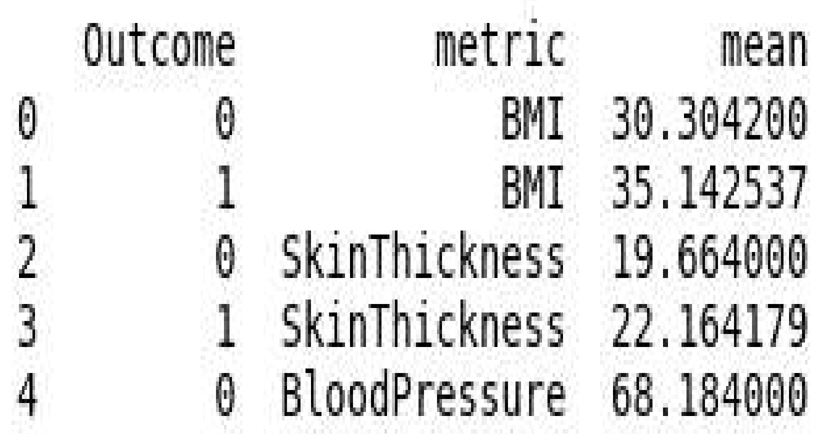
```
# Reset index of the dataset.
df_grouped_mean = df_grouped_mean.reset_index()
print(df_grouped_mean)
```

```
   Outcome        BMI  ...     Insulin  DiabetesPedigreeFunction
0        0  30.304200  ...   68.792000                  0.429734
1        1  35.142537  ...  100.335821                  0.550500

[2 rows x 8 columns]
```

- We call this dataframe **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 let’s convert this wide data to the **long format**
  - We are going to leave the `categorical` variable and the mean values as is in their columns
  - All of our other variables will appear as a single metric column

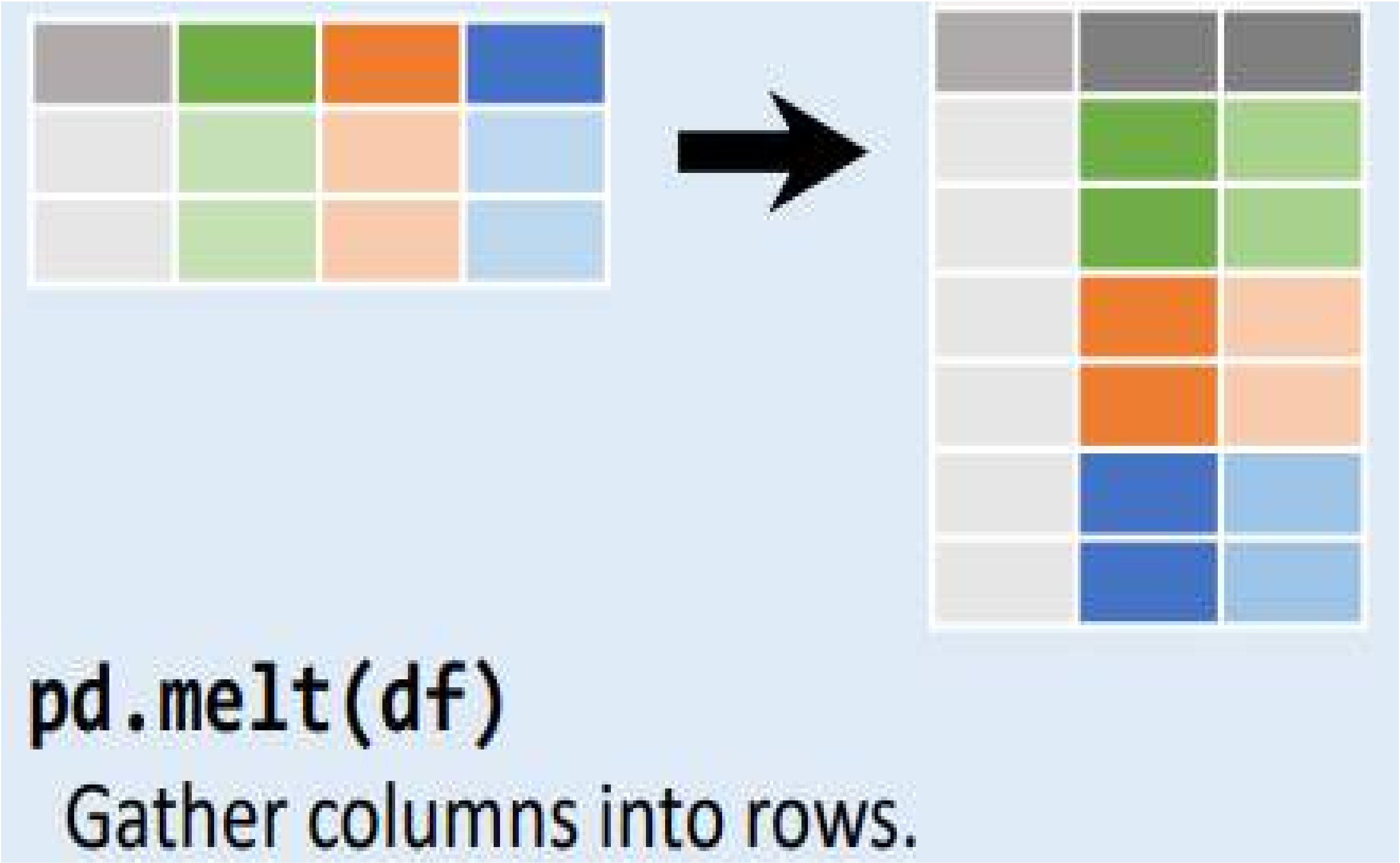


```
   Outcome         metric       mean
0        0            BMI  30.304200
1        1            BMI  35.142537
2        0  SkinThickness  19.664000
3        1  SkinThickness  22.164179
4        0  BloodPressure  68.184000
```

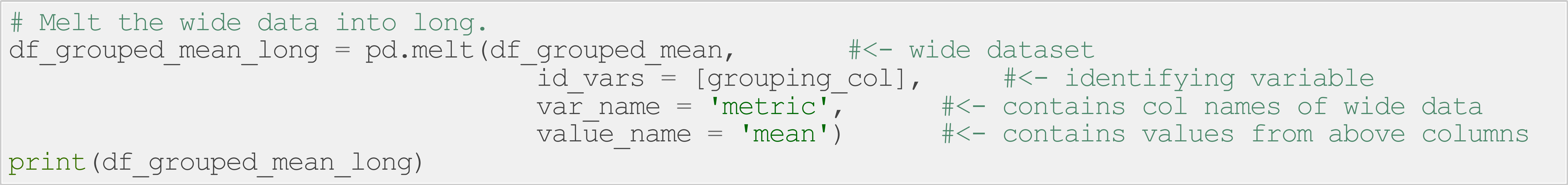
- This format is convenient to work with when we run analysis and plot the data

Wide to long format: melt

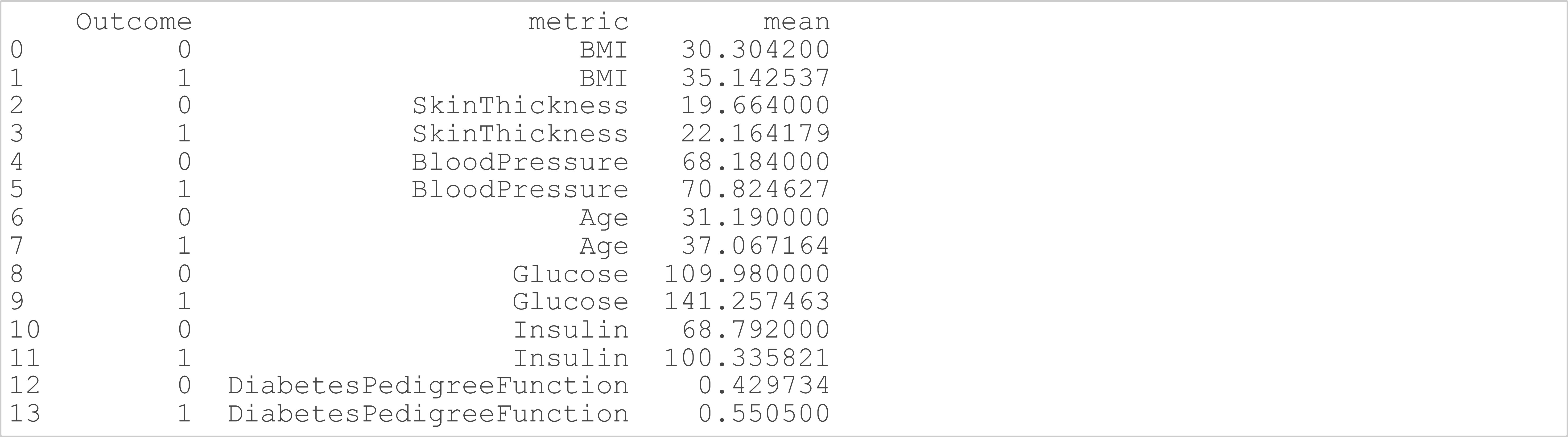
- To **convert from wide to long format**, we use the Pandas `melt` function with the following arguments:
  i. Wide dataframe
  ii. Variable(s) that will be preserved as the `ids` of the data (like categorical variables)
  iii. Name of the variable that will now contain the column names from the wide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

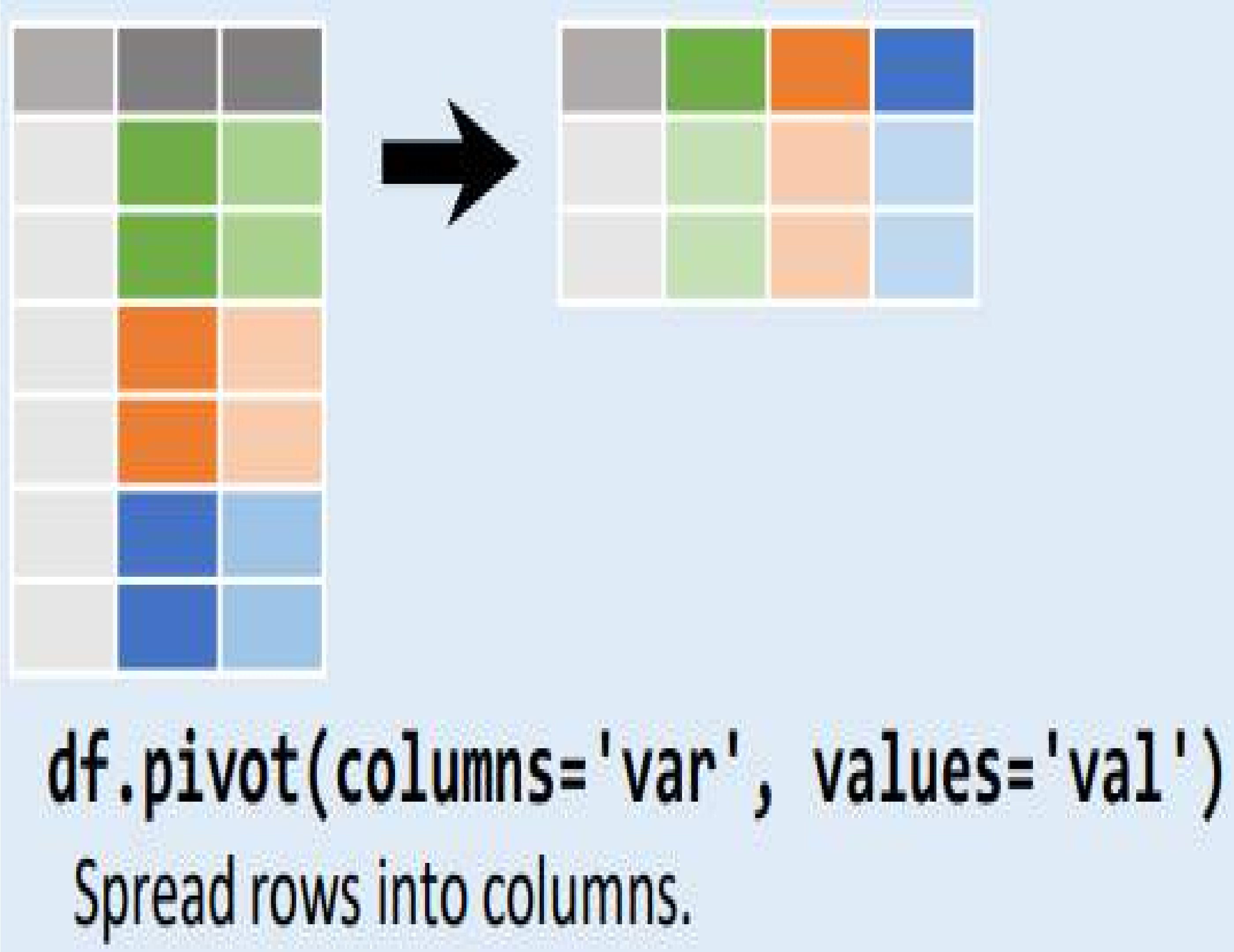


```
# Melt the wide data into long.
df_grouped_mean_long = pd.melt(df_grouped_mean,      #<- wide dataset
                                  id_vars = [grouping_col],     #<- identifying variable
                                  var_name = 'metric',      #<- contains col names of wide data
                                  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             BloodPressure   68.184000
5         1             BloodPressure   70.824627
6         0                       Age   31.190000
7         1                       Age   37.067164
8         0                   Glucose  109.980000
9         1                   Glucose  141.257463
10        0                   Insulin   68.792000
11        1                   Insulin  100.335821
12        0  DiabetesPedigreeFunction    0.429734
13        1  DiabetesPedigreeFunction    0.550500
```

Long to wide format: pivot



```
df.pivot(columns='var', values='val')
```

Spread rows into columns.

We can 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, we 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**

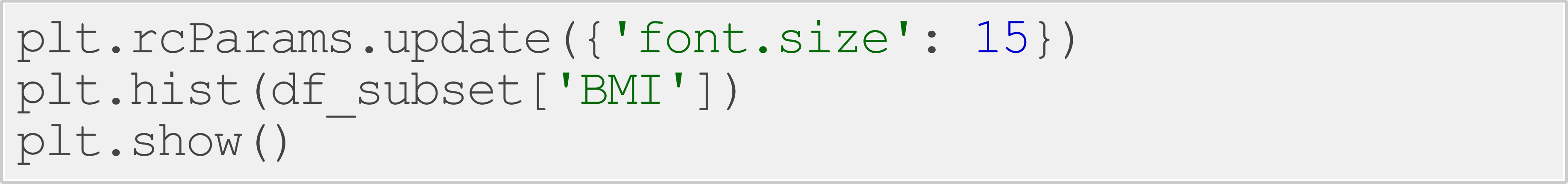
- They are mainly used in the initial stages of EDA when we want to learn more about individual variables in our data

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

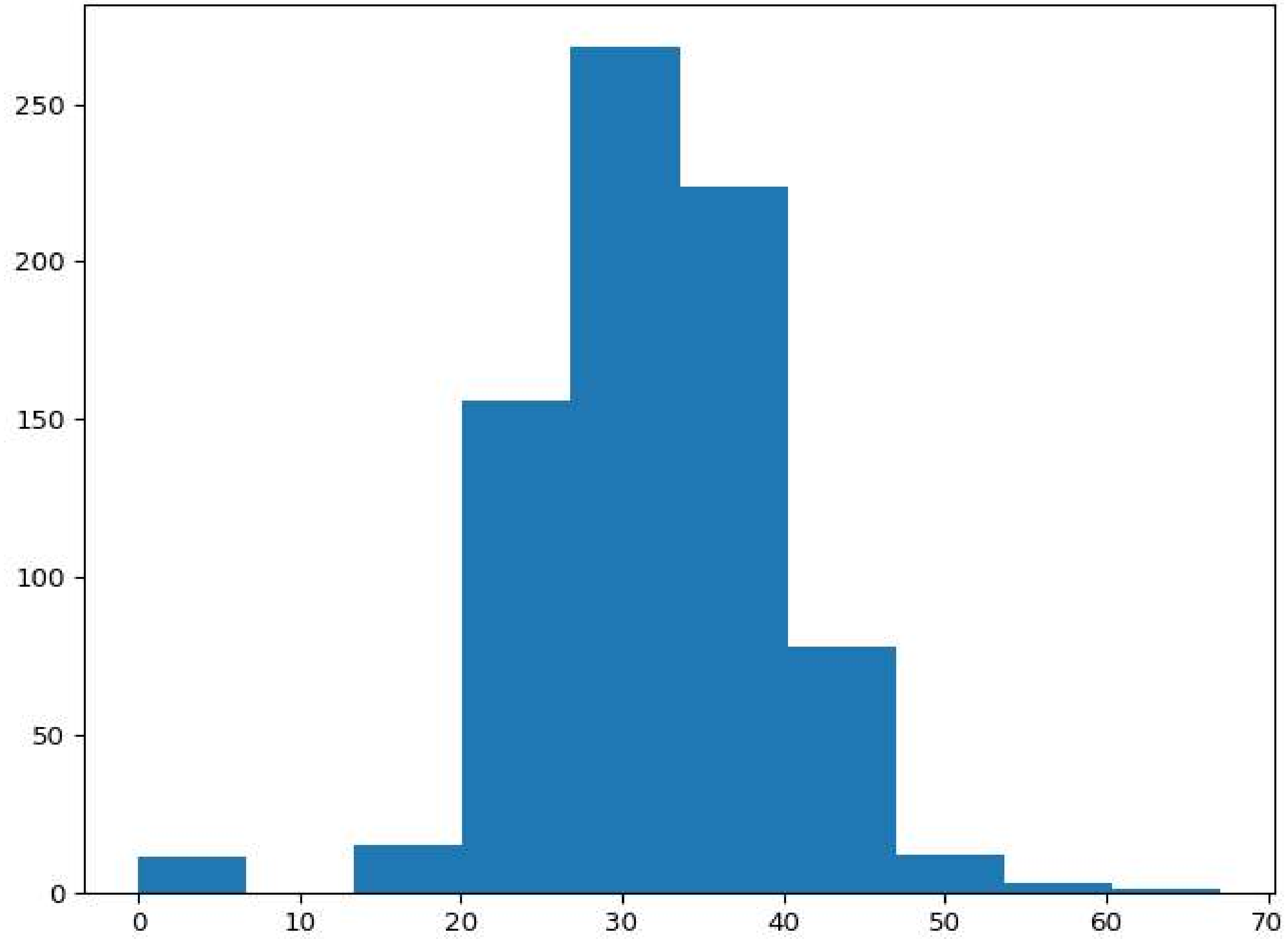
- Univariate plots include the following popular graphs: `histogram`, `boxplot`, `density curve`, `dot plot`, `QQ plot`, and `bar plot`

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
- We can use `plt.hist()` to produce a basic histogram of any numeric variable



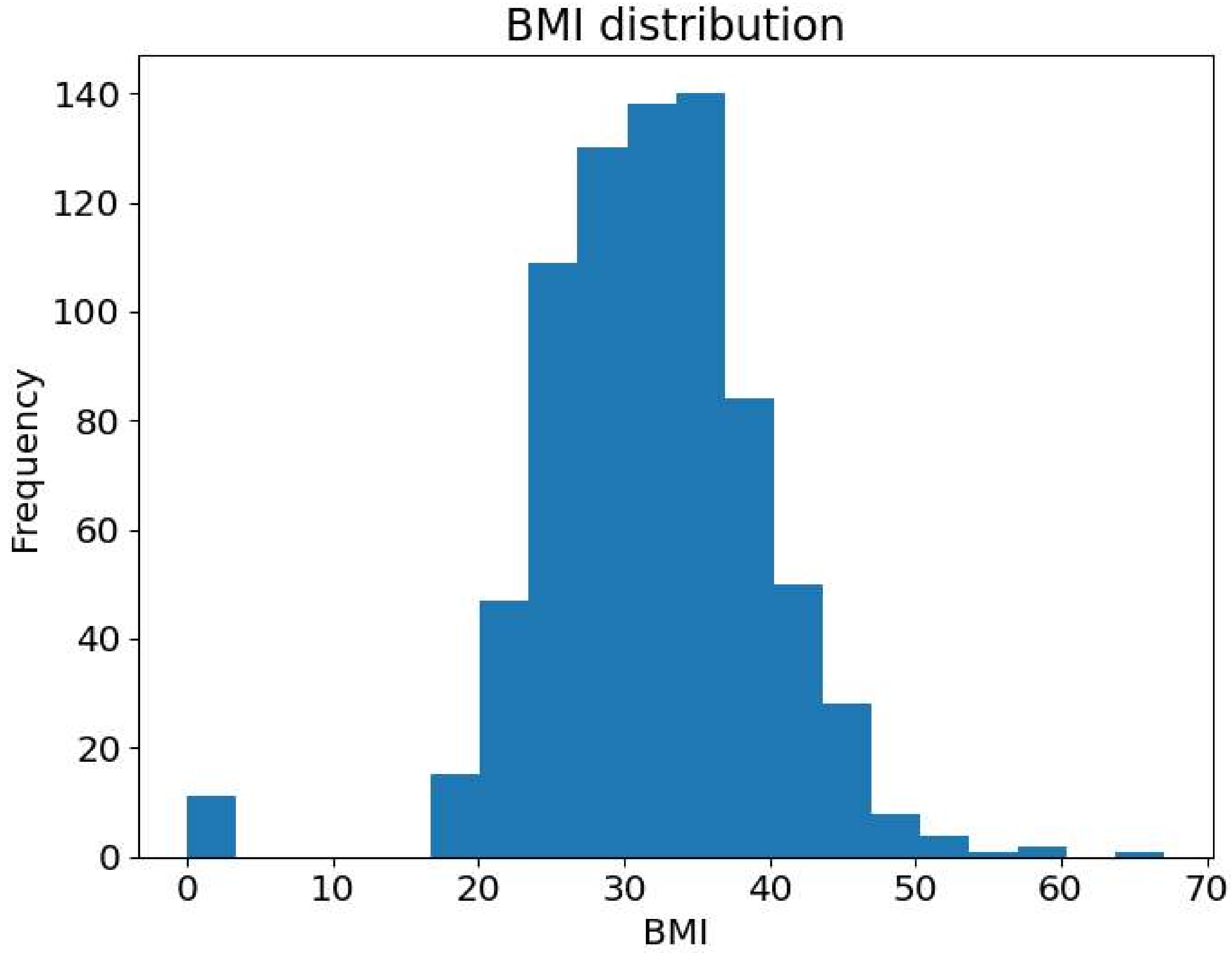
```
plt.rcParams.update({'font.size': 15})
plt.hist(df_subset['BMI'])
plt.show()
```



Univariate plots: histogram (cont'd)

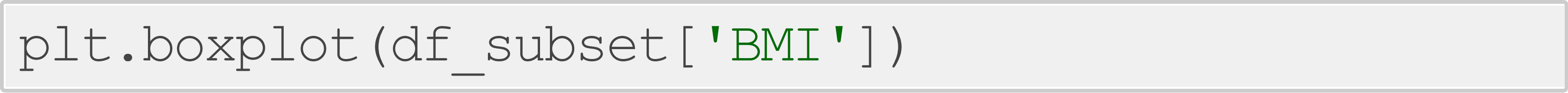
- Bins represent the intervals in which we want to group the observations
- Control the number of bins with the `bins` parameter
- As the **number of bins increases**, the range of values each bin represents decreases, and so does the height of the bar

```
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()
```

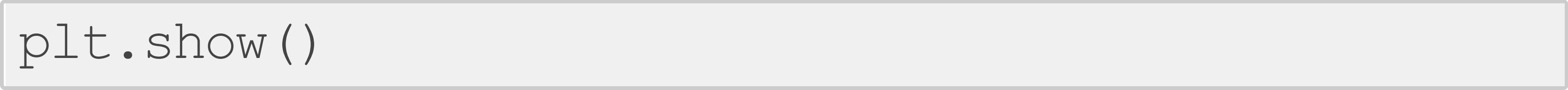


Univariate plots: boxplot

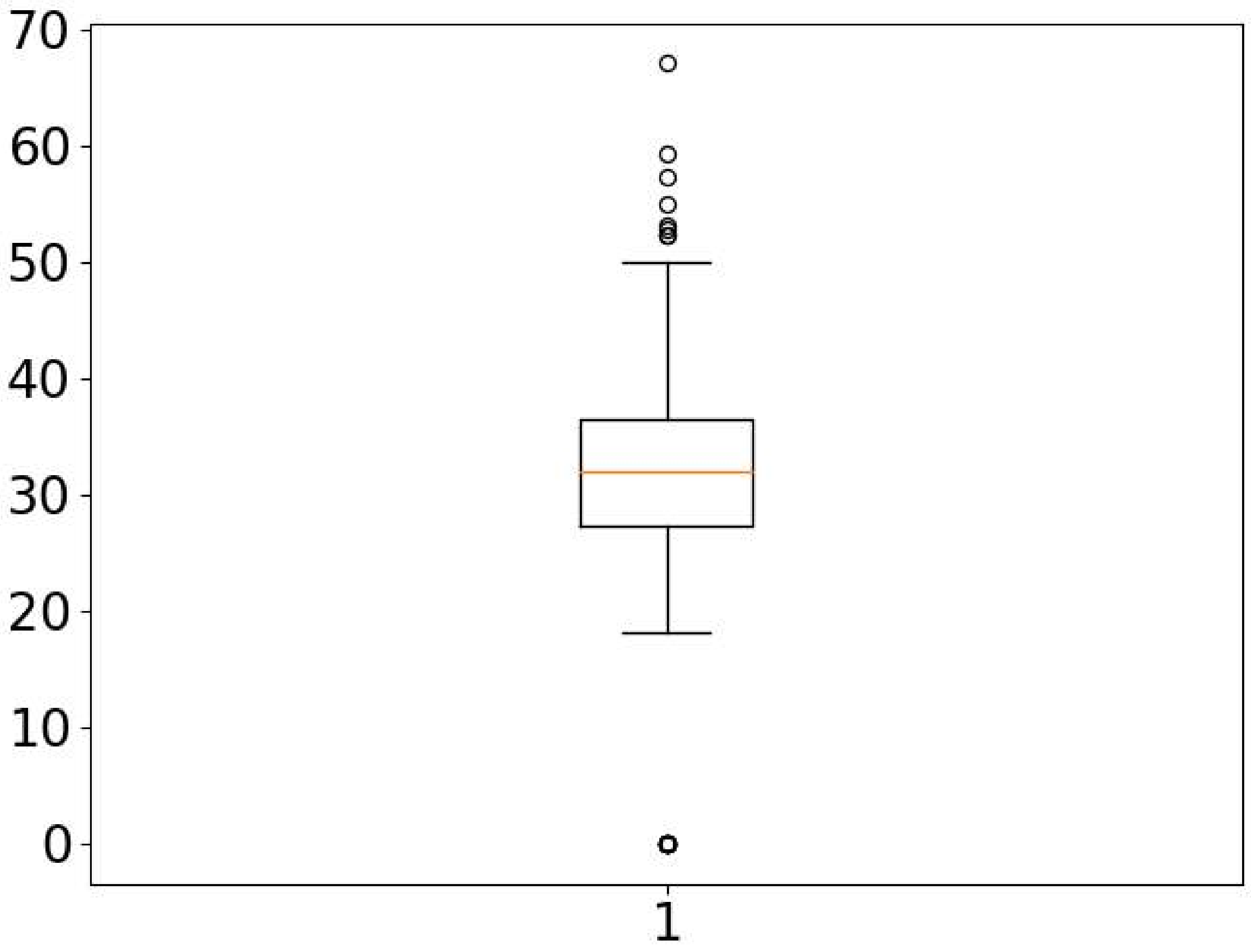
- A boxplot is a visual summary of the **25th, 50th, and 75th percentiles**
- The orange line shows the median of `ppl_total`
- The top and bottom of the box are the `25th` and `75th` percentile respectively
- The outermost lines are called the `whiskers`
- Values beyond whiskers are considered **outliers**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



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



```
plt.show()
```



Univariate plots: boxplot (cont'd)

- You can change the orientation of the plot to horizontal by setting `vert = False`

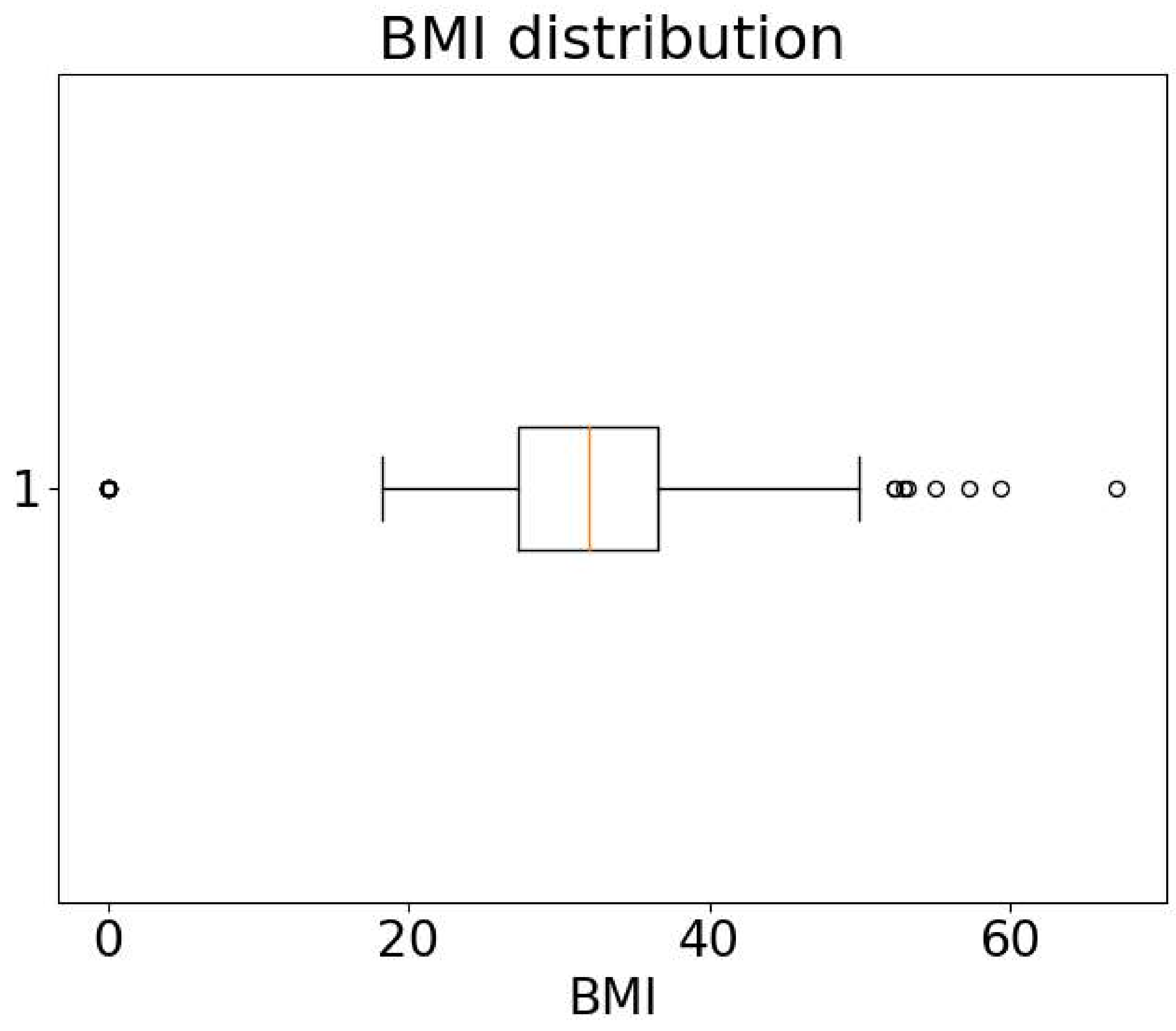
- Answer in chat: By looking at this boxplot, what can you tell about the **'BMI'** distribution in our data?

```
plt.boxplot(df_subset['BMI'], vert = False)
```

```
plt.xlabel('BMI')         # label x-axis
```

```
plt.title('BMI distribution')        # add plot
title
```

```
plt.show()
```



Univariate plots: bar chart

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

- We can 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.head())
```

```
   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  BloodPressure  68.184000
```

Univariate plots: bar chart - cont’d

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

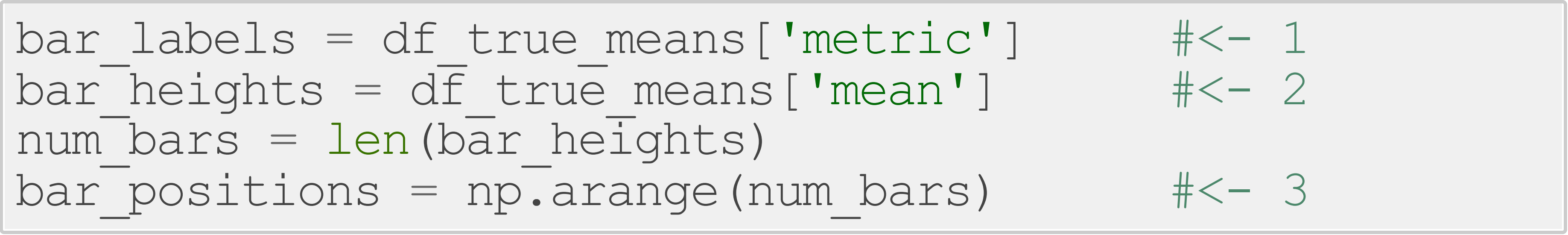
```
query = 'Outcome' + "==" + str('0')
df_true_means = df_grouped_mean_long.query(query)[['metric','mean']]
print(df_true_means)
```

```
                      metric        mean
0                        BMI   30.304200
2              SkinThickness   19.664000
4              BloodPressure   68.184000
6                        Age   31.190000
8                    Glucose  109.980000
10                   Insulin   68.792000
12  DiabetesPedigreeFunction    0.429734
```

Univariate plots: bar chart - cont’d

- Now, get the data we need 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_true_means['metric']     #<- 1
bar_heights = df_true_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 we can either **adjust** the figure size or label orientation

```
# Adjust figure size before plotting.
plt.figure(figsize = (15, 12))
```

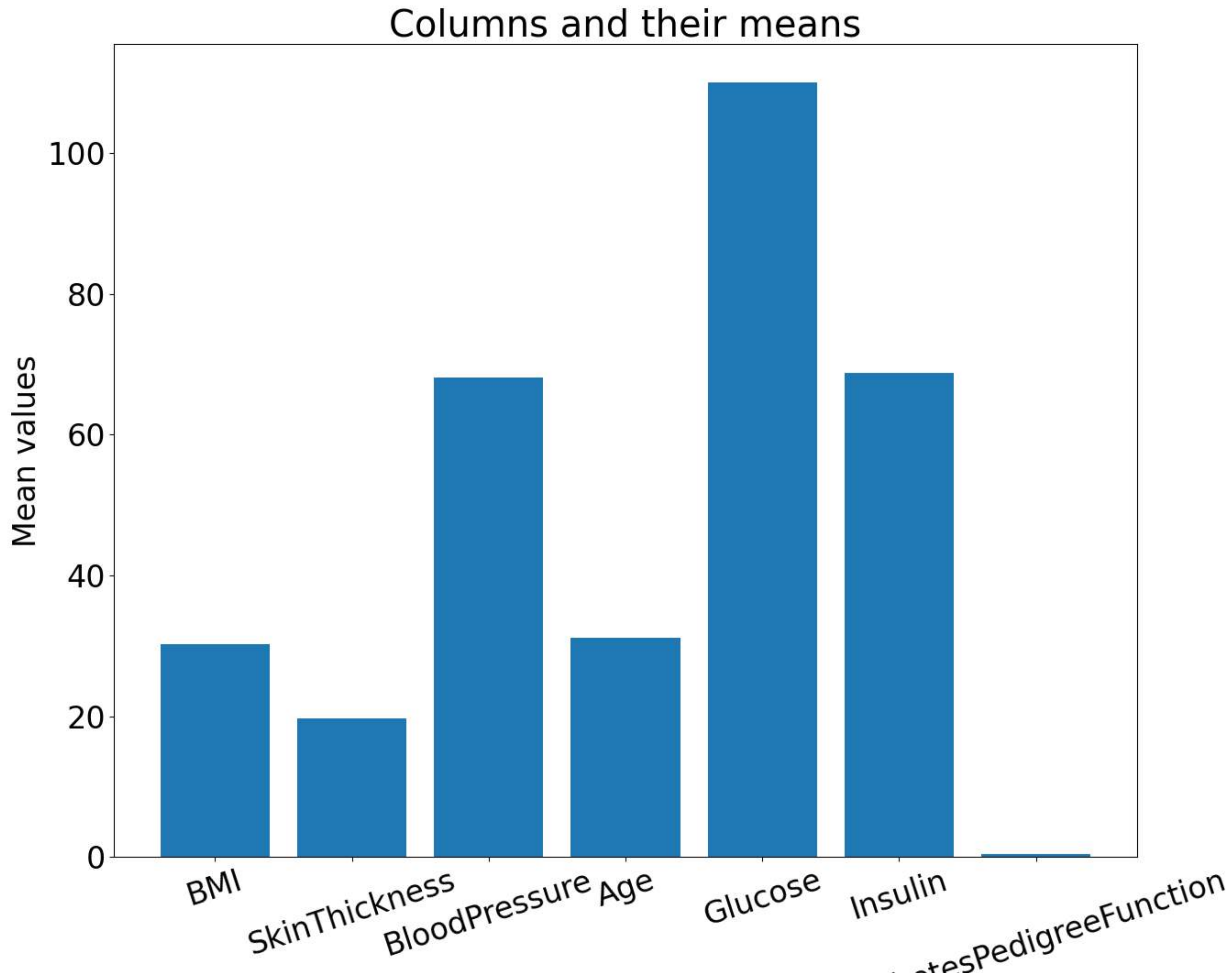
```
plt.bar(bar_positions, bar_heights)
```

```
plt.xticks(bar_positions,
           bar_labels,
           rotation = 18)
```

```
plt.ylabel('Mean values')
```

```
plt.title('Columns and their means')  #
<- add plot title
```

```
plt.show()
```



Customize anything

- All possible style customizations are available in a `matplotlibrc` file
- *This sample* contains all of them, and any of those parameters can be passed to `rcParams` variable like we did earlier
- This sample contains a script of parameters and their default values
- Here's 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

Link: ***Click here to complete the 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