DATA SOCIETY®

Introduction to visualization in python - day 1

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

Module completion checklist

Objective	Complete
Reshape data using pandas	
Define use cases of Exploratory Data Analysis (EDA)	
Create histograms, boxplots, and bar charts	
Create scatterplots	
Customize graphs	
Create compound visualizations	
Saving your plots and your data	
Best practices of data visualization	

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 `af-werx` folder

```
# Set `main_dir` to the location of your `af-werx` folder (for Linux).
main_dir = "/home/[username]/Desktop/af-werx"
```

```
# Set `main_dir` to the location of your `af-werx` folder (for Mac).
main_dir = '/Users/[username]/Desktop/af-werx'
```

```
# Set `main_dir` to the location of your `af-werx` folder (for Windows).
main_dir = "C:\\Users\\[username]\\Desktop\\af-werx"
```

```
# Make `data_dir` from the `main_dir` and
# remainder of the path to data directory.
data_dir = main_dir + "/data"

# Create a plot directory to save our plots
plot_dir = main_dir + "/plots"
```

Loading packages and setting working directory

- Load the packages we will be using

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

- Set working directory to `data_dir`

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

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

```
/home/[user-name]/Desktop/af-werx/data
```

Data wrangling and exploration

- Remember, a data scientist must be able to:
  - **Wrangle** the data (gather, clean, and sample data to get a suitable data set)
  - **Manage** the data for easy access by the organization
  - **Explore** the data to generate a hypothesis
- **The techniques we will learn today will help us achieve these goals!**

- **We will work with the Costa Rican dataset and see what we can discover. We will be:**
  - Cleaning the dataset
  - Wrangling the data for the purpose of visualizing the data and identifying patterns
  - Building static and interactive data visualizations

Costa Rica poverty dataset

- We are now going to explore the dataset we described in the case study. We will load in our Costa Rican poverty dataset as `costa_rica_poverty`. This dataset includes information about:
- A target variable which represents **income level**, in which the **goal is to understand the relationship between the individual and household characteristics and the resulting income level**
- Individuals and households in the dataset are characterized by variables ranging from features about the house they live in, gender split of the household, education, region and a few other features
- Of the 84 characteristics, there are
  - **21 features** about the person or household's home
  - **26 features** about the gender split within the household and about the household
  - **15 features** about region and education
  - *22 other features* that also seem to be potential poverty level indicators about the household

Load the dataset

- Let's load the entire dataset
- For visualizations, we will be taking a specific subset
- We are now going to use the function `read_csv` to read in our `costa_rican_poverty` dataset

```
household_poverty = pd.read_csv("costa_rica_poverty.csv")
print(household_poverty.head())
```

```
  household_id        ind_id  rooms  ...  age  Target  monthly_rent
0    21eb7fcc1  ID_279628684      3  ...   43       4      190000.0
1    0e5d7a658  ID_f29eb3ddd      4  ...   67       4      135000.0
2    2c7317ea8  ID_68de51c94      8  ...   92       4           NaN
3    2b58d945f  ID_d671db89c      5  ...   17       4      180000.0
4    2b58d945f  ID_d56d6f5f5      5  ...   37       4      180000.0

[5 rows x 84 columns]
```

- The entire dataset consists of `9557` observations and `84` variables

Subsetting data

- In this module, we will explore a subset of this dataset, which includes the following variables:
  - **ppl_total**
  - **dependency_rate**
  - **num_adults**
  - **monthly rent**
  - **rooms**
  - **age**
  - *Target*
- We are choosing these variables because they illustrate the concepts best
- However, you should be able to visualize and work with all of your data

Subsetting data

- Let's subset our data so that we have the variables we need
- We are keeping household_id, ppl_total, dependency_rate, num_adults, rooms, age, monthly_rent, and Target
- Let's name this subset costa_viz

```
costa_viz = household_poverty[['ppl_total', 'dependency_rate',
                               'num_adults', 'rooms', 'age', 'monthly_rent',
                               'Target']]
print(costa_viz.head())
```

```
   ppl_total  dependency_rate  num_adults  rooms  age  monthly_rent  Target
0          1               37           1      3   43      190000.0       4
1          1               36           1      4   67      135000.0       4
2          1               36           1      8   92           NaN       4
3          4               38           2      5   17      180000.0       4
4          4               38           2      5   37      180000.0       4
```

Data prep: clean NAs

- Depending on **subject matter**, missing values might mean something
- Let's define the choices on **how we can handle NAs in our data:**
  - drop columns that contain any `NA`s
  - drop columns with a certain % of `NA`s
  - impute missing values
  - convert column with missing values to categorical
- Let's look at the count of NAs by column first:

```
print(costa_viz.isnull().sum())
```

```
ppl_total              0
dependency_rate        0
num_adults             0
rooms                  0
age                    0
monthly_rent        6860
Target                 0
dtype: int64
```

Data cleaning: NAs

- `monthly_rent` has many `NA` values!
- We could just drop this column, as the number is over 50%
- However, in this instance, we'll keep it, and **impute missing values** using the mean of the column
- There isn't a mathematical method for a precise percentage of NAs that we are OK with
- **That's why your subject matter expertise is so important!**

```
# Set the dataframe equal to the imputed dataset.
costa_viz = costa_viz.fillna(costa_viz.mean())
# Check how many values are null in monthly_rent.
print(costa_viz.isnull().sum())
```

```
ppl_total          0
dependency_rate    0
num_adults         0
rooms              0
age                0
monthly_rent       0
Target             0
dtype: int64
```

Converting the target variable

- Let's convert poverty to a variable with two levels, which will help to balance it out
- The four original levels would also increase the complexity of the visualizations and the code
- For this reason, we will convert levels 1,2 and 3 to `vulnerable` and 4 to `non_vulnerable`
- The levels translate to 1, 2 and 3 as being **vulnerable** households
- Level 4 is **non vulnerable**

```
costa_viz['Target'] = np.where(costa_viz['Target'] <= 3, 'vulnerable', 'non_vulnerable')
```

```
print(costa_viz['Target'].head())
```

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

Data prep: target

- The next step of our data cleanup is to ensure the target variable is binary and has a label
- Let's look at the `dtype` of `Target`

```
print(costa_viz.Target.dtypes)
```

```
object
```

- We want to convert this to `bool` so that it is a binary class

```
costa_viz["Target"] = np.where(costa_viz["Target"] == "non_vulnerable", True, False)

# Check class again.
print(costa_viz.Target.dtypes)
```

```
bool
```

Data reshaping: wide vs long

- When we talk about data *reshaping*, what we usually mean is converting between what is called either **wide** or **long** data format
  - **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 `mean` of groups)
- Let's make a dataframe with two rows and six columns that looks like this, it represents a typical **wide** dataframe

Target	ppl_total	dependency_rate	num_adults	rooms	age
`False`	4.358607	26.011233	2.388093	4.533839	31.314238
`True`	3.796531	25.425284	2.713809	5.205971	36.078886

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:
  - `ppl_total`
  - `dependency_rate`
  - `num_adults`
  - `rooms`
  - `age`

Prepare data: group and summarize (cont'd)

```
# Group data by `Target` variable.
grouped = costa_viz.groupby('Target')
```

```
# Compute mean on the listed variables using the grouped data.
costa_grouped_mean = grouped.mean()[['ppl_total','dependency_rate','num_adults','rooms','age']]
print(costa_grouped_mean)
```

```
        ppl_total  dependency_rate  num_adults     rooms        age
Target
False    4.358607        26.011233    2.388093  4.533839  31.314238
True     3.796531        25.425284    2.713809  5.205971  36.078886
```

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

```
   Target  ppl_total  dependency_rate  num_adults     rooms        age
0   False   4.358607        26.011233    2.388093  4.533839  31.314238
1    True   3.796531        25.425284    2.713809  5.205971  36.078886
```

- The reason we call this dataframe **wide** is because each variable has its own column (i.e. `ppl_total`, `age`, etc)
- It makes the table easier to present, but is inconvenient to run analyses on or visualize

Why long?

- Now let's convert this **wide** data to the **long** format
- The `metric` variable, which was previously presented in 5 columns (i.e. `ppl_total`, `age`, etc), should be put into a single column
- The `mean` variable was the values in the columns corresponding to those variables
- That's the format we expect to get when we convert our **wide** dataframe to **long**
- This format is very convenient to work with when we run analysis and plot data

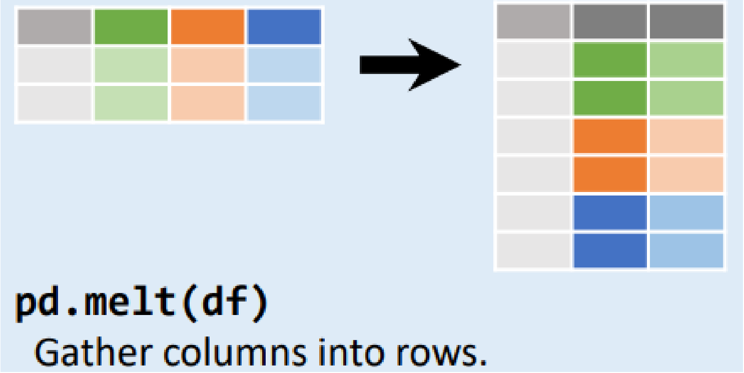
mrc_class	metric	mean
`False`	ppl_total	4.358607
`True`	ppl_total	3.796531
`False`	dependency_rate	26.011233
`True`	dependency_rate	25.425284
`False`	num_adults	2.388093
`True`	num_adults	2.713809
`False`	rooms	4.533839
`True`	rooms	5.205971
`False`	age	31.314238
`True`	age	36.078886

Wide to long format: melt

To convert from **wide** to **long** format, we use the Pandas `melt` function with the following arguments:

1. Wide dataframe
2. Variable(s) that will be preserved as the `ids` of the data (i.e. like `Target` with values `True` and `False` in our case)
3. Name of the variable that will now contain the column names from the wide data we want to melt together
4. Name of the column that will contain respective values corresponding to the melted columns

```
pd.melt(df,                                  #<- 1
        id_vars = ['id_col'],                #<- 2
        var_name = 'some_var',               #<- 3
        value_name = 'some_val')             #<- 4
```



Wide to long format: melt (cont'd)

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

```
   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
5    True       num_adults   2.713809
6   False            rooms   4.533839
7    True            rooms   5.205971
8   False              age  31.314238
9    True              age  36.078886
```

Long to wide format: pivot

```
df.pivot(index = ['id_col'],      #<- 1
         columns = 'some_var',    #<- 2
         values = 'some_val')     #<- 3
```

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 values of which column will be converted to column names
3. Lastly, we supply the `values` argument, which is the field to use to fill in the values of the wide data

Note: *There is a slight difference in syntax between `melt`, which is a Pandas function, and `pivot`, which is a method of a dataframe. You would say `pd.melt()` but `df.pivot()` where `df` corresponds to any dataframe!*

Long to wide format: pivot (cont'd)

```
# Melt the long data into wide.
costa_grouped_mean_wide = costa_grouped_mean_long.pivot(
                                                     index = 'Target',    #<- identifying variable
                                                     columns = 'metric', #<- col names of wide data
                                                     values = 'mean')     #<- values from above columns
print(costa_grouped_mean_wide)
```

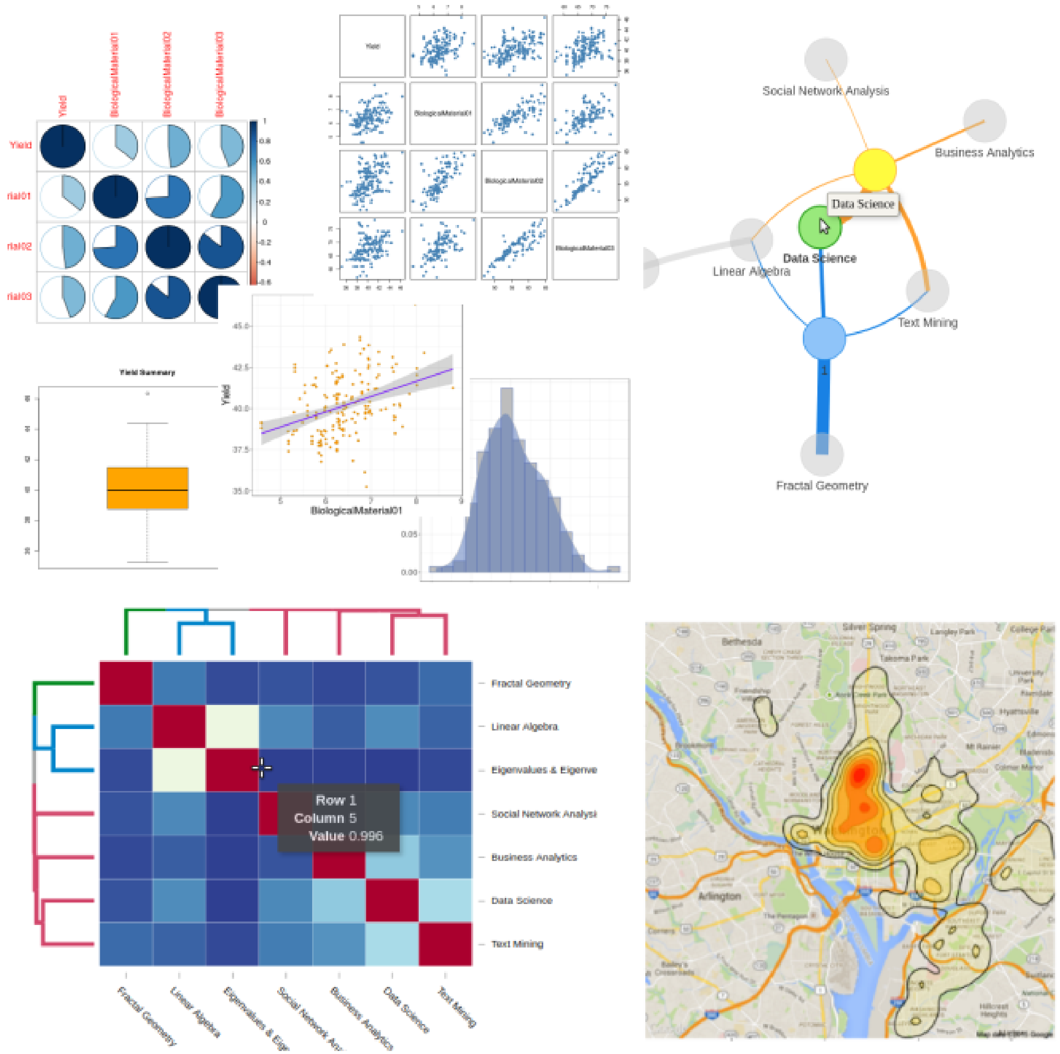
```
metric         age  dependency_rate  num_adults  ppl_total     rooms
Target
False    31.314238        26.011233    2.388093   4.358607  4.533839
True     36.078886        25.425284    2.713809   3.796531  5.205971
```

Module completion checklist

Objective	Complete
Reshape data using pandas	✔
Define use cases of Exploratory Data Analysis (EDA)	
Create histograms, boxplots, and bar charts	
Create scatterplots	
Customize graphs	
Create compound visualizations	
Saving your plots and your data	
Best practices of data visualization	

Why build a visualization?

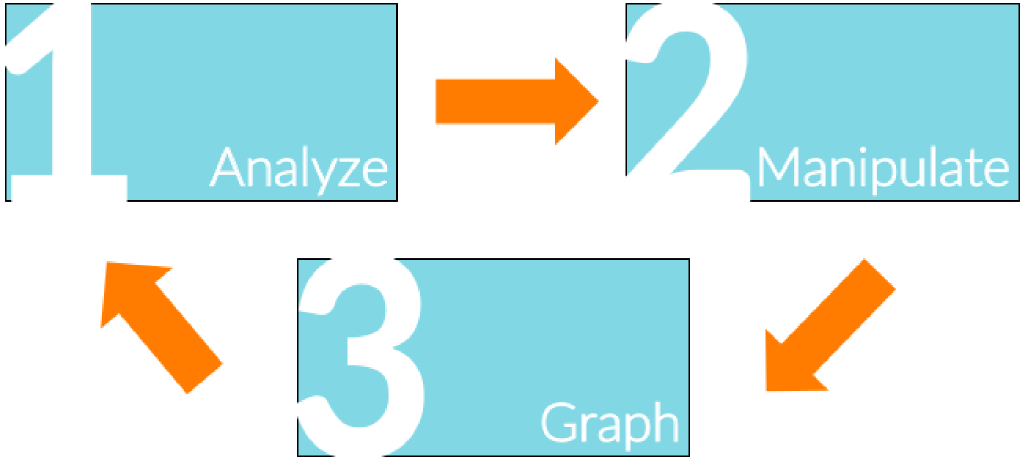
- To provide valuable insights that are **interpretable** and **relevant**
- To give a visual or graphical representation of data / concepts
- To communicate ideas
- To provide an accessible way to see and understand trends, outliers, and patterns in data
- To confirm a hypothesis about the data



Exploratory data analysis(EDA)

Python is a powerful tool for EDA because the graphics tie in with the functions used to analyze data. You can create graphs without breaking your train of thought as you explore your data. Visualization is an iterative process and consists of a few steps:

1. Analyze
2. Manipulate
3. Graph
4. Repeat



Exploratory data analysis in Python

Python's capabilities

1. Visualization tools available through multitudes of packages (e.g. `matplotlib`, `seaborn`)
2. The visualizations created are high quality graphics that can be saved as `SVG`, `PNG`, `JPEG`, `BMP`, `PDF`

What we will cover

1. Visualize Costa Rican poverty dataset by using `matplotlib` package
2. Save our graphs as `PNG` images
3. Interpret the graphs and create a story as if we were going to publish a report

Knowledge check 1

Exercise 1

Module completion checklist

Objective	Complete
Reshape data using pandas	✔
Define use cases of Exploratory Data Analysis (EDA)	✔
Create histograms, boxplots, and bar charts	
Create scatterplots	
Customize graphs	
Create compound visualizations	
Saving your plots and your data	
Best practices of data visualization	

Performing exploratory data analysis

- In these next sections, we will explore our dataset by visualizing different attributes and learning more about them
- It's a best practice to explore your data before you perform any analyses
- In this case, we're going to explore:
  - the distribution of the number of rooms
  - what is considered an outlier for the household sizes
  - the distribution of family size
  - mean values across variables
- Let's get started!

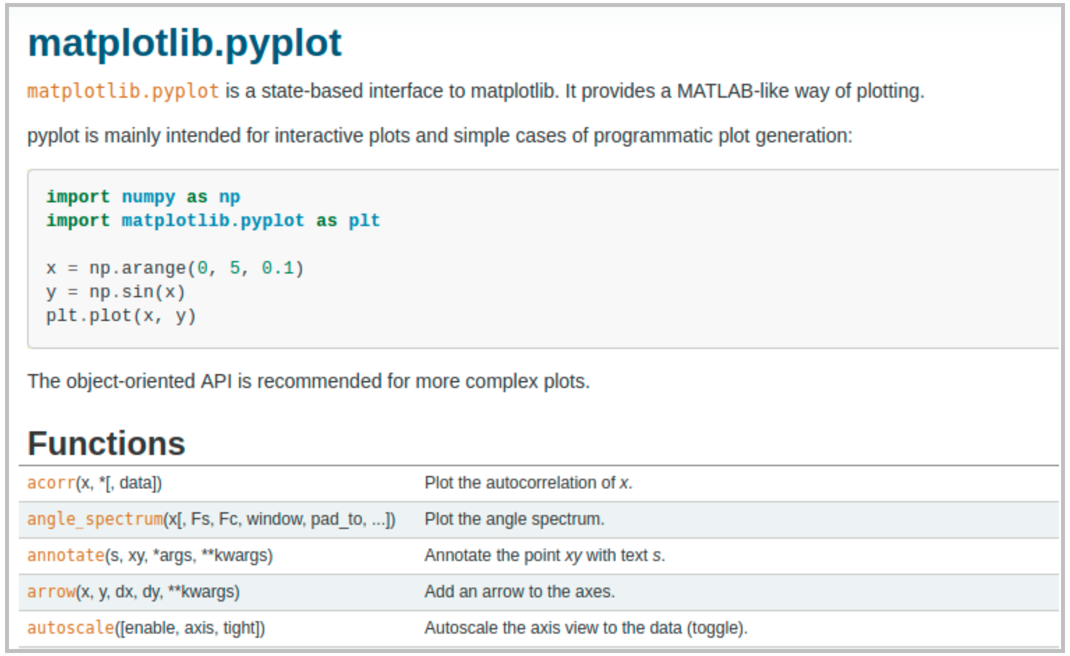
Visualizing data with matplotlib

- `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).

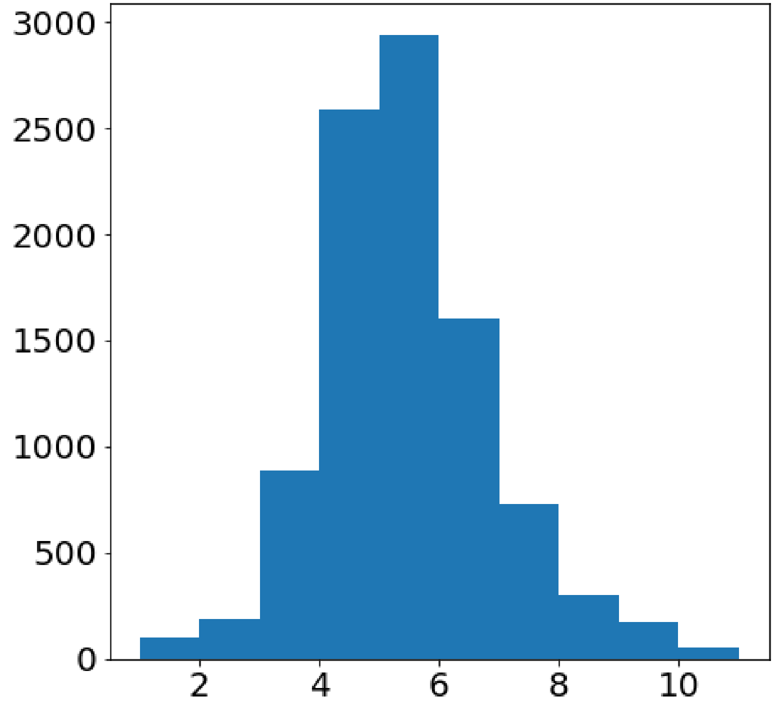
Univariate plots: histogram

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

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

- **What can we learn about the distribution of the number of rooms?**

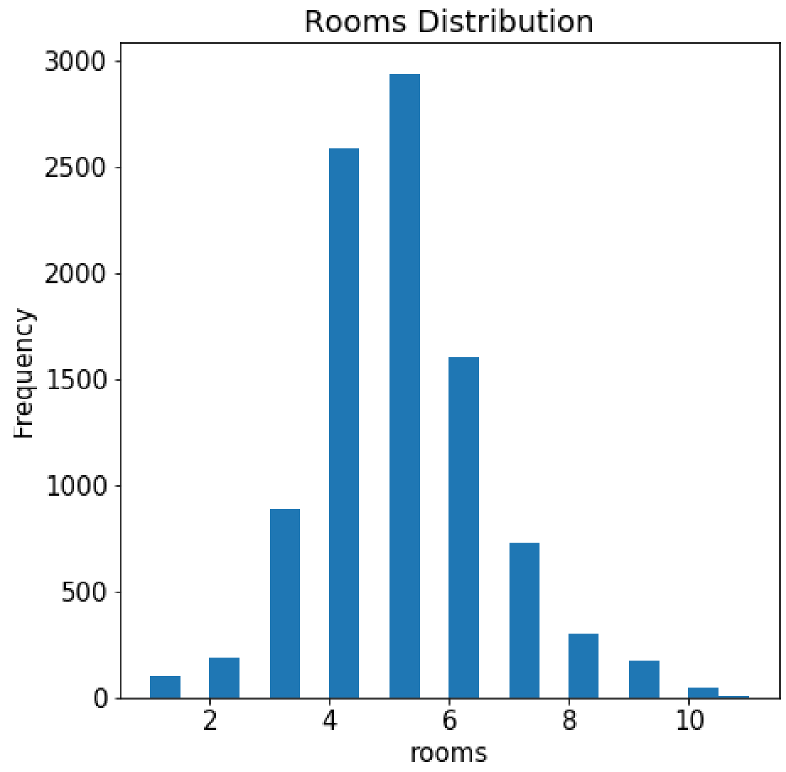
```
(array([  97.,  188.,  890., 2587., 2940.,
1607.,  732.,  298.,  168.,
         50.]), array([ 1.,  2.,  3.,  4.,  5.,
6.,  7.,  8.,  9., 10., 11.]), <a list of 10
Patch objects>)
```



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()
```



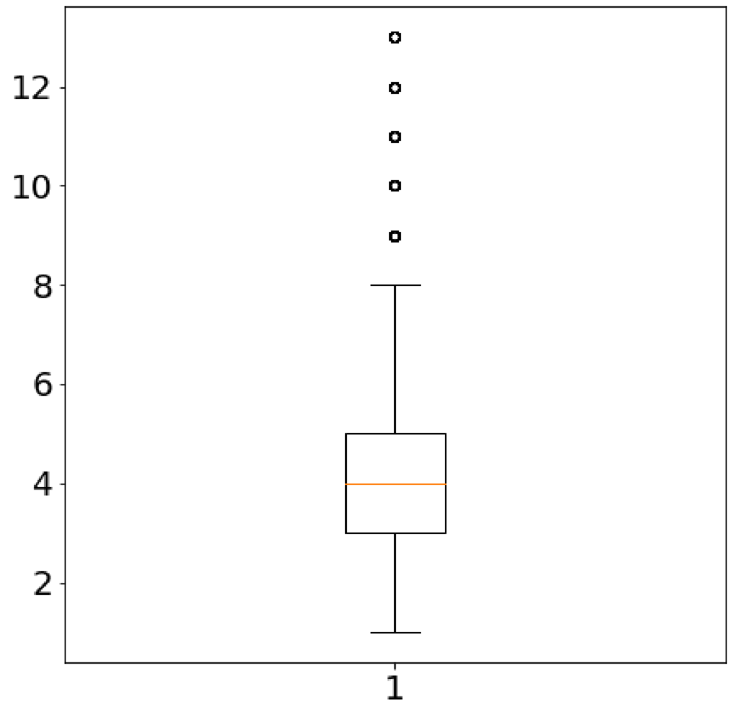
Univariate plots: boxplot

- 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'])
```

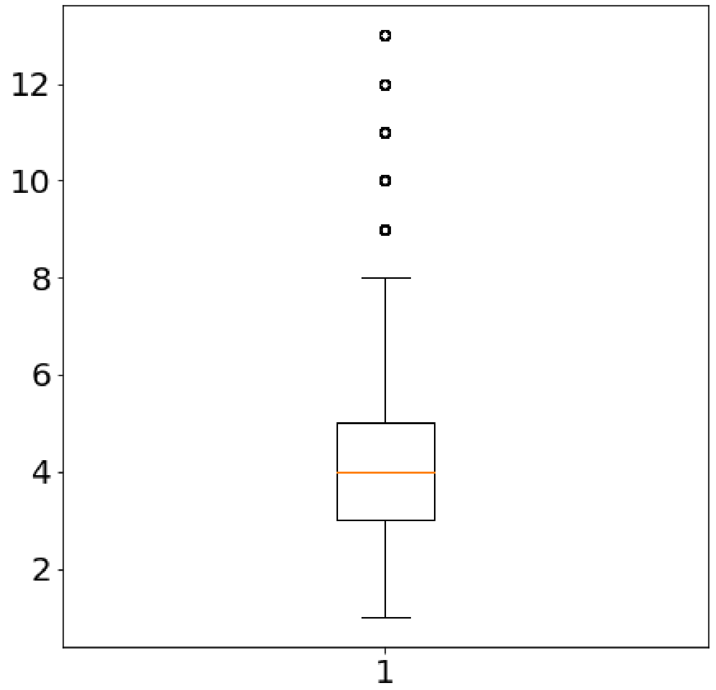
```
{'whiskers': [<matplotlib.lines.Line2D object at
0x12f4e0f98>, <matplotlib.lines.Line2D object at
0x12f4ed320>], 'caps': [<matplotlib.lines.Line2D
object at 0x12f4ed668>, <matplotlib.lines.Line2D
object at 0x12f4ed9b0>], 'boxes':
[<matplotlib.lines.Line2D object at
0x12f4e0da0>], 'medians':
[<matplotlib.lines.Line2D object at
0x12f4edcf8>], 'fliers':
[<matplotlib.lines.Line2D object at
0x12f4f8080>], '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 `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
- **What is the median number of rooms in this dataset? What number of rooms would be considered outliers?**



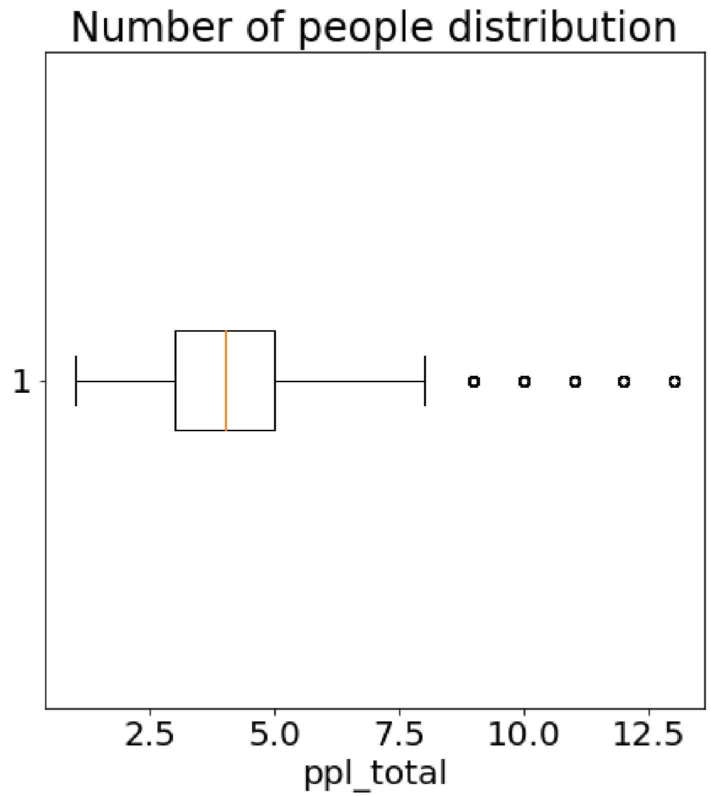
Univariate plots: boxplot (cont'd)

- You can change the 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
0x12f7375f8>, <matplotlib.lines.Line2D object at
0x12f737978>], 'caps': [<matplotlib.lines.Line2D
object at 0x12f737cc0>, <matplotlib.lines.Line2D
object at 0x13072f048>], 'boxes':
[<matplotlib.lines.Line2D object at
0x12f7374a8>], 'medians':
[<matplotlib.lines.Line2D object at
0x13072f390>], 'fliers':
[<matplotlib.lines.Line2D object at
0x13072f6d8>], '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 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
```

```
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

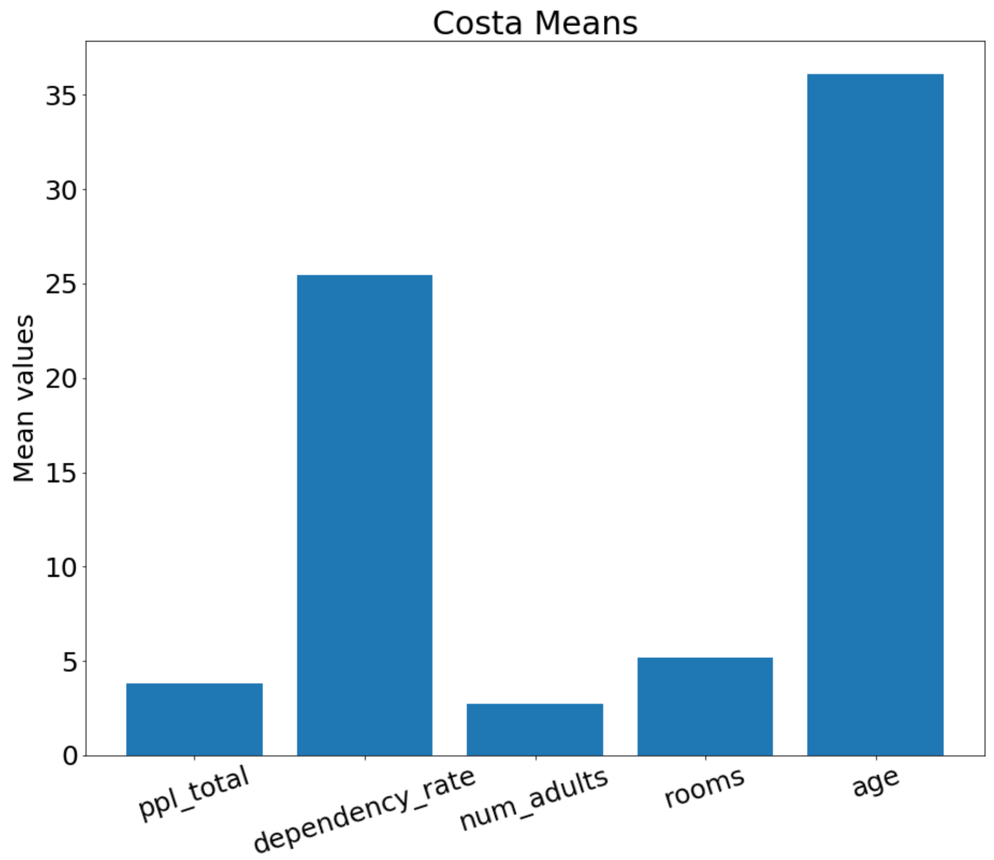
```
# Adjust figure size before plotting.
plt.figure(figsize = (15, 12))
plt.bar(bar_positions, bar_heights)
```

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

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

```
([<matplotlib.axis.XTick object at
0x130f5b860>, <matplotlib.axis.XTick object
at 0x130f5b198>, <matplotlib.axis.XTick
object at 0x130f53a90>,
<matplotlib.axis.XTick object at
0x12f7379b0>, <matplotlib.axis.XTick object
at 0x12ec33c50>], <a list of 5 Text
xticklabel objects>)
```

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



Results of univariate EDA

- We just spent some time looking at variables on their own, here are some questions you should ask after performing exploratory data analysis:

1. **Did any of the variables have uneven distributions?**
2. **Did any of the variables have a high proportion of outliers?**
3. **Did we learn anything unexpected from this EDA?**

- Asking these questions early will help us avoid erroneous models later!

Module completion checklist

Objective	Complete
Reshape data using pandas	✔
Define use cases of Exploratory Data Analysis (EDA)	✔
Create histograms, boxplots, and bar charts	✔
Create scatterplots	
Customize graphs	
Create compound visualizations	
Saving your plots and your data	
Best practices of data visualization	

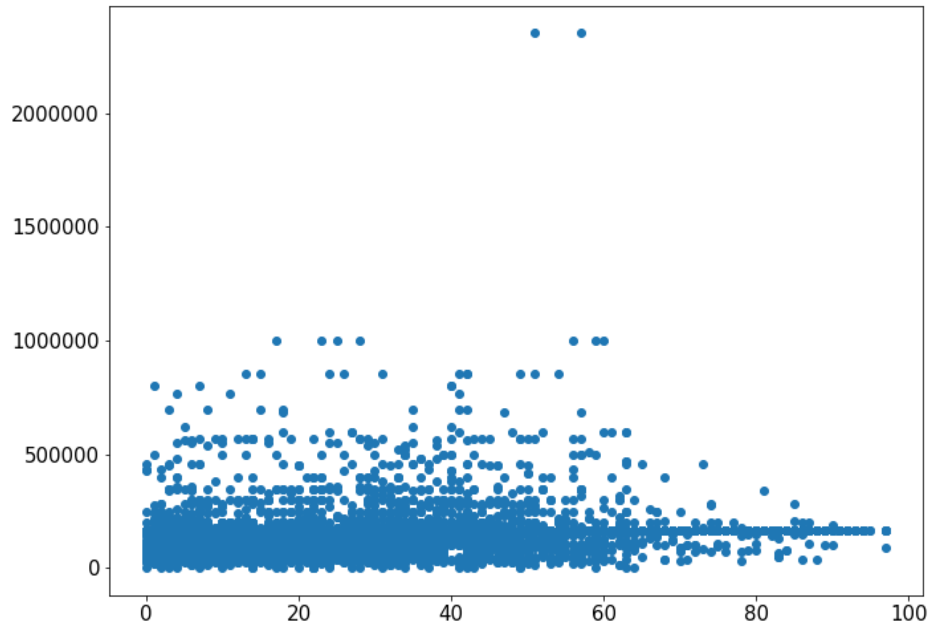
Exploring variables against each other

- Now that we've looked at variables on an individual level, let's explore how the variables interact with one another
- We're going to explore how `age` and `monthly_rent` interact with different types of visualizations

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

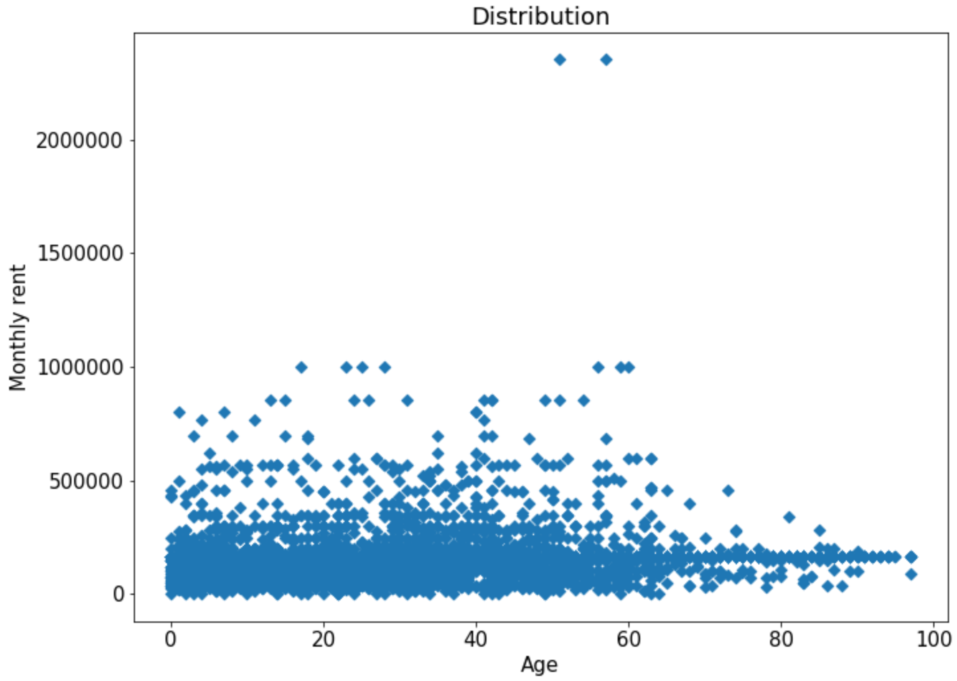
```
plt.scatter(costa_viz['age'],
            costa_viz['monthly_rent'])
plt.show()
```



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 2

Exercise 2

Module completion checklist

Objective	Complete
Reshape data using pandas	✔
Define use cases of Exploratory Data Analysis (EDA)	✔
Create histograms, boxplots, and bar charts	✔
Create scatterplots	✔
Customize graphs	
Create compound visualizations	
Saving your plots and your data	
Best practices of data visualization	

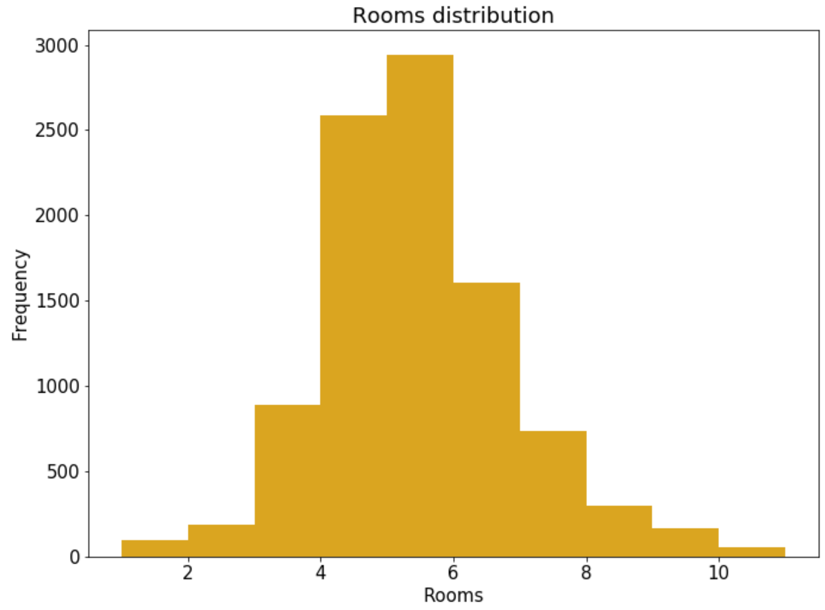
Customize colors

- 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

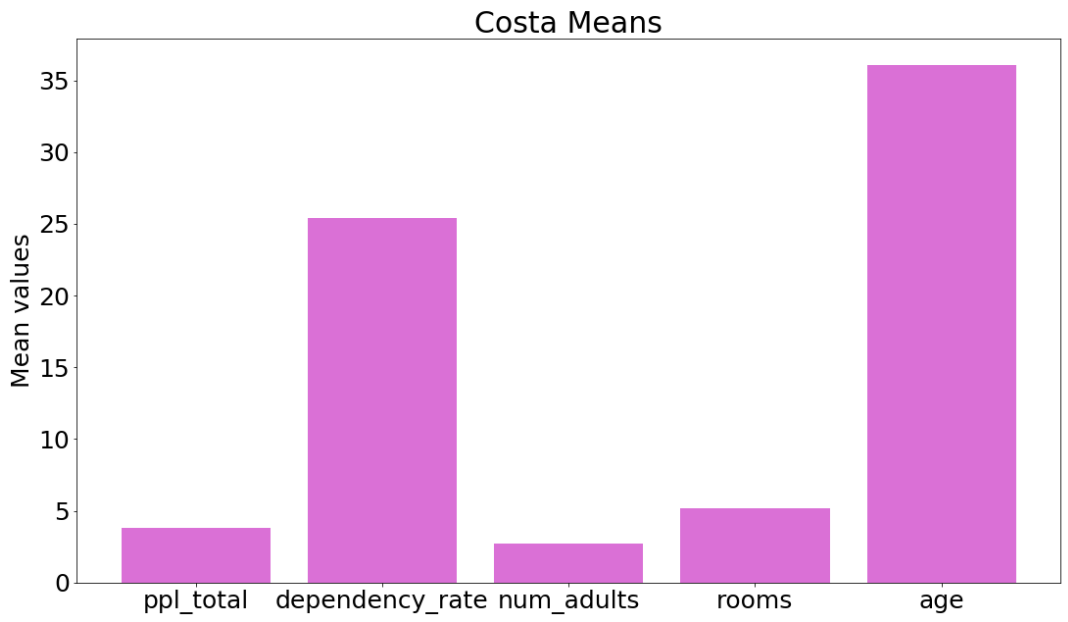
```
plt.figure(figsize = (18, 10))
plt.bar(bar_positions,
        bar_heights,
        color = "orchid")
```

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

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

```
([<matplotlib.axis.XTick object at
0x1332be470>, <matplotlib.axis.XTick
object at 0x135053400>,
<matplotlib.axis.XTick object at
0x1350530b8>, <matplotlib.axis.XTick
object at 0x13507fc50>,
<matplotlib.axis.XTick object at
0x135e20198>], <a list of 5 Text
xticklabel objects>)
```

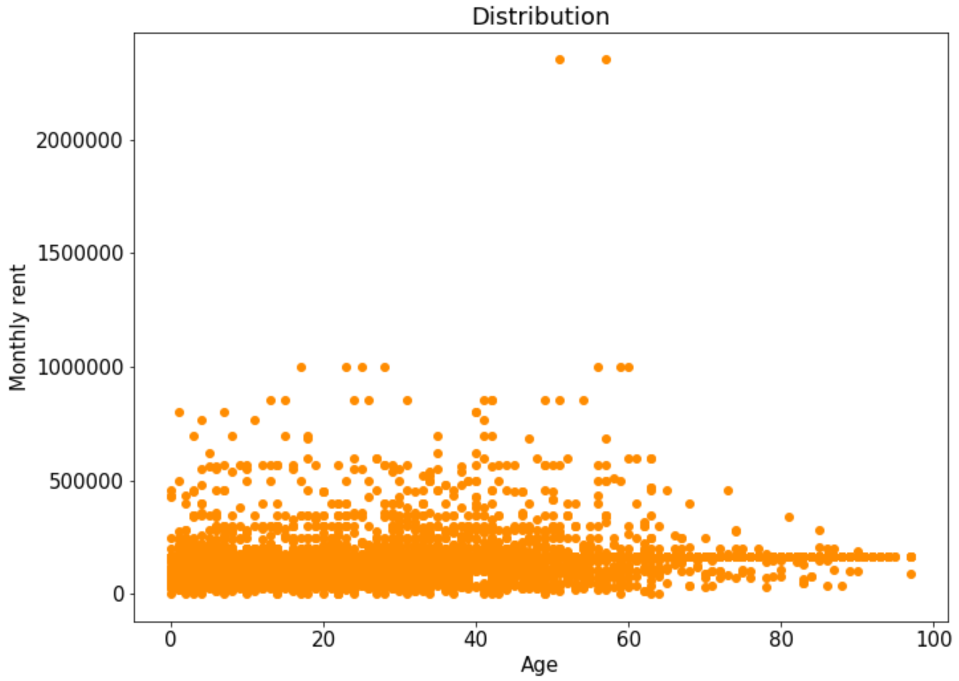
```
plt.ylabel('Mean values')
plt.title('Costa Means')
plt.show()
```



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())
```

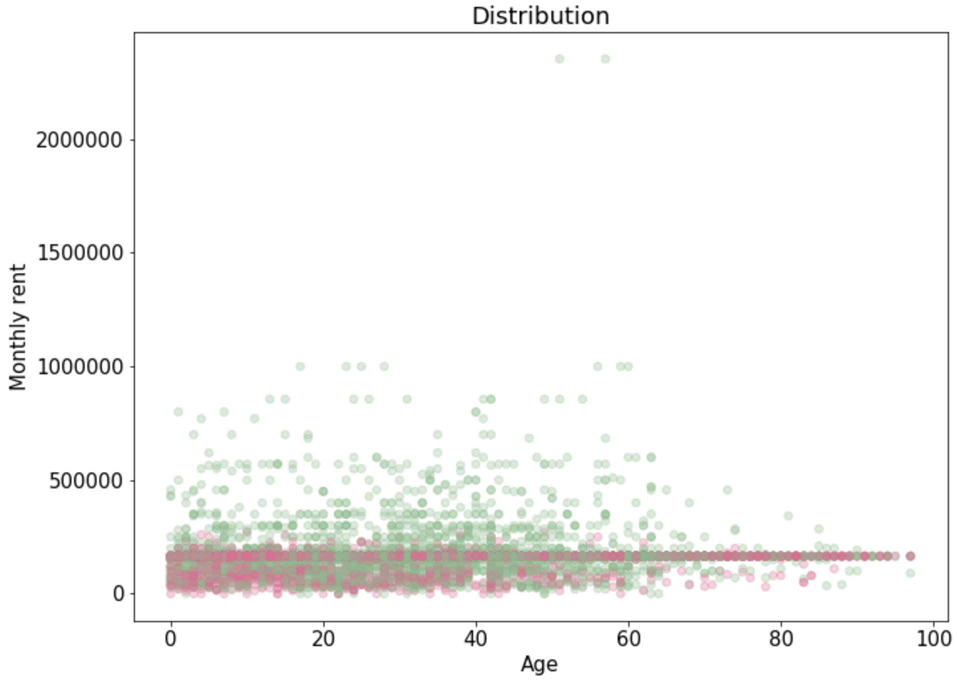
```
0    darkseagreen
1    darkseagreen
2    darkseagreen
3    darkseagreen
4    darkseagreen
Name: Target, dtype: object
```

```
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)
```

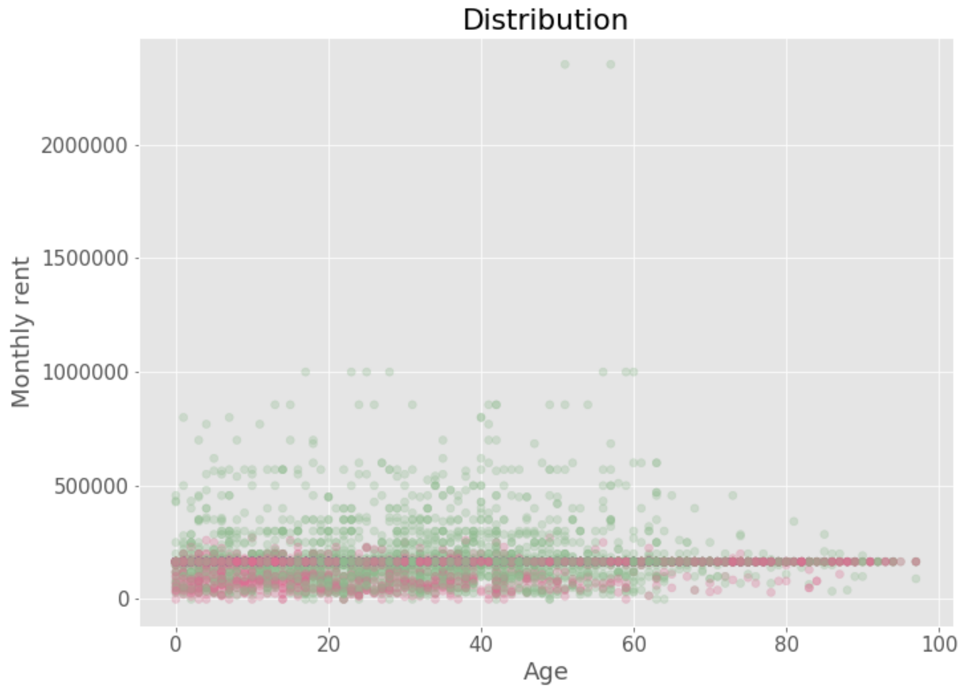
```
['seaborn-dark', 'seaborn-darkgrid', 'seaborn-ticks', 'fivethirtyeight', 'seaborn-whitegrid',
'classic', '_classic_test', 'fast', 'seaborn-talk', 'seaborn-dark-palette', 'seaborn-bright', 'seaborn-
pastel', 'grayscale', 'seaborn-notebook', 'ggplot', 'seaborn-colorblind', 'seaborn-muted', 'seaborn',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

- 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

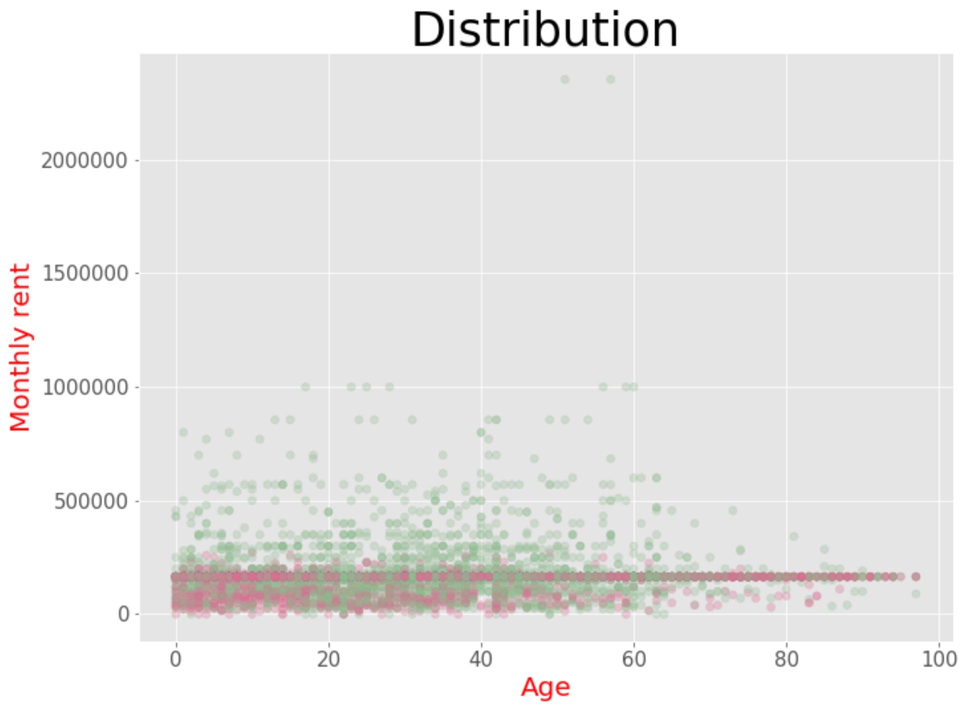
- As with all other plotting libraries, `matplotlib` comes with some pre-set defaults for all things you see in your plot
- To adjust any pre-set defaults, we will use `plt.rcParams` variable, which is a dictionary-like object
- You can either set those parameters on a one-off basis or you can create 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()
```

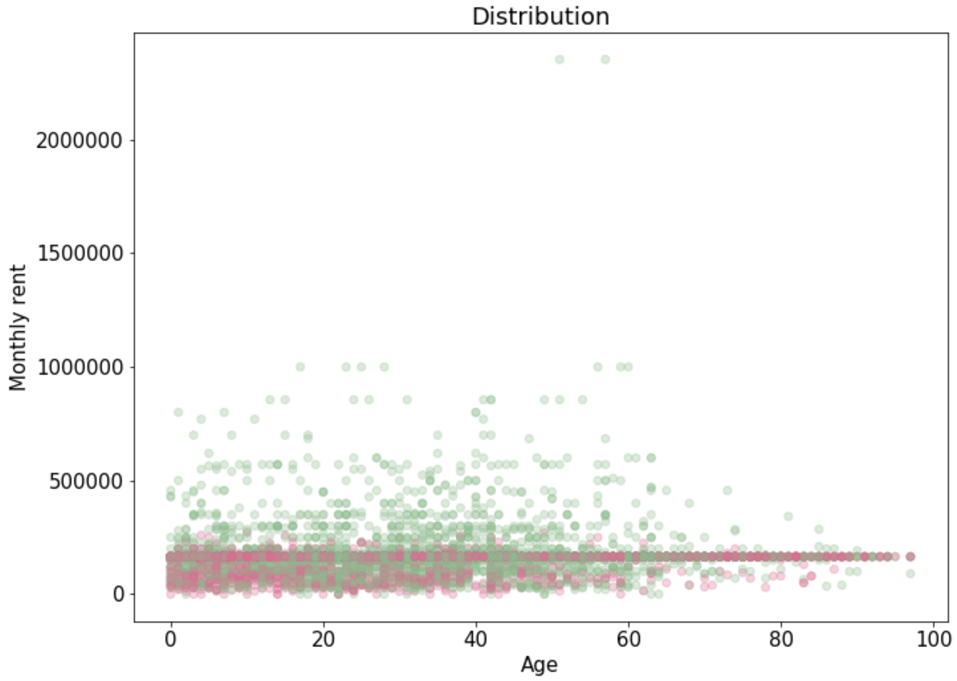


Customize plot settings: reset defaults

- 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 3

Exercise 3

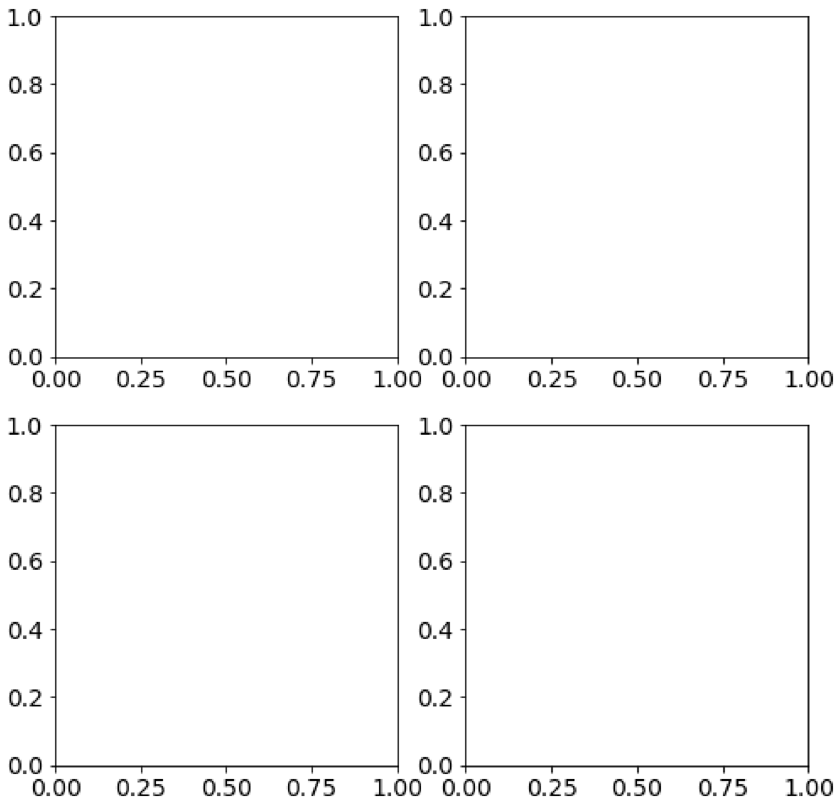
Module completion checklist

Objective	Complete
Reshape data using pandas	✔
Define use cases of Exploratory Data Analysis (EDA)	✔
Create histograms, boxplots, and bar charts	✔
Create scatterplots	✔
Customize graphs	✔
Create compound visualizations	
Saving your plots and your data	
Best practices of data visualization	

Compound visualizations: grids

- What if we want to display multiple plots on one sheet?
- 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

```
# 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 0x1388a97f0>
  <matplotlib.axes._subplots.AxesSubplot object at 0x115d99278>]
 [<matplotlib.axes._subplots.AxesSubplot object at 0x115dbf7f0>
  <matplotlib.axes._subplots.AxesSubplot object at 0x13c150d68>]]
```

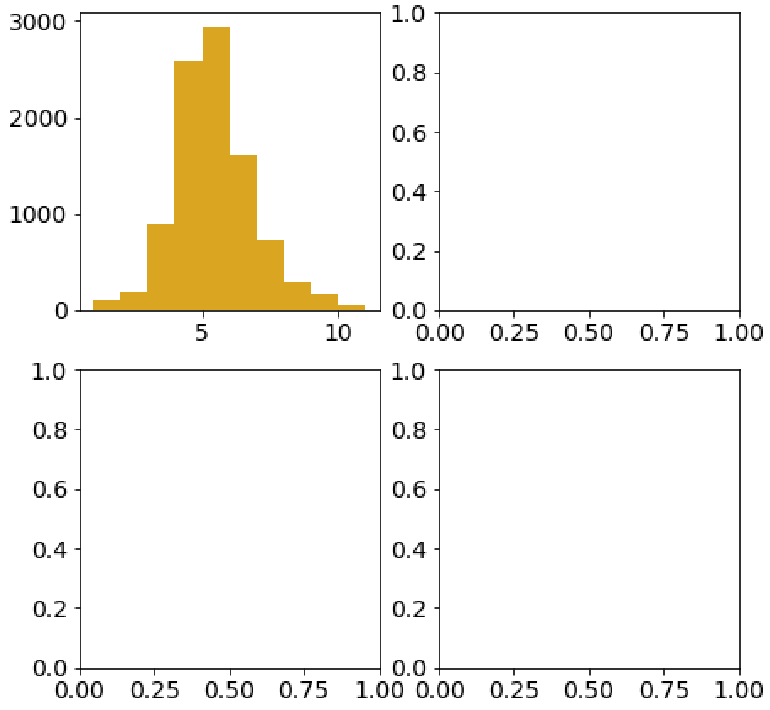
- Since it's a `2 x 2` grid, we have a 2D array with 4 entries that we will **“fill”** with values that are plots

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]`

```
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>)
```

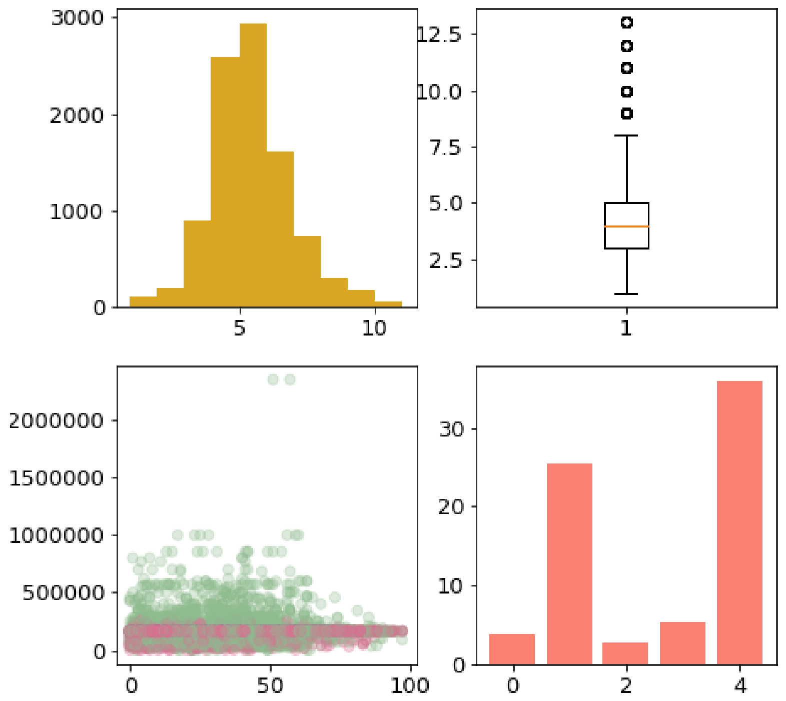


Compound visualizations: axes (cont'd)

- Let's fill out the remaining 3 plots

```
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 0x13e54eef0>, <matplotlib.axis.XTick object at 0x13e54ee80>,
<matplotlib.axis.XTick object at 0x13e57bac8>, <matplotlib.axis.XTick object at 0x12b7e1cf8>,
<matplotlib.axis.XTick object at 0x12b7e1128>]
```

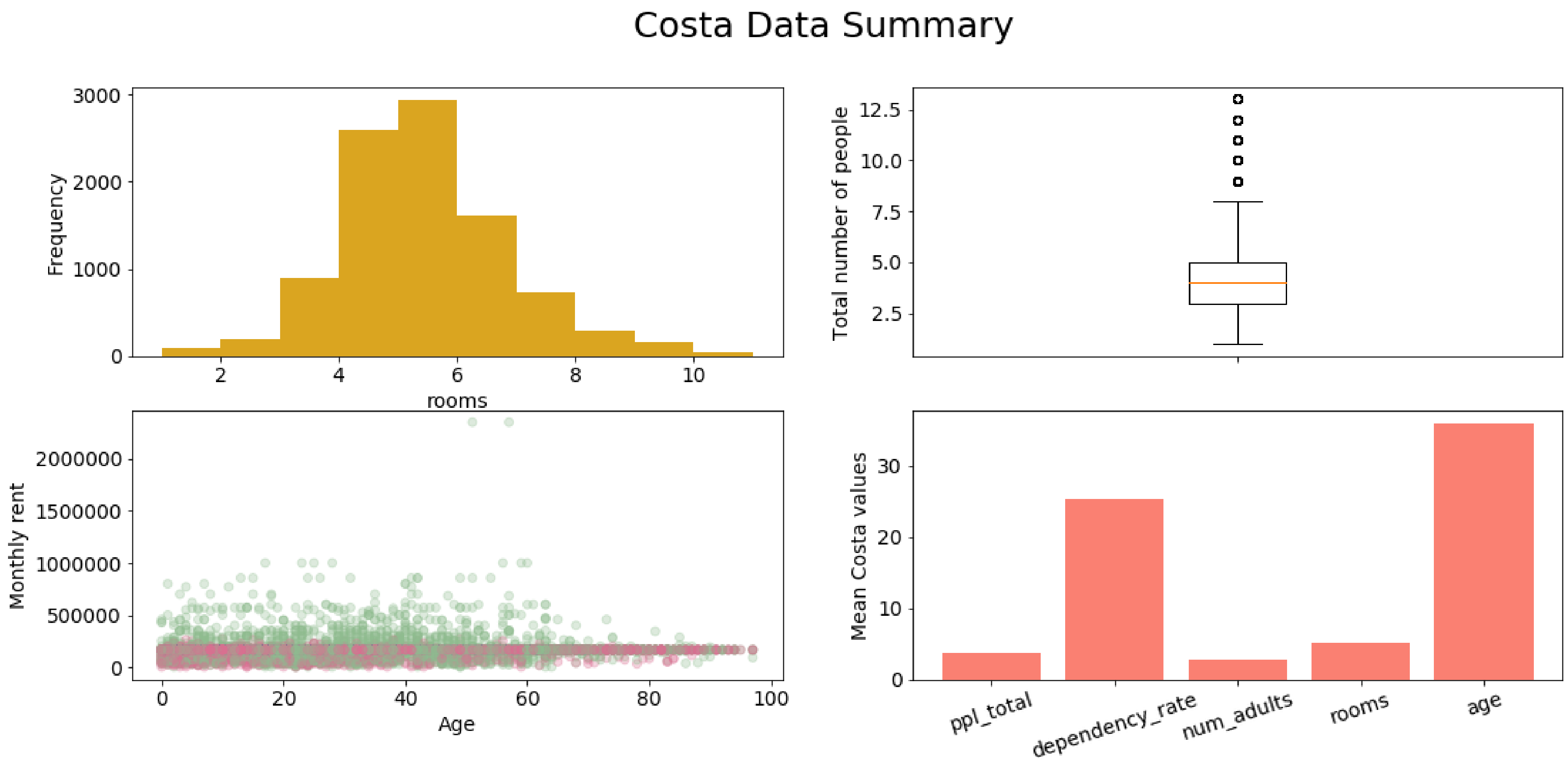
```
axes[1, 1].xaxis.set_ticklabels(bar_labels,rotation = 18)
```

```
[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()
```

Compound visualizations: figure adjustments



Compound visualizations: layered plots

- We can create figures containing multiple plots, layered on top of each other using the same plotting area `plt.subplots()`
- Layered plots allow any number of plotting layers, which makes them very flexible, especially for those datasets where looking at patterns across multiple categories is important!

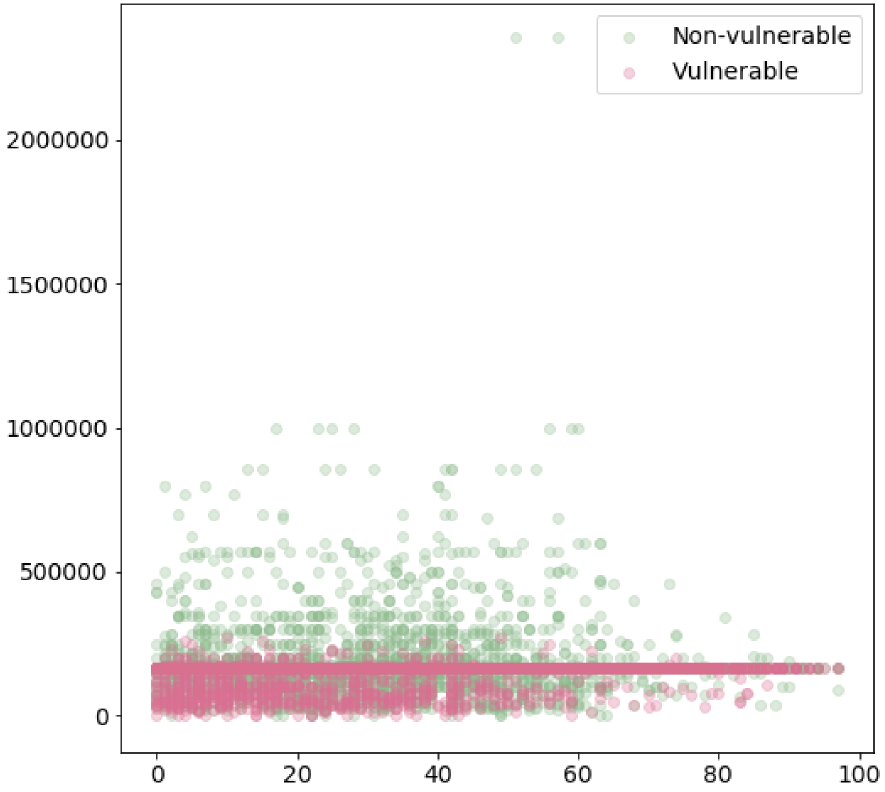
```
plt.clf()                    #<- clear plotting area
fig, axes = plt.subplots() #<- create a new figure and axes objects for plotting
```

Compound visualizations: layered plots

```
for key, value in color_dict.items():

    age = costa_viz.query('Target=='+str(key))['age']
    monthly_rent = costa_viz.query('Target=='+str(key))['monthly_rent']

    if key == 0:
        Flag = "Vulnerable"
    else:
        Flag = "Non-vulnerable"

    axes.scatter(age,
                 monthly_rent,
                 c = value,
                 label = Flag,
                 alpha = 0.3)
axes.legend()  #<- add a legend that would automatically get labels and colors from layers!
```

Compound visualizations: layered plots (cont'd)

```
plt.show()
```



Compound visualizations: layered plots (cont'd)

- Let's create a layered bar chart now

```
# We already have `Target` = `True` mean data.
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
```

```
# Let's get the `Target` = `False` mean data.
costa_false_means = costa_grouped_mean_long.query('Target == False')[['metric','mean']]
print(costa_false_means)
```

```
             metric       mean
0         ppl_total   4.358607
2   dependency_rate  26.011233
4        num_adults   2.388093
6             rooms   4.533839
8               age  31.314238
```

Compound visualizations: layered plots (cont'd)

```
# Mean values for `Target` = `False` data.
false_bar_heights = costa_false_means['mean']
# Mean values for `Target` = `True` data.
true_bar_heights = costa_true_means['mean']
# Labels of bars, their width, and positions are shared for both categories.
bar_labels = costa_false_means['metric']
num_bars = len(bar_labels)
bar_positions = np.arange(num_bars)
width = 0.35
```

```
# Clear the plotting area for the new plot.
plt.clf()
# Create the figure and axes objects.
fig, axes = plt.subplots()
```

Compound visualizations: layered plots (cont'd)

```
false_bar_chart = axes.bar(bar_positions,           #<- set `false` bar positions
                            false_bar_heights,     #<- set `false` bar heights
                            width,                 #<- set width of the bars
                            color = color_dict[0]) #<- set color to corresponding to `False` in 
dictionary
```

```
true_bar_chart = axes.bar(bar_positions + width, #<- set `true` bar positions
                          true_bar_heights,      #<- set `true` bar heights
                          width,                 #<- set width of the bars
                          color = color_dict[1]) #<- set color to corresponding to `True` in dictionary
```

Compound visualizations: layered plots (cont'd)

```
# Add text for labels, title and axes ticks.
axes.set_ylabel('Mean values')
axes.set_title('Costa metrics summary by Target')
axes.set_xticks(bar_positions + width/2)
```

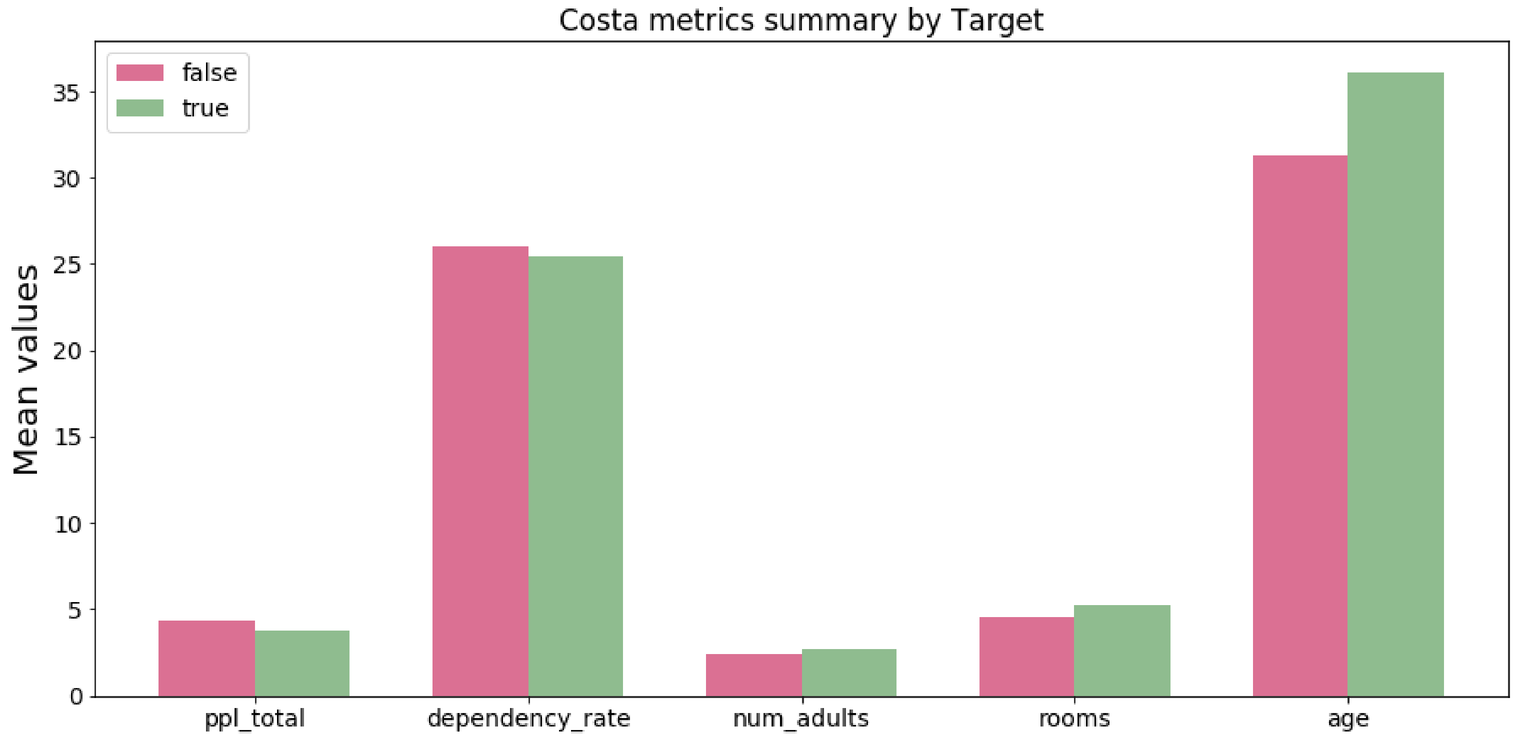
```
[<matplotlib.axis.XTick object at 0x140ca45c0>, <matplotlib.axis.XTick object at 0x13f69f518>,
<matplotlib.axis.XTick object at 0x140cb1b00>, <matplotlib.axis.XTick object at 0x140d3e828>,
<matplotlib.axis.XTick object at 0x140d3ed30>]
```

```
axes.set_xticklabels(bar_labels)
```

```
[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: layered plots (cont'd)

```
# Add a legend for each chart and corresponding labels.
axes.legend((false_bar_chart, true_bar_chart), ('false', 'true'))
# Adjust figure size.
fig.set_size_inches(15, 7)
plt.show()
```



Module completion checklist

Objective	Complete
Reshape data using pandas	✔
Define use cases of Exploratory Data Analysis (EDA)	✔
Create histograms, boxplots, and bar charts	✔
Create scatterplots	✔
Customize graphs	✔
Create compound visualizations	✔
Saving your plots and your data	
Best practices of data visualization	

Saving your plots

- You will be saving all of your graphs in the `plots` folder
- Save the current plot with `fig.savefig()`, where `fig` is any figure you want to save

```
fig.savefig(main_dir + '/plots/costa_metrics_by_target.png')
```

- Now open your plots folder and look at the file you have saved

Saving your data

- To save your data to a `csv` file, we will use a simple `df.to_csv()` function, where `df` is any dataframe
- When saving to a `csv` format, make sure to provide:
  - the path to your file with its name
  - the `index` argument (if it is set to `True`, the dataframe will be written with its index as the leftmost column)

```
costa_grouped_mean_long.to_csv(data_dir + '/costa_summary_by_target.csv',
                                   index = False)
```

- Now open your data folder and look at the file you have saved

Module completion checklist

Objective	Complete
Reshape data using pandas	✔
Define use cases of Exploratory Data Analysis (EDA)	✔
Create histograms, boxplots, and bar charts	✔
Create scatterplots	✔
Customize graphs	✔
Create compound visualizations	✔
Saving your plots and your data	✔
Best practices of data visualization	

Visualization best practices

Four pillars

- **Purpose** – Identify the stakeholders and their objectives
- **Content** – Pull out the content that matters most to the stakeholders
- **Structure** – Which chart best displays the content you want to display?
- **Formatting** – Are the titles and axes easily readable? Are the colors aesthetically pleasing?

Knowledge check 4

Exercise 4

Module completion checklist

Objective	Complete
Reshape data using pandas	✔
Define use cases of Exploratory Data Analysis (EDA)	✔
Create histograms, boxplots, and bar charts	✔
Create scatterplots	✔
Customize graphs	✔
Create compound visualizations	✔
Saving your plots and your data	✔
Best practices of data visualization	✔

Workshop: next steps!

- **Now we are at the workshop portion of the day**

Workshops are to be completed in the afternoon either with a dataset for a capstone project or with another dataset of your choosing. Make sure to annotate and comment your code so that it is easy for others to understand what you are doing. This is an exploratory exercise to get you comfortable with the content we discussed today

- Today, you can try out the concepts covered in each module
  - Transform your data for visualizations
  - Create simple plots to view trends and patterns and customize them
  - Visualize compound/grid plots and layered bar charts
  - Save the above summary plot as a `.png` file in the `plots` folder

This completes our module

Congratulations!