# DATA SOCIETY®

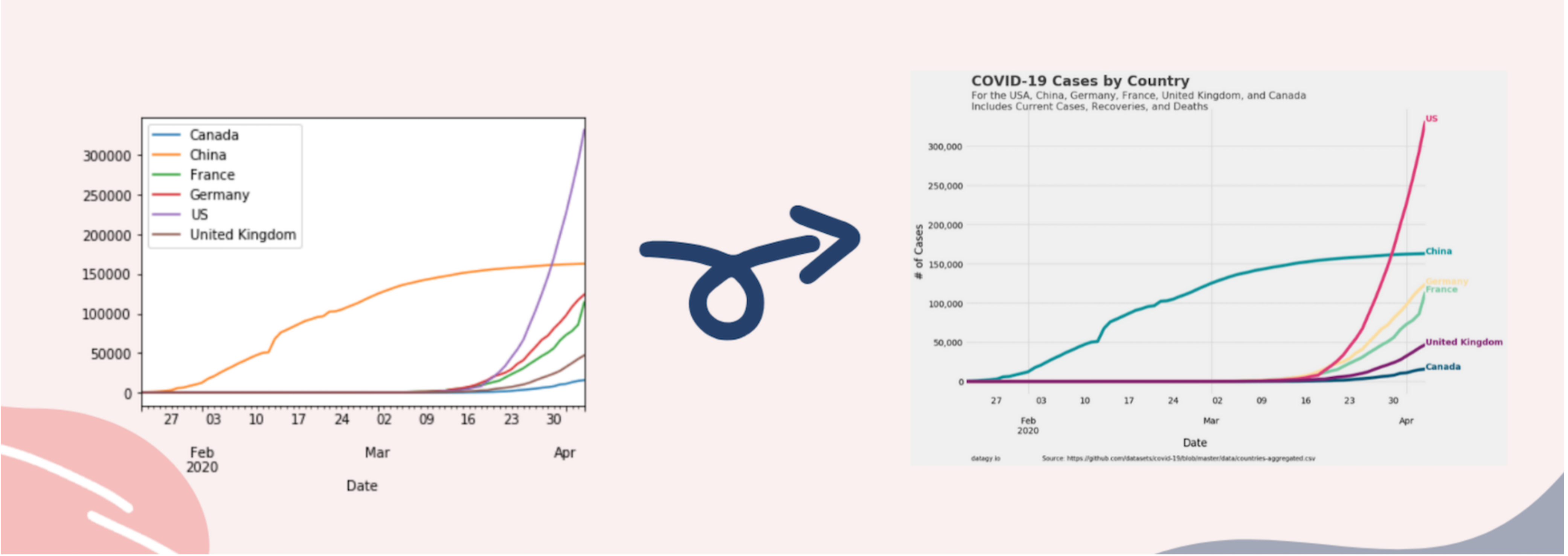
## Visualization in Python - Day 2

*"One should look for what is and not what he thinks should be."*
*-Albert Einstein.*

# Rising Data Science Interest

- With the rise of COVID-19 the need for fast visualizations to put data in perspective has risen also
- Have a look at this Python Visualization article while we wait to start class

***https://towardsdatascience.com/visualizing-covid-19-data-beautifully-in-python-in-5-minutes-or-less-affc361b2c6a***



# Recap

**Topics we learned so far:**

- Importance of visualizations in Python
- Cleaning the Costa Rican dataset using basic data cleaning procedures
- Reshaping data using Pandas
- Defined use cases in exploratory data analysis

**Today, we'll explore the differences between univariate plots and bivariate plots, create some basic visualizations in python and customize them. We'll also learn how to create a violin plot and a compound visualization.**

# Module completion checklist

| Objective | Complete |
|---|---|
| Visualizing data with matplotlib | |
| Create histograms, boxplots, and bar charts | |
| Create scatterplots | |
| Customize graphs | |
| Create violin plots | |
| Create compound visualizations in grid format | |

# Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- Let the `main_dir` be the variable corresponding to your `skillsoft-data-viz-with-python` folder

```
# Set `home_dir` to the root directory of your computer.
home_dir = os.path.expanduser("~")
# Set `main_dir` to the location of your `skillsoft-data-viz-with-python` folder.
main_dir = os.path.join(home_dir, "Desktop", "skillsoft-data-viz-with-python")
```

```
# Make `data_dir` from the `main_dir` and
# remainder of the path to data directory.
data_dir = os.path.join(main_dir, "data")

# Create a plot directory to save our plots
plot_dir = os.path.join(main_dir, "plots")
```

# Loading packages

- Load the packages we will be using

```
import pandas as pd
import numpy as np
import pickle
import os
import matplotlib.pyplot as plt
```

# Working directory

- Set working directory to `data_dir`

```
# Set working directory.
os.chdir(data_dir)
```

```
# Check working directory.
print(os.getcwd())
```

```
~/Desktop/skillsoft-data-viz-with-python/data
```

# Loading datasets

- Before creating visualizations in Python, let's load the cleaned Costa Rican data set, long and wide grouped data sets we pickled earlier

```
costa_viz = pickle.load(open("costa_viz.sav","rb"))
```

```
costa_grouped_mean_long = pickle.load(open("costa_grouped_mean_long.sav","rb"))
```

```
costa_grouped_mean_wide = pickle.load(open("costa_grouped_mean_wide.sav","rb"))
```

# Visualizing data with matplotlib

- `matplotlib` is a popular plotting library among scientists and data analysts
- It is one of the older Python plotting libraries, and for this reason, 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 comes with some plotting capabilities, and these are actually just based on `matplotlib`
- You can begin to explore the different types of plots you can create with `matplotlib` by browsing their ***gallery***

# Importing matplotlib

- We 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 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). |

# Univariate plots

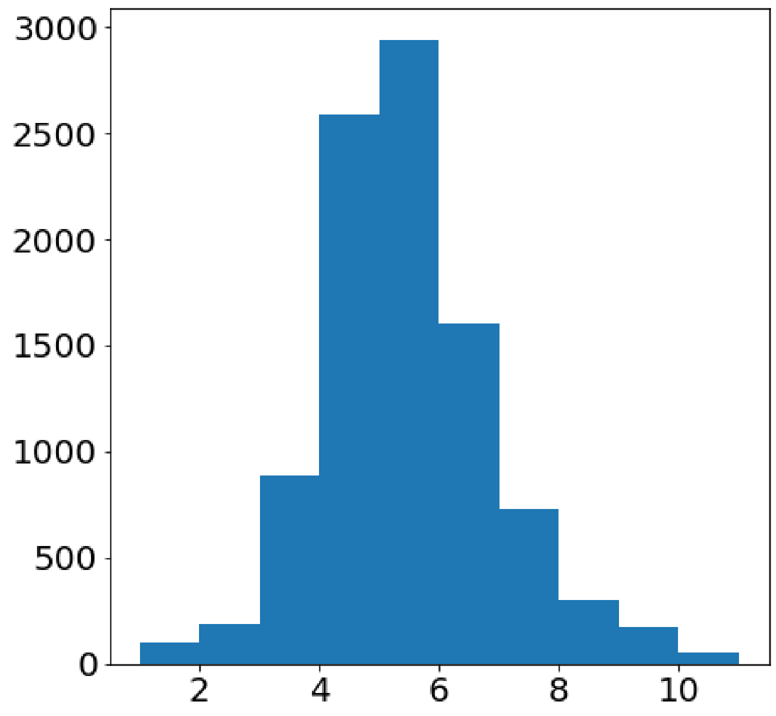
- Univariate plots are used to visualize distribution of a **single variable**
- They are used primarily in the initial stages of EDA when we would like to learn more about individual variables in our data
- They are also used in combination with other univariate plots to compare data distributions of different variables
- Univariate plots include the following popular graphs: `boxplot`, `histogram`, `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(costa_viz['rooms'])
plt.show()
```

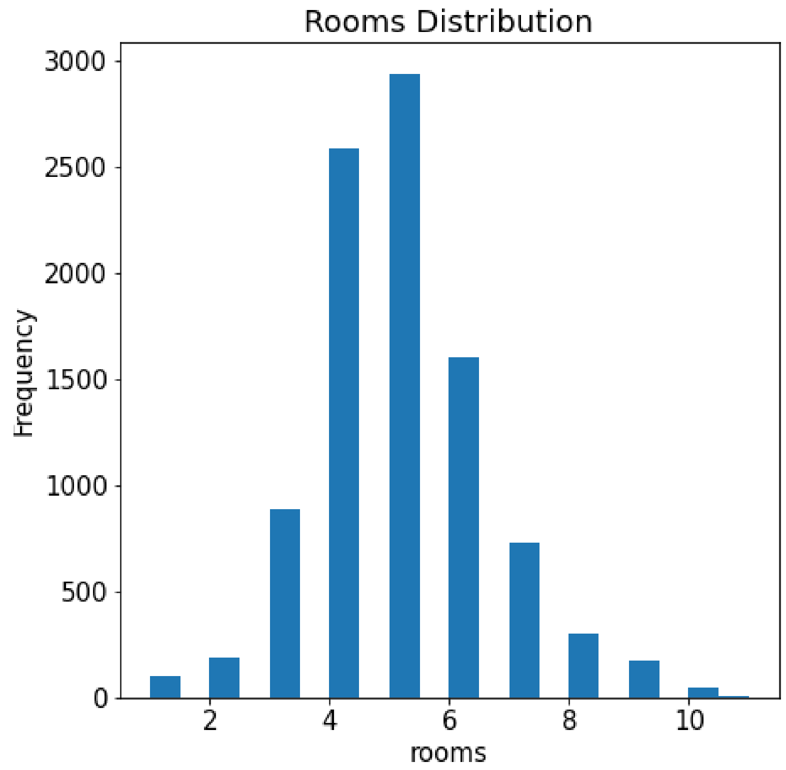
```
(array([  97.,  188.,  890., 2587., 2940.,
1607.,  732.,  298.,  168.,
          50.]), array([ 1.,  2.,  3.,  4.,  5.,
6.,  7.,  8.,  9., 10., 11.]), <a list of 10
Patch objects>)
```



# Univariate plots: histogram (cont'd)

- Bins represent the intervals in which we want to group the observations
- Control the number of bins with `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(costa_viz['rooms'], bins = 20)
plt.xlabel('rooms')                  #<- label x-axis
plt.ylabel('Frequency')              #<- label y-axis
plt.title('Rooms Distribution')      #<- add plot title
plt.show()
```



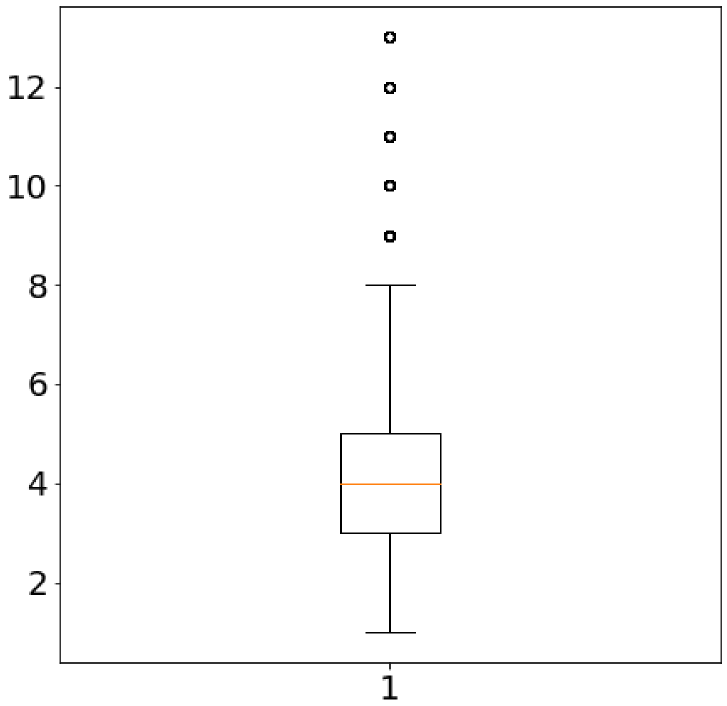
# Univariate plots: boxplot

- A boxplot is a visual summary of the 25th, 50th and 75th percentiles
- It also calculates an upper and lower threshold on what values should be considered outliers

```
plt.boxplot(costa_viz['ppl_total'])
```

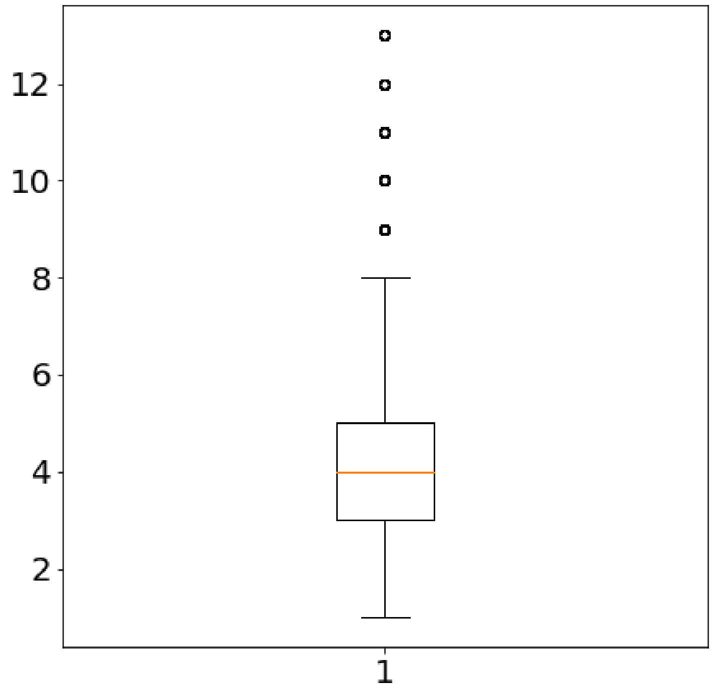
```
{'whiskers': [<matplotlib.lines.Line2D object at
0x7fa941518fd0>, <matplotlib.lines.Line2D object
at 0x7fa940c13bd0>], 'caps':
[<matplotlib.lines.Line2D object at
0x7fa920cce150>, <matplotlib.lines.Line2D object
at 0x7fa920cce690>], 'boxes':
[<matplotlib.lines.Line2D object at
0x7fa940c131d0>], 'medians':
[<matplotlib.lines.Line2D object at
0x7fa920ccec10>], 'fliers':
[<matplotlib.lines.Line2D object at
0x7fa920cd5190>], 'means': []}
```

```
plt.show()
```



# Univariate plots: boxplot interpretation

- The **orange line** shows the median of `ppl_total`
- The top and bottom of the box are the `75th` and `25th` percentile respectively
- The outermost lines are called the `whiskers`
- Values beyond whiskers are considered outliers - they are substantially outside the rest of the data



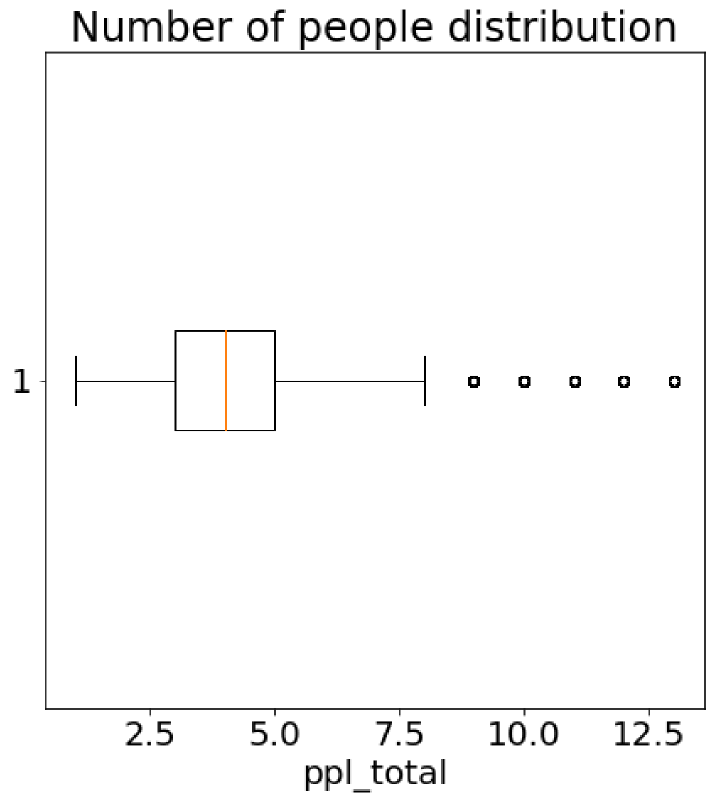
# Univariate plots: boxplot (cont'd)

- You can change to orientation of the plot to horizontal by setting `vert = False`
- **By looking at this boxplot, what can you tell about the `ppl_total` distribution in our data?**

```
plt.boxplot(costa_viz['ppl_total'], vert = False)
```

```
{'whiskers': [<matplotlib.lines.Line2D object at 0x7fa930ebdc50>, <matplotlib.lines.Line2D object at 0x7fa930ec2850>], 'caps': [<matplotlib.lines.Line2D object at 0x7fa930ec2f10>, <matplotlib.lines.Line2D object at 0x7fa930ec24d0>], 'boxes': [<matplotlib.lines.Line2D object at 0x7fa930f1ad90>], 'medians': [<matplotlib.lines.Line2D object at 0x7fa940bdbb10>], 'fliers': [<matplotlib.lines.Line2D object at 0x7fa940bdbc90>], 'means': []}
```

```
plt.xlabel('ppl_total')       #<- label x-axis
# Add plot title
plt.title('Number of people distribution')
plt.show()
```



# Univariate plots: bar chart

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

```
plt.bar(bar_positions,      #<- numpy array of positions
        bar_heights)        #<- list, numpy array, or pandas series of numbers
plt.xticks(bar_positions,   #<- numpy array of positions
           bar_labels)      #<- list or pandas series of character strings
```

# Univariate plots: bar chart (cont'd)

- When plotting bar charts of any complexity, the best type of data to use is **long** data
- Let's use our `costa_grouped_mean_long` data we created earlier to create a simple bar chart of the means of the variables

```
print(costa_grouped_mean_long.head())
```

```
   Target           metric       mean
0   False        ppl_total   4.358607
1    True        ppl_total   3.796531
2   False  dependency_rate  26.011233
3    True  dependency_rate  25.425284
4   False       num_adults   2.388093
```

- Let's filter `Target` as `True` and only keep two columns: `metric` and `mean`

```
costa_true_means = 
costa_grouped_mean_long.query('Target == True')
[['metric','mean']]
print(costa_true_means)
```

```
            metric       mean
1        ppl_total   3.796531
3  dependency_rate  25.425284
5       num_adults   2.713809
7            rooms   5.205971
9              age  36.078886
```

# Univariate plots: bar chart (cont'd)

Let's now get the data we need and assign it to the three variables for convenience and clarity

1. The **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. The **bar positions** are going to be a range of numbers from based on the number of categories (i.e. bars)

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

# Univariate plots: bar chart (cont'd)

```
print(bar_labels)
```

```
1          ppl_total
3    dependency_rate
5         num_adults
7              rooms
9                age
Name: metric, dtype: object
```

```
print(bar_positions)
```

```
[0 1 2 3 4]
```

```
print(bar_heights)
```

```
1     3.796531
3    25.425284
5     2.713809
7     5.205971
9    36.078886
Name: mean, dtype: float64
```

# Univariate plots: bar chart (cont'd)

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

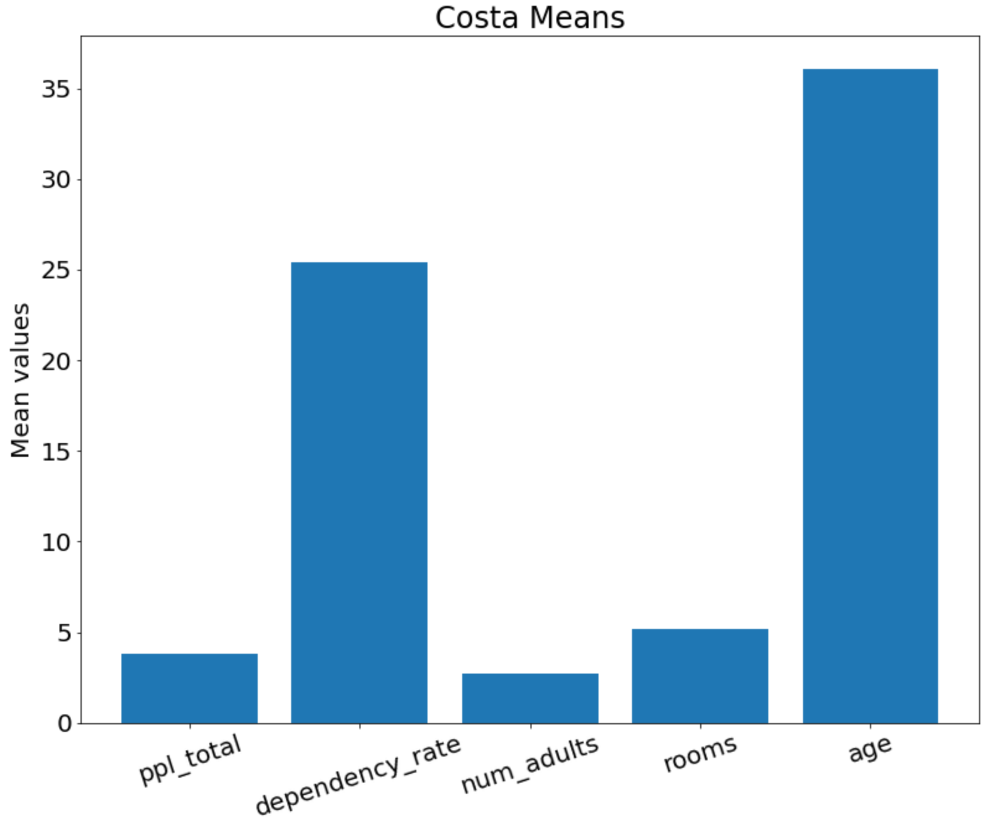
```
# Adjust figure size before plotting.
plt.figure(figsize = (13, 10))
plt.bar(bar_positions, bar_heights)
```

```
<BarContainer object of 5 artists>
```

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

```
([<matplotlib.axis.XTick object at
0x7fa941bd8490>, <matplotlib.axis.XTick
object at 0x7fa941bd8450>,
<matplotlib.axis.XTick object at
0x7fa941bd5b10>, <matplotlib.axis.XTick
object at 0x7fa910a50f10>,
<matplotlib.axis.XTick object at
0x7fa910a59490>], [Text(0, 0, 'ppl_total'),
Text(0, 0, 'dependency_rate'), Text(0, 0,
'num_adults'), Text(0, 0, 'rooms'), Text(0,
0, 'age')])
```

```
plt.ylabel('Mean values')
plt.title('Costa Means')  #<- add plot title
plt.show()
```

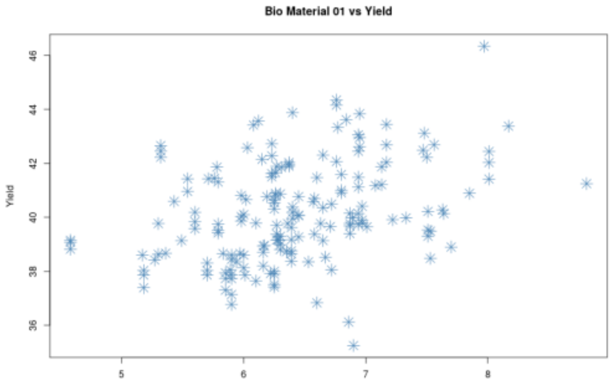


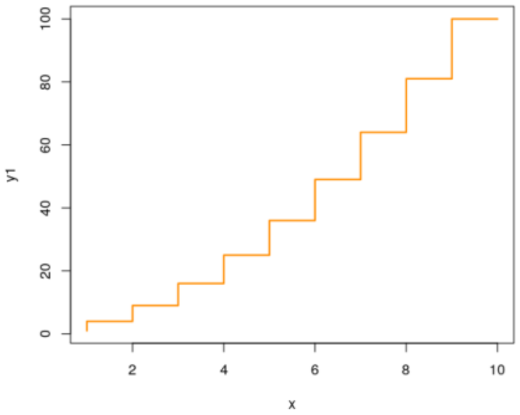
# Module completion checklist

| Objective | Complete |
| --- | --- |
| Visualizing data with matplotlib | ✔ |
| Create histograms, boxplots, and bar charts | ✔ |
| Create scatterplots | |
| Customize graphs | |
| Create violin plots | |
| Create compound visualizations in grid format | |

# Bivariate plots

- Bivariate plots are used to visualize data distribution and relationships between **two variables**
- They are used heavily throughout different stages of EDA **to learn more about how one variable is related to another**
- They are also used **in combination with other bivariate plots to compare relationships between different pairs of variables**
- Bivariate plots include `scatterplots` and `line graphs`

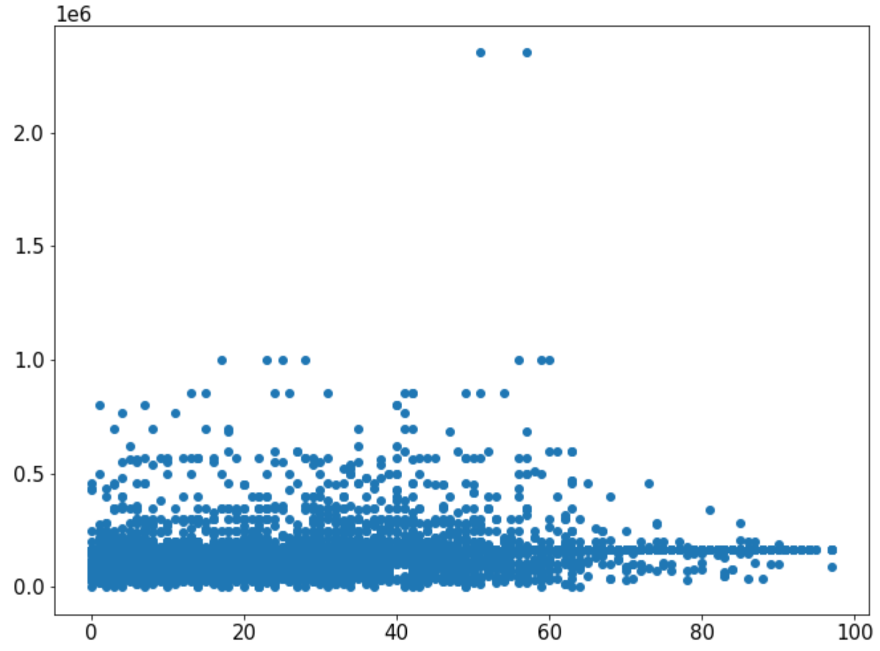


# Bivariate plots: scatterplot

- A scatterplot is the most **common bivariate plot** type
- It's one of the most popular plots in scientific computing, machine learning, and data analysis
- Great for showing **patterns between 2 variables** (hence *bivariate*)
- Let's plot `age` against `monthly_rent` for each observation
- Takes an array of `x` values and an array of `y` values

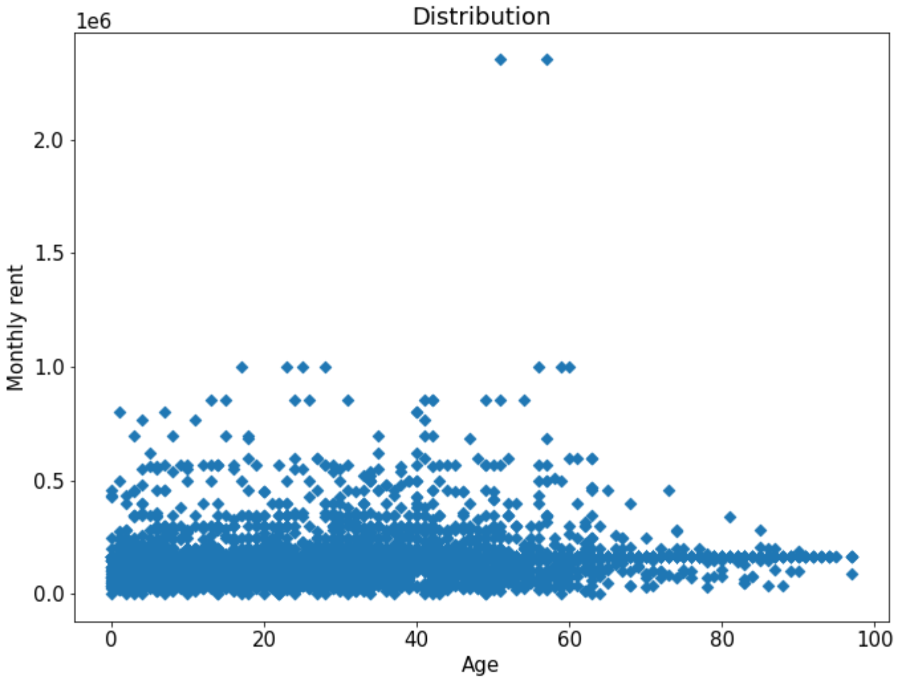
```
plt.scatter(costa_viz['age'],
            costa_viz['monthly_rent'])
plt.show()
```



# Bivariate plots: scatterplot (cont'd)

- You can change the marker type to a shape other than a point
- For a list of marker and line types, see ***documentation***

```
plt.scatter(costa_viz['age'],
            costa_viz['monthly_rent'],
            marker = "D") #<- set marker type to
diamond
plt.xlabel('Age')
plt.ylabel('Monthly rent')
plt.title('Distribution')
plt.show()
```




- By looking at this scatterplot, what patterns do you see in the relationship between the two variables?

# Knowledge check 1

# Exercise 1

# Module completion checklist

| Objective | Complete |
| --- | --- |
| Visualizing data with matplotlib | ✔ |
| Create histograms, boxplots, and bar charts | ✔ |
| Create scatterplots | ✔ |
| Customize graphs | |
| Create violin plots | |
| Create compound visualizations in grid format | |

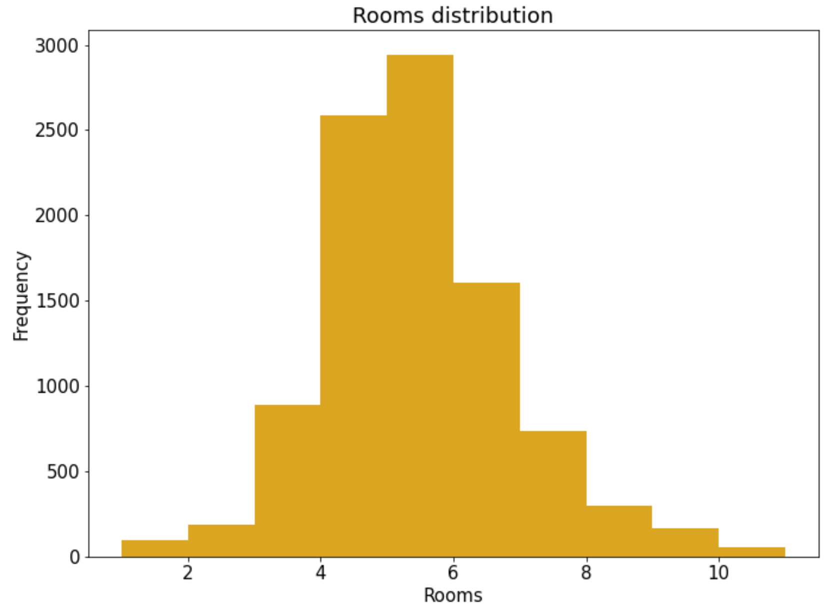
# Customize colors

- You can also change the color of the marker by setting an argument specific to visualization type
- The basic options are `b` (blue), `g` (green), `r` (red), `c` (cyan), `m` (magenta), `y` (yellow), `k` (black), and `w` (white)
- You can also use any color by providing its ***RGB code***
- The list of named colors in `matplotlib` is also available in this handy ***reference table / color map visualization***

# Customize color: histogram

- To change the color of a histogram, add an argument `facecolor` and then set it to the color of your choice

```
plt.hist(costa_viz['rooms'],
            facecolor = 'goldenrod') #<- set color
plt.xlabel('Rooms')
plt.ylabel('Frequency')
plt.title('Rooms distribution')
plt.show()
```



# Customize color: bar chart

- To change the color of a bar chart, add an argument `color` and then set it to the color of your choice

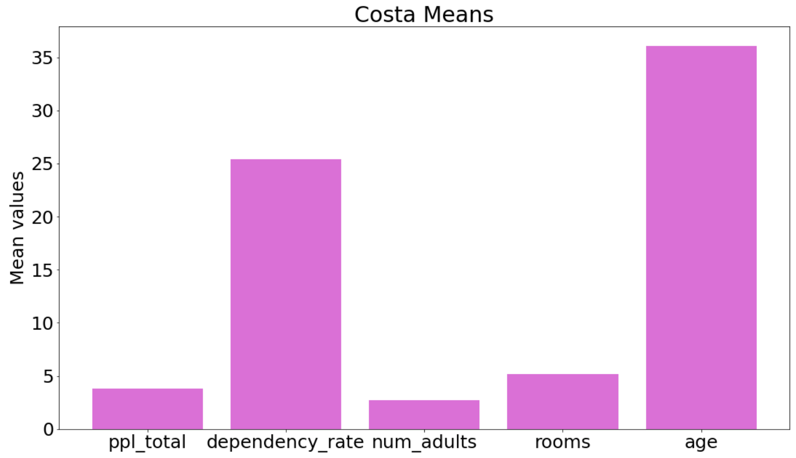
```
plt.figure(figsize = (18, 10))
plt.bar(bar_positions,
        bar_heights,
        color = "orchid")
```

```
<BarContainer object of 5 artists>
```

```
plt.xticks(bar_positions, bar_labels)
```

```
([<matplotlib.axis.XTick object at 0x7fa941a9f090>,
<matplotlib.axis.XTick object at 0x7fa941be1c50>,
<matplotlib.axis.XTick object at 0x7fa910a5e210>,
<matplotlib.axis.XTick object at 0x7fa941abfa90>,
<matplotlib.axis.XTick object at 0x7fa941abfe90>],
[Text(0, 0, 'ppl_total'), Text(0, 0,
'dependency_rate'), Text(0, 0, 'num_adults'), Text(0,
0, 'rooms'), Text(0, 0, 'age')])
```

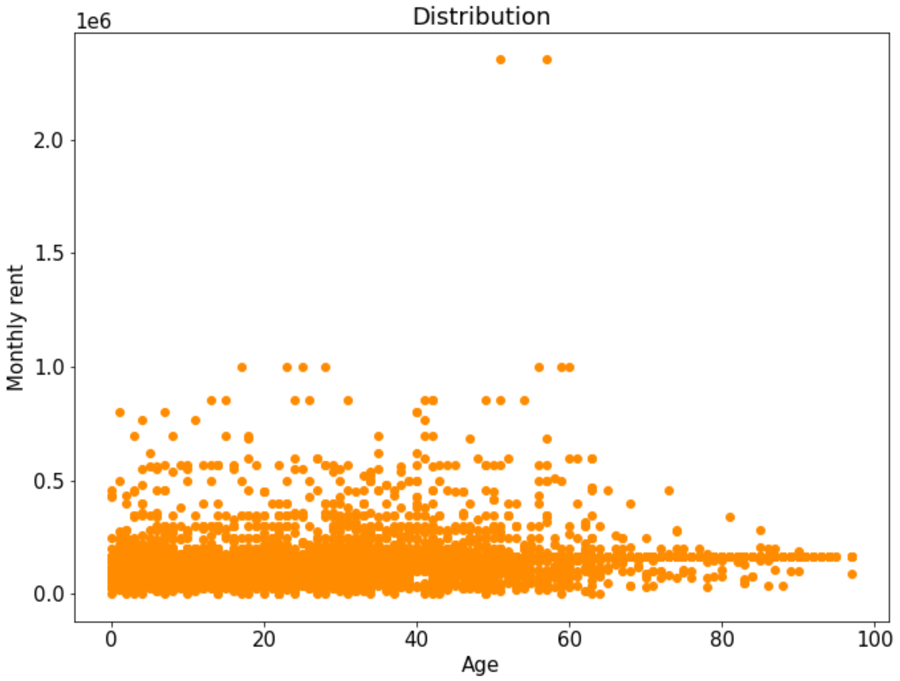
```
plt.ylabel('Mean values')
plt.title('Costa Means')
plt.show()
```



# Customize color: scatterplot

- To change the color of a scatterplot, add an argument `c` and then set it to the color of your choice

```
plt.scatter(costa_viz['age'],
            costa_viz['monthly_rent'],
            c = 'darkorange') #<- set marker
type to diamond
plt.xlabel('Age')
plt.ylabel('Monthly rent')
plt.title('Distribution')
plt.show()
```



# Customize color: map colors

- When plotting data using scatterplots, we might want to see values corresponding to 2 or more distinct categories
- We can achieve that by coloring observations that belong to different categories

```
print(costa_viz.head())
```

```
   ppl_total  dependency_rate  num_adults  rooms  age   monthly_rent  Target
0          1               37           1      3   43  190000.000000    True
1          1               36           1      4   67  135000.000000    True
2          1               36           1      8   92  165231.606971    True
3          4               38           2      5   17  180000.000000    True
4          4               38           2      5   37  180000.000000    True
```

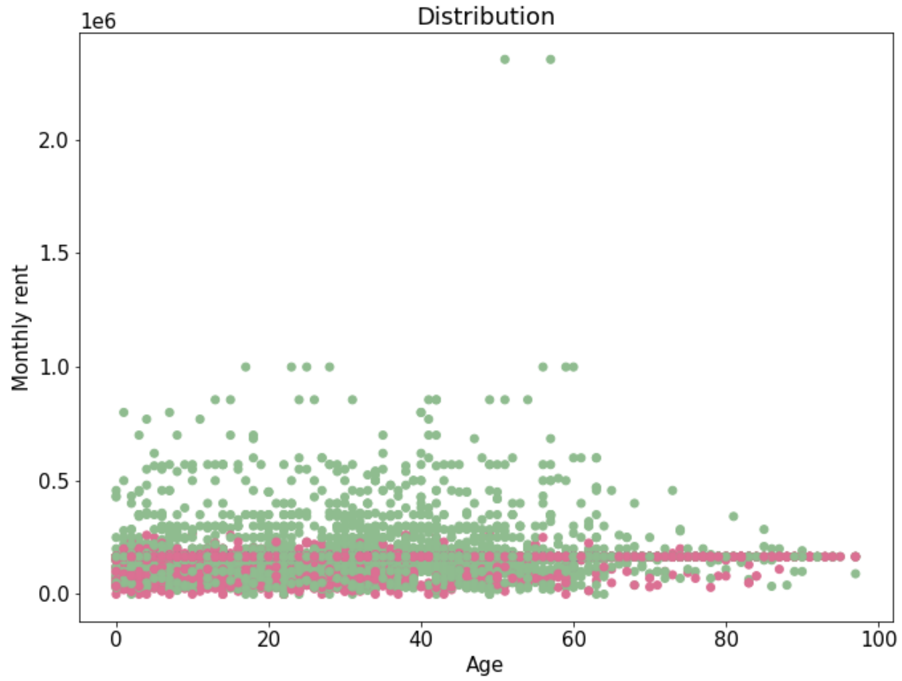
- In this example, we could color the observations based on `Target` binary variable
- Let's add a new column to the dataframe called `color` with
  - `True` corresponding to `darkseagreen` color, and
  - `False` corresponding to `palevioletred` color

# Customize color: map colors (cont'd)

```
color_dict = {True: 'darkseagreen',
              False: 'palevioletred'}
color = costa_viz['Target'].map(color_dict)
print(color.head())
```

```
0    darkseagreen
1    darkseagreen
2    darkseagreen
3    darkseagreen
4    darkseagreen
Name: Target, dtype: object
```

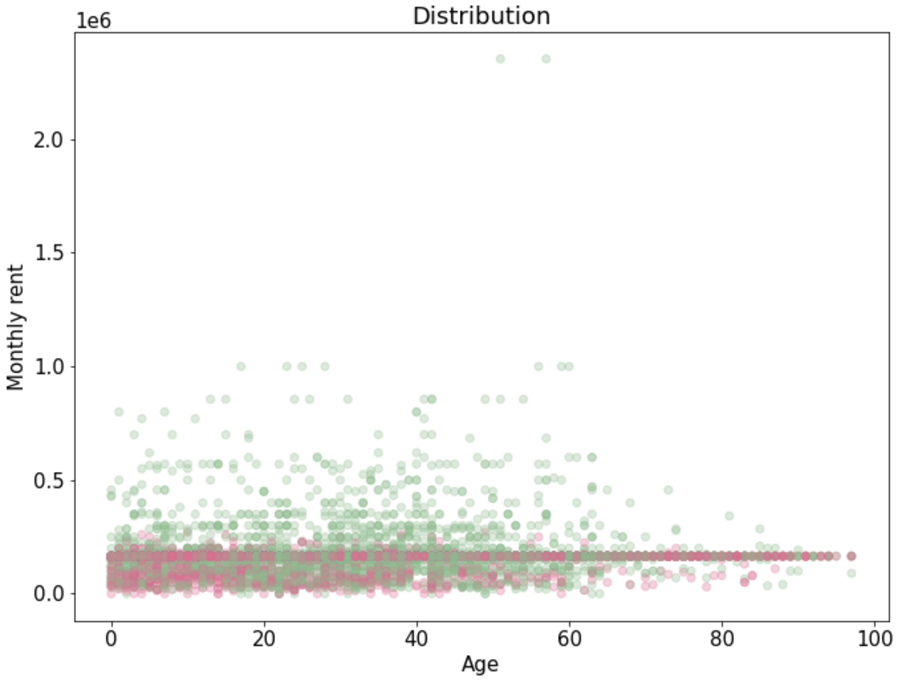
```
plt.scatter(costa_viz['age'],
            costa_viz['monthly_rent'],
            c = color)
plt.xlabel('Age')
plt.ylabel('Monthly rent')
plt.title('Distribution')
plt.show()
```



# Customize color: opacity

- When plotting many data points on one graph, lots of them get overplotted on top of each other
- That makes it difficult to discern how many observations are in the “clumps”
- One way to address overplotting is by setting the `alpha` parameter, which is responsible for regulating the **opacity** of the color
- It must be a value between `0` and `1`, where `0` is **transparent** and `1` is **opaque**

```
plt.scatter(costa_viz['age'],
            costa_viz['monthly_rent'],
            c = color,
            alpha = 0.3)
plt.xlabel('Age')
plt.ylabel('Monthly rent')
plt.title('Distribution')
plt.show()
```



# Customize plot settings: available styles

- There are a number of pre-defined styles provided by `matplotlib`
- You can preview available styles by running the following command

```
# Print all available styles.
print(plt.style.available)
```

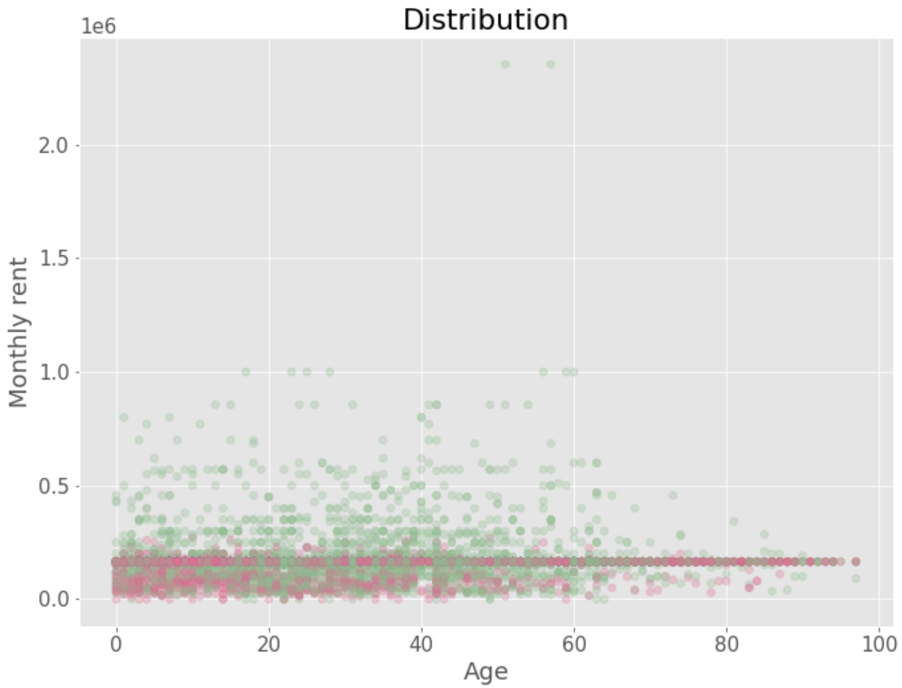
```
['Solarize_Light2', '_classic_test_patch', 'bmh', 'classic', 'dark_background', 'fast',
'fivethirtyeight', 'ggplot', 'grayscale', 'seaborn', 'seaborn-bright', 'seaborn-colorblind', 'seaborn-
dark', 'seaborn-dark-palette', 'seaborn-darkgrid', 'seaborn-deep', 'seaborn-muted', 'seaborn-notebook',
'seaborn-paper', 'seaborn-pastel', 'seaborn-poster', 'seaborn-talk', 'seaborn-ticks', 'seaborn-white',
'seaborn-whitegrid', 'tableau-colorblind10']
```

- You can see that one of the styles available is called "ggplot", which emulates the aesthetics of `ggplot2`, one of the most widely used plotting libraries in `R`
- To use this style, run the following command

```
# Use ggplot style in matplotlib.
plt.style.use('ggplot')
```

# Customize plot settings: test ggplot style

```
plt.scatter(costa_viz['age'],
            costa_viz['monthly_rent'],
            c = color,
           alpha = 0.3)
plt.xlabel('Age')
plt.ylabel('Monthly rent')
plt.title('Distribution')
plt.show()
```



# Customize plot settings: changing other presets

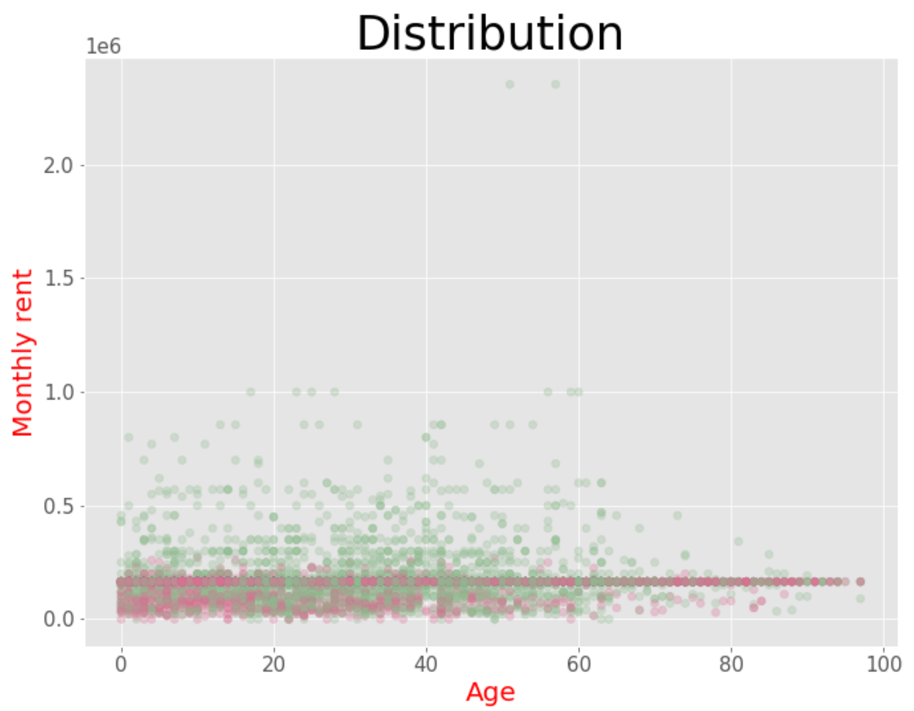
- As with all other plotting libraries, `matplotlib` comes with some pre-set defaults for all things you see in your plot
- To adjust any pre-set defaults, we will use `plt.rcParams` variable, which is a dictionary-like object
- You can either set those parameters on one-off basis or you can create a file with your presets and save it for your use for every project you work on (we will not cover it in class, but you can find more information about it including a sample file ***here***)

# Customize plot settings: labels

- The most common thing you would adjust is the **label** appearance for the following
  - `x-` and `y-axis`
  - `x-` and `y-axis` ticks
  - title

```
plt.rcParams['axes.labelsize'] = 20
plt.rcParams['axes.labelcolor'] = 'red'
plt.rcParams['axes.titlesize'] = 35
```

```
plt.scatter(costa_viz['age'],
            costa_viz['monthly_rent'],
            c = color,
            alpha = 0.3)
plt.xlabel('Age')
plt.ylabel('Monthly rent')
plt.title('Distribution')
plt.show()
```

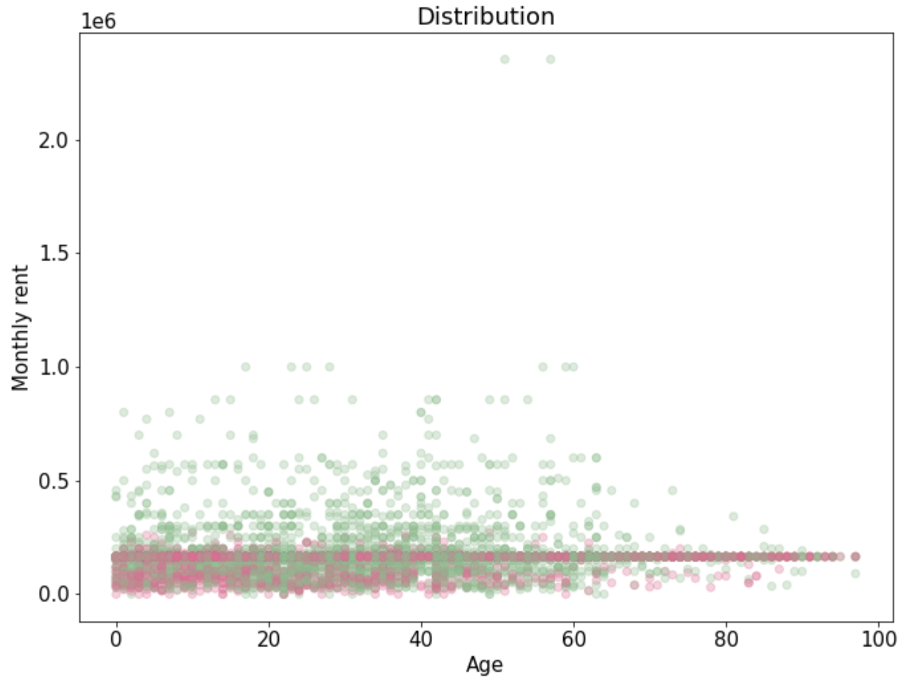


# Customize plot settings: reset defaults

- We have obviously updated the labels, but not necessarily in a good way
- When you need to reset the `rcParams` to default, we can use this function

```
plt.rcdefaults()
```

```
plt.scatter(costa_viz['age'],
            costa_viz['monthly_rent'],
            c = color,
           alpha = 0.3)
plt.xlabel('Age')
plt.ylabel('Monthly rent')
plt.title('Distribution')
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
# default face and edge color, default tick sizes,
# default fontsizes for ticklabels, and so on.  See
# http://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 linewidth
#axes.grid             : False    # display grid or not
#axes.titlesize        : large    # fontsize of the axes title
#axes.titlepad         : 6.0      # pad between axes and title in points
#axes.labelsize        : medium   # fontsize of the x any y labels
#axes.labelpad         : 4.0      # space between label and axis
#axes.labelweight      : normal   # weight of the x and y labels
#axes.labelcolor       : black
#axes.axisbelow        : 'line'   # draw axis gridlines and ticks below
                                  # patches (True); above patches but below
                                  # lines ('line'); or above all (False)
```

# Knowledge check 2

# Exercise 2

# Module completion checklist

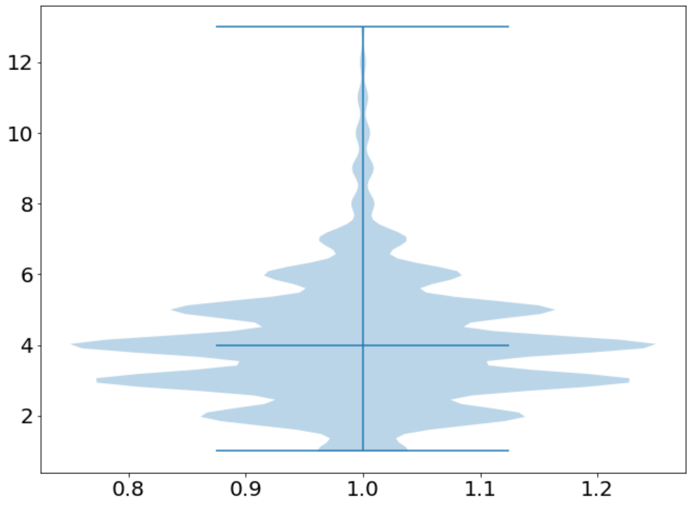
| Objective | Complete |
|---|---|
| Visualizing data with matplotlib | ✔ |
| Create histograms, boxplots, and bar charts | ✔ |
| Create scatterplots | ✔ |
| Customize graphs | ✔ |
| Create violin plots | |
| Create compound visualizations in grid format | |

# Complex univariate plots: violin plots

- Violin plots are primarily used to look at the **variations in the data**
- The characteristics of violin plot are similar to the box plot, except they visualize the **probability density** of the entire data
- Just like box plots, they consist of a marker which shows the median
- The violin plot has elongated projections when the density is high and flat projections when the probability density is low
- The attributes `showmeans` and `showmedians` can be set to true or false to show the mean/median and vice versa

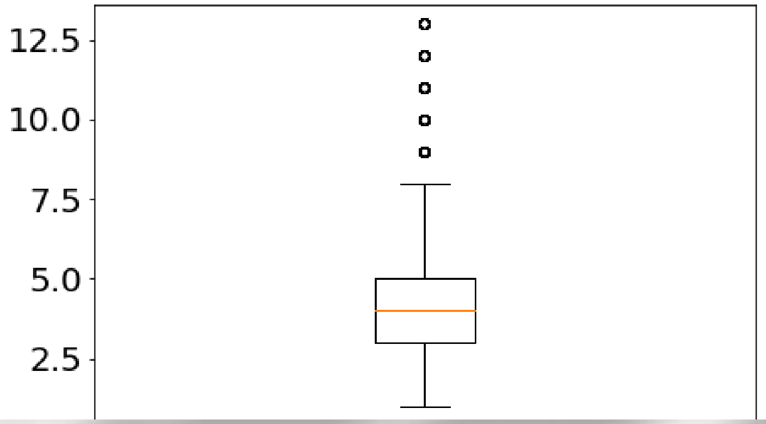
```
plt.violinplot(costa_viz['ppl_total'],
               showmeans=False,
               showmedians=True)
```

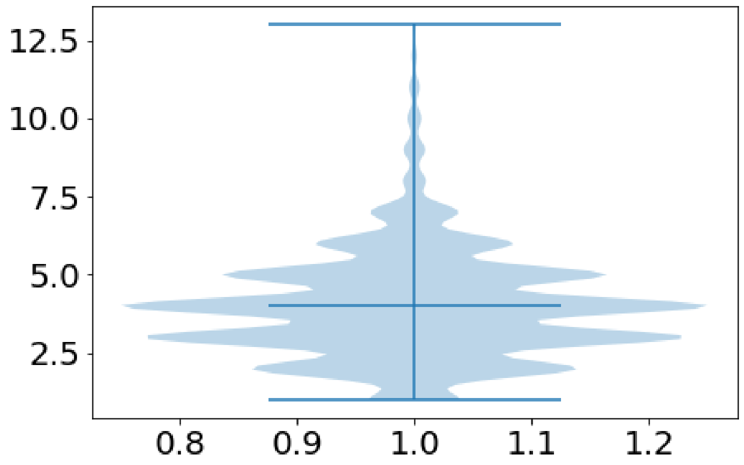
```
plt.show()
```



# Univariate plots: violin plot interpretation

- The **blue line** shows the median of `ppl_total`
- The **immediate areas around the median of the violin plot where the probability density is higher represent the `25th` and `75th` percentile approximately**
- By comparing the box plot we created earlier with the violin plot, we understand that violin plot is lot more useful to understand the exact probability distribution of data



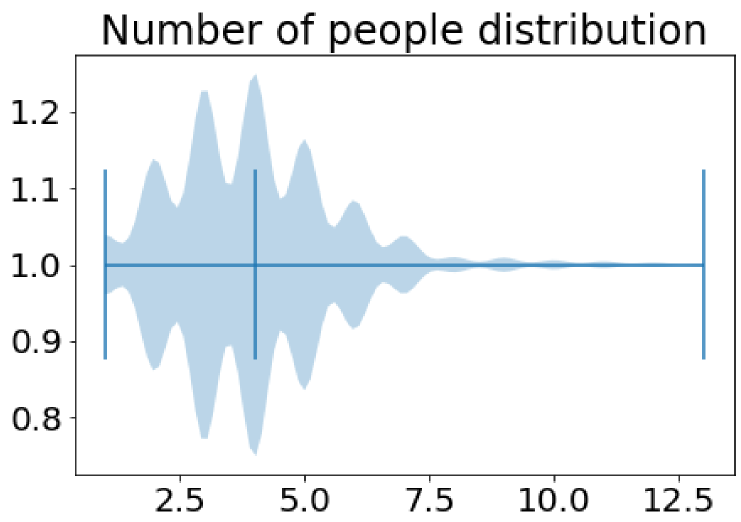
# Univariate plots: violin plot (cont'd)

- You can change the orientation of the plot to horizontal by setting `vert = False`
- **By looking at this violin plot, what can you tell about the `ppl_total` distribution in our data?**

```
plt.violinplot(costa_viz['ppl_total'], vert =
False, showmeans=False, showmedians=True)
```

```
{'bodies':
[<matplotlib.collections.PolyCollection object
at 0x7fa934525f50>], 'cmaxes':
<matplotlib.collections.LineCollection object at
0x7fa9007eb210>, 'cmins':
<matplotlib.collections.LineCollection object at
0x7fa9007eb490>, 'cbars':
<matplotlib.collections.LineCollection object at
0x7fa9007eb410>, 'cmedians':
<matplotlib.collections.LineCollection object at
0x7fa9007eb910>}
```

```
# Add plot title
plt.title('Number of people distribution')
plt.show()
```



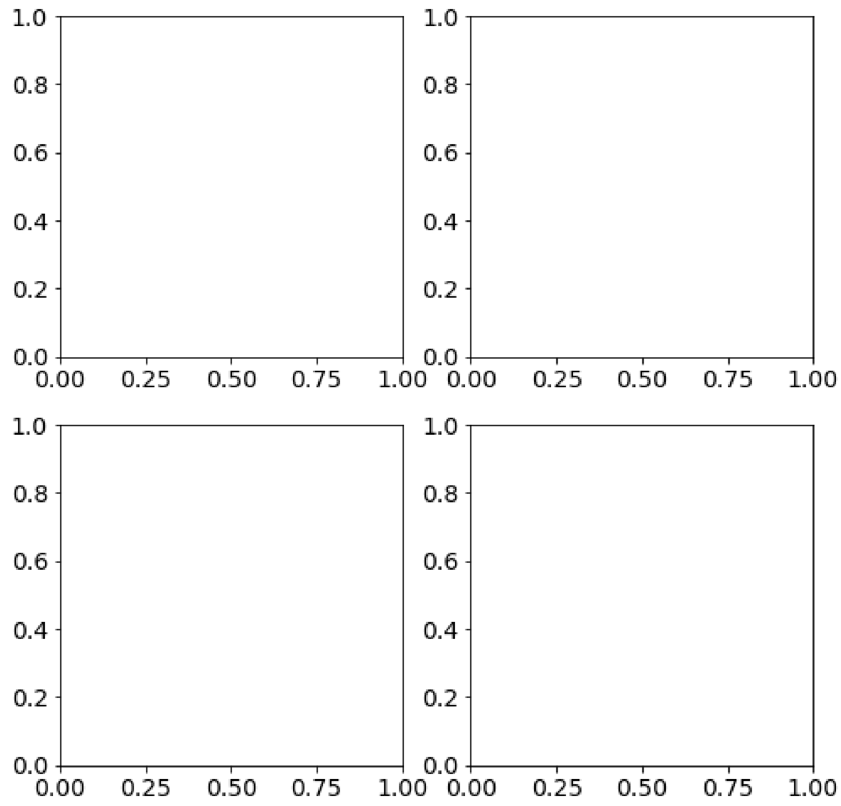
# Module completion checklist

| Objective | Complete |
|---|---|
| Visualizing data with matplotlib | ✔ |
| Create histograms, boxplots, and bar charts | ✔ |
| Create scatterplots | ✔ |
| Customize graphs | ✔ |
| Create violin plots | ✔ |
| Create compound visualizations in grid format | |

# Compound visualizations: grids

- We can create figures containing multiple plots, laid out in a *grid*, using `plt.subplots()`
- The `subplots` function returns two values, a **Figure** object and a **Axes** object
  - The **Figure** contains the entire grid and all of the elements inside
  - The **Axes** is an array, where each member contains a particular subplot
- **Why do you think grid or compound visualizations are useful?**
- **Where would you use such visualizations in your work?**

```
# Create a 2 x 2 figure and axes grid.
fig, axes = plt.subplots(2, 2)
plt.show()
```



# Compound visualizations: axes

- `Axes` is just an array

```
print(axes)
```

```
[[<matplotlib.axes._subplots.AxesSubplot object at 0x7fa9007eb6d0>
  <matplotlib.axes._subplots.AxesSubplot object at 0x7fa8f0106410>]
 [<matplotlib.axes._subplots.AxesSubplot object at 0x7fa910b41990>
  <matplotlib.axes._subplots.AxesSubplot object at 0x7fa9116d8f10>]]
```

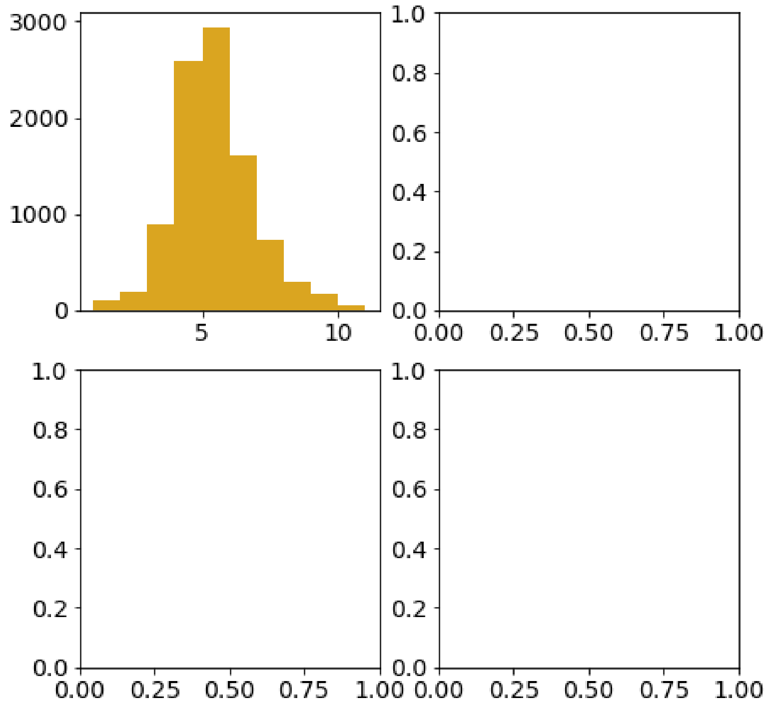
- Since it's a `2 x 2` grid, we have a 2D array with 4 entries that we will **"fill"** with values that are plots

# Compound visualizations: axes (cont'd)

- To access each element of the array, use simple 2D array subsetting style `[row_id, col_id]`
- Instead of attaching a particular plot like a histogram, for instance, to a `plt` object, we will attach it to the `axes[row_id, col_id]`

```
plt.figure(figsize = (8, 8))
fig, axes = plt.subplots(2, 2)
axes[0, 0].hist(costa_viz['rooms'],
                facecolor = 'goldenrod')
plt.show()
```

```
(array([  97.,  188.,  890., 2587., 2940.,
1607.,  732.,  298.,  168.,
         50.]), array([ 1.,  2.,  3.,  4.,  5.,
6.,  7.,  8.,  9., 10., 11.]), <a list of 10
Patch objects>)
```



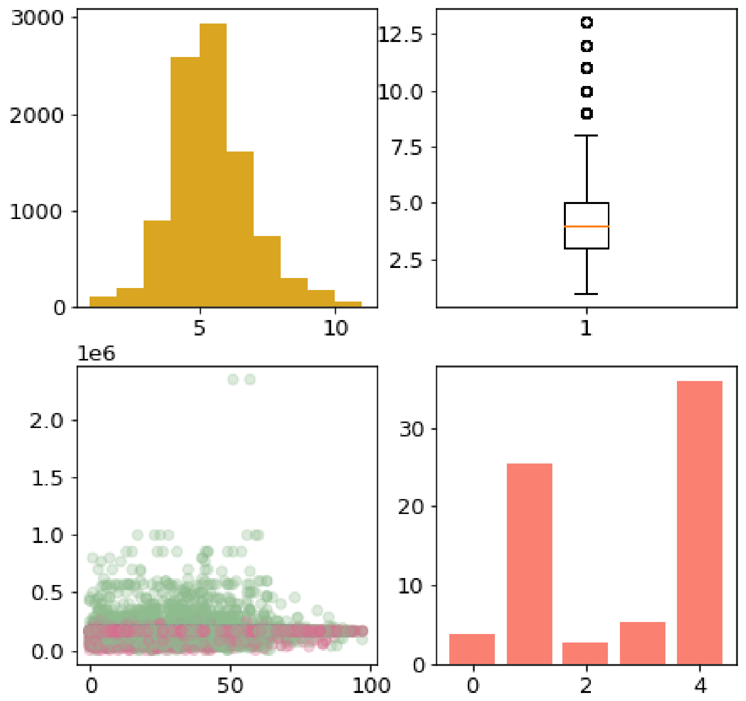
# Compound visualizations: axes (cont'd)

- Let's fill out three remaining plots

```
plt.figure(figsize = (12, 8))
fig, axes = plt.subplots(2, 2)
axes[0, 0].hist(costa_viz['rooms'],
                facecolor = 'goldenrod') #
<- set color
axes[0, 1].boxplot(costa_viz['ppl_total'])
axes[1, 0].scatter(costa_viz['age'],

costa_viz['monthly_rent'],
                   c = color,
                   alpha = 0.3)
axes[1, 1].bar(bar_positions, bar_heights,
               color = "salmon")
```

```
plt.show()
```



# Compound visualizations: labeling axes

- To label each plot's axis, use `axes[row_id, col_id].set_xlabel` format

```
# Histogram of rooms distribution.
axes[0, 0].set_ylabel('Frequency')
axes[0, 0].set_xlabel('rooms')

# Boxplot of ppl_total.
axes[0, 1].set_ylabel('Total number of people')

# Scatterplot of distribution.
axes[1, 0].set_xlabel('Age')
axes[1, 0].set_ylabel('Monthly rent')

# Mean values of categories of variable means based on Target.
axes[1, 1].set_ylabel('Mean Costa values')
```

# Compound visualizations: labeling ticks

- To set ticks on each axis, use `axes[row_id, col_id].xaxis.set_ticks` format

```
# No labels for ticks for boxplot.
axes[0, 1].xaxis.set_ticklabels([""])
```

```
# Tick positions set to bar positions in bar chart.
axes[1, 1].xaxis.set_ticks(bar_positions)

# Tick labels set to bar categories in bar chart.
```

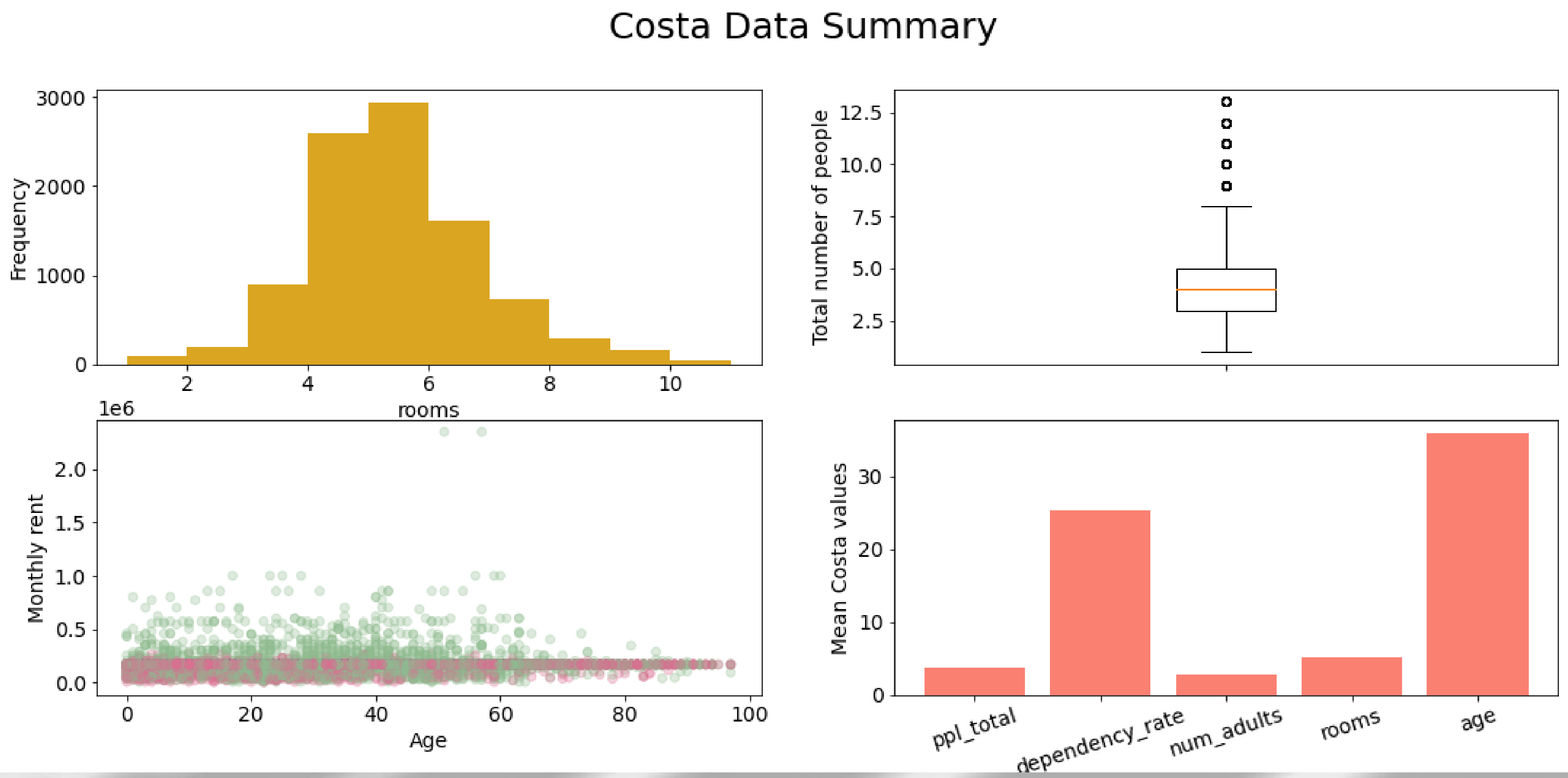
```
[<matplotlib.axis.XTick object at 0x7fa9344b4bd0>, <matplotlib.axis.XTick object at 0x7fa9344b4910>,
<matplotlib.axis.XTick object at 0x7fa941a92e10>, <matplotlib.axis.XTick object at 0x7fa941a924d0>,
<matplotlib.axis.XTick object at 0x7fa920b977d0>]
```

```
axes[1, 1].xaxis.set_ticklabels(bar_labels,rotation = 18)
```

```
[Text(0, 0, 'ppl_total'), Text(0, 0, 'dependency_rate'), Text(0, 0, 'num_adults'), Text(0, 0, 'rooms'),
Text(0, 0, 'age')]
```

# Compound visualizations: figure adjustments

```
plt.rcParams['axes.labelsize'] = 20
plt.rcParams['figure.titlesize'] = 25
fig.set_size_inches(18, 7.5)
fig.suptitle('Costa Data Summary')
plt.show()
```



# Knowledge check 3

# Exercise 3

# Module completion checklist

| Objective | Complete |
|---|---|
| Visualizing data with matplotlib | ✔ |
| Create histograms, boxplots, and bar charts | ✔ |
| Create scatterplots | ✔ |
| Customize graphs | ✔ |
| Create violin plots | ✔ |
| Create compound visualizations in grid format | ✔ |

# Summary

## So far, we have:

1. Visualized Costa Rican poverty dataset by using `matplotlib` package
2. Understood univariate and bivariate plots
3. Customized plots
4. Created a violin plot
5. Compounded multiple plots together

## What we will cover in the next session:

1. Saving plots and the data
2. Best practices for data visualization

# This completes our module

**Congratulations!**