Visualisation Using Python

One of the main advantages of using Python for data exploration it has a wide selection of packages for plotting data and generating nice looking graphics with an easy to use interface.

Amongst these packages, the most popular for plotting is `matplotlib` , which is built to resemble the plot functions of MATLAB. `matplotlib` forms basis of many other popular Python plotting packages.

In this section, we will be mainly looking at two packages:

- `matplotlib`
- `seaborn`

The `seaborn` package is a more user friendly version of `matplotlib` , tailored for Data Science. Whereas `matplotlib` is designed for flexibility and general plots, `seaborn` is specifically designed for visualising data, with simple APIs for most statistical plot types.

matplotlib Package ¶

In [1]:

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

Figures and Axes

Reference: [https://matplotlib.org/faq/usage_faq.html#usage (https://matplotlib.org/faq/usage_faq.html#usage)](https://matplotlib.org/faq/usage_faq.html#usage)

In `matplotlib` language:

- a *Figure* is the canvas upon which one or more plots are to be shown.
- an *Axes* is one plot with x and y axes.

Both figures and axes are objects, and their properties define the apperance of a plot. For example, the figure defines the colour of the background, the title of the image etc. The axes defines the title of a particular plot, the x and y axes labels, the corresponding tick spacing and labels, legends etc.

Data is plotted within Axes. A figure may contain several axes.

If we do not explicitly define a figure and/or axes, then `matplotlib` will use a builtin default figure and axes.

Two API modes

Reference: https://matplotlib.org/tutorials/introductory/lifecycle.html (https://matplotlib.org/tutorials/introductory/lifecycle.html)

There are *two* parallel interfaces for `matplotlib` :

- Object Oriented API
- MATLAB-like state based API

The Object Oriented API follows the Python object model, in that to produce a plot, we need to

- Create a figure object
- Using a figure method to create one or more axes within the figure object
- Using the axes object method, for each axes, create one or more plots within.

The MATLAB-like API do not concern with objects, these are taken care of within the implementation. The user only need to call a few global functions defined within `matplotlib.pyplot` , any new plots being made are added to the same "state", i.e. a default figure and axes made for the session.

We will demonstrate the two modes below.

For this course, we will concentrate later on the MATLAB-like API, it is in general easier to use.

Simple Plots (Object-Oriented API)

Suppose we wish to visualise a function, say $\sin(x)$. We need to first generate a grid of x-values, from which we can compute the $y = \sin(x)$

In [2]:

```
import numpy as np
X = np.arange(0, 2*np.pi, 0.01)
Y = np.sin(X)
```

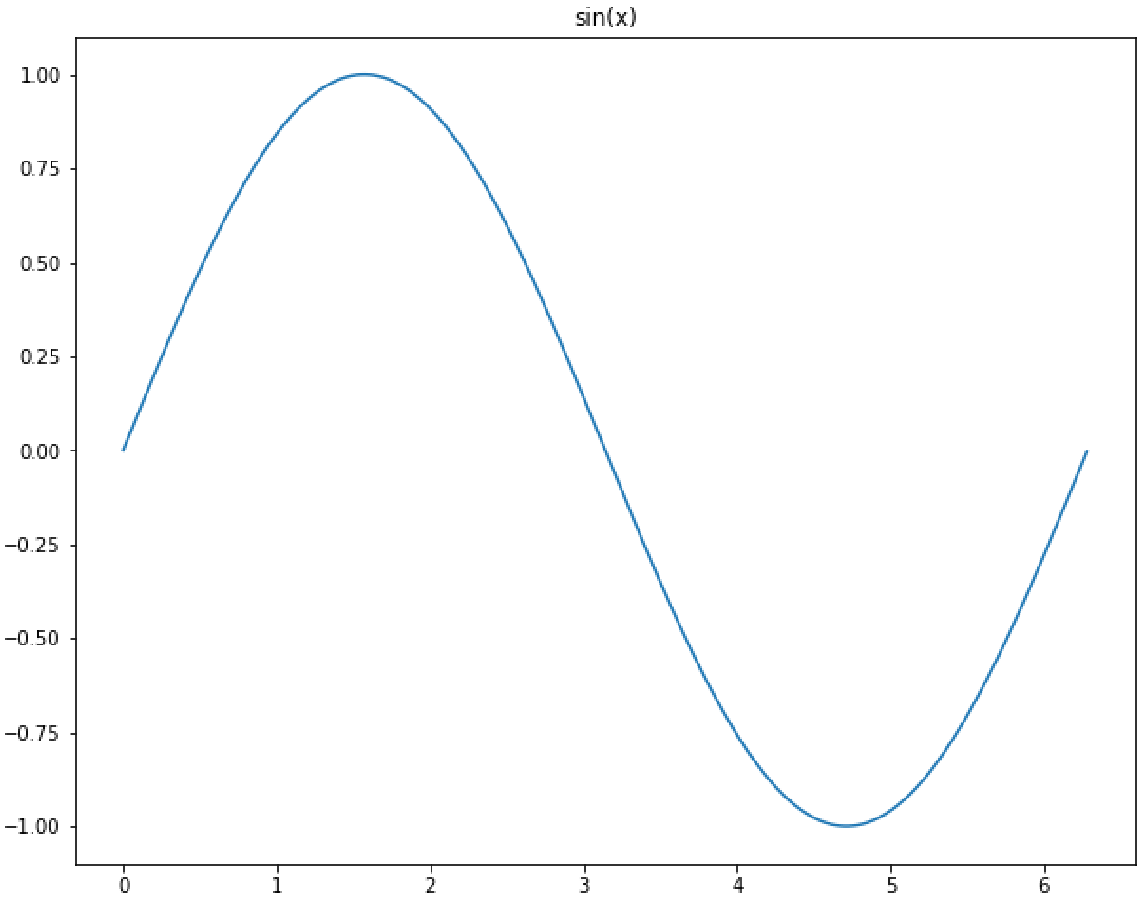
The function can then be visualised as a line plot of X against Y using the `.plot()` function:

In [3]:

```
fig = plt.figure(figsize=(10,8))
# three arguments, are n axes in row, n axes in col and
# index of this axis
ax = fig.add_subplot(1,1,1, title='sin(x)')
ax.plot(X, Y)
```

Out[3]:

```
[<matplotlib.lines.Line2D at 0x117d20410>]
```



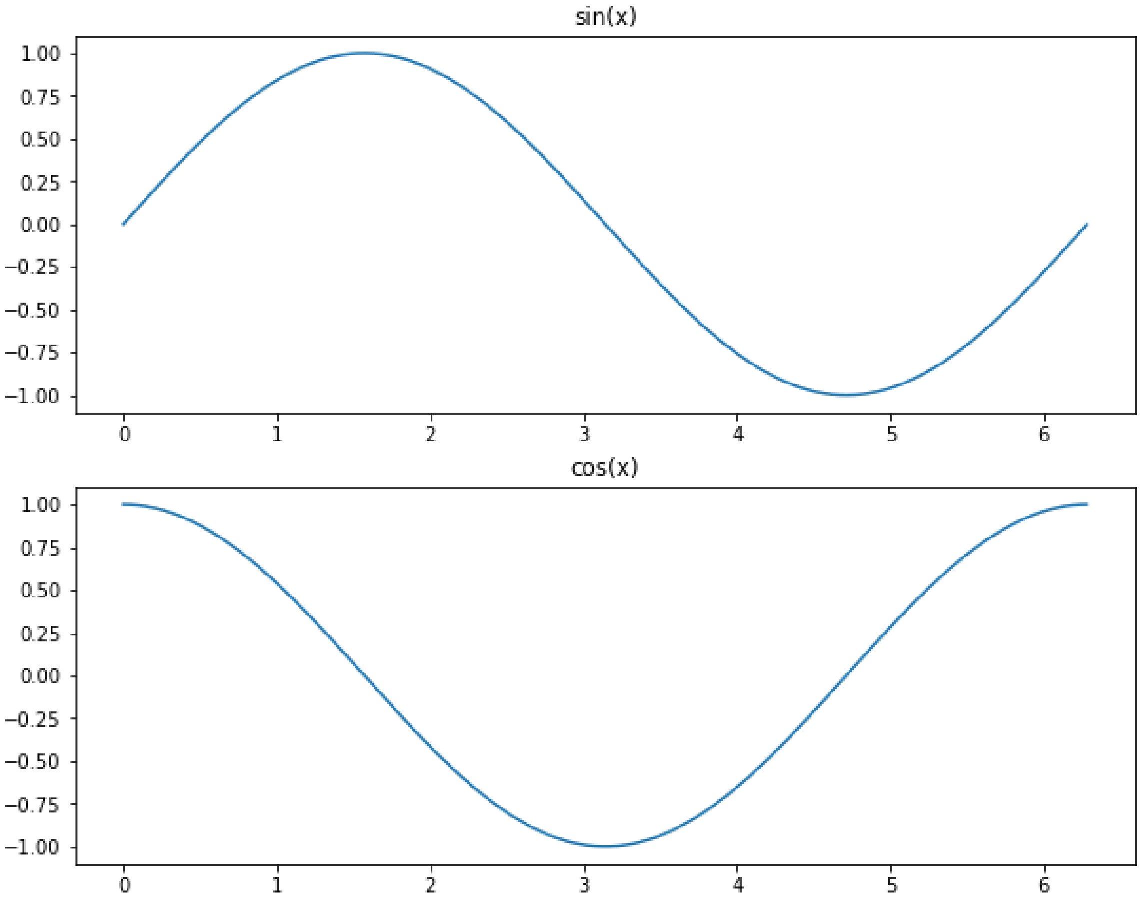
To gain a better idea of how the figure object works, we will plot a graph of $\cos(x)$ next to our graph of $\sin(x)$. To do this, we need to specify the dimensions that we want our figure to have.

In [4]:

```
fig = plt.figure(figsize=(10,8))    # using default size

ax1 = fig.add_subplot(2,1,1, title='sin(x)')
ax1.plot(X, Y)

ax2 = fig.add_subplot(2,1,2, title='cos(x)')
Y2 = np.cos(X)
ax2.plot(X, Y2)
```

Out[4]:

```
[<matplotlib.lines.Line2D at 0x11827e5d0>]
```



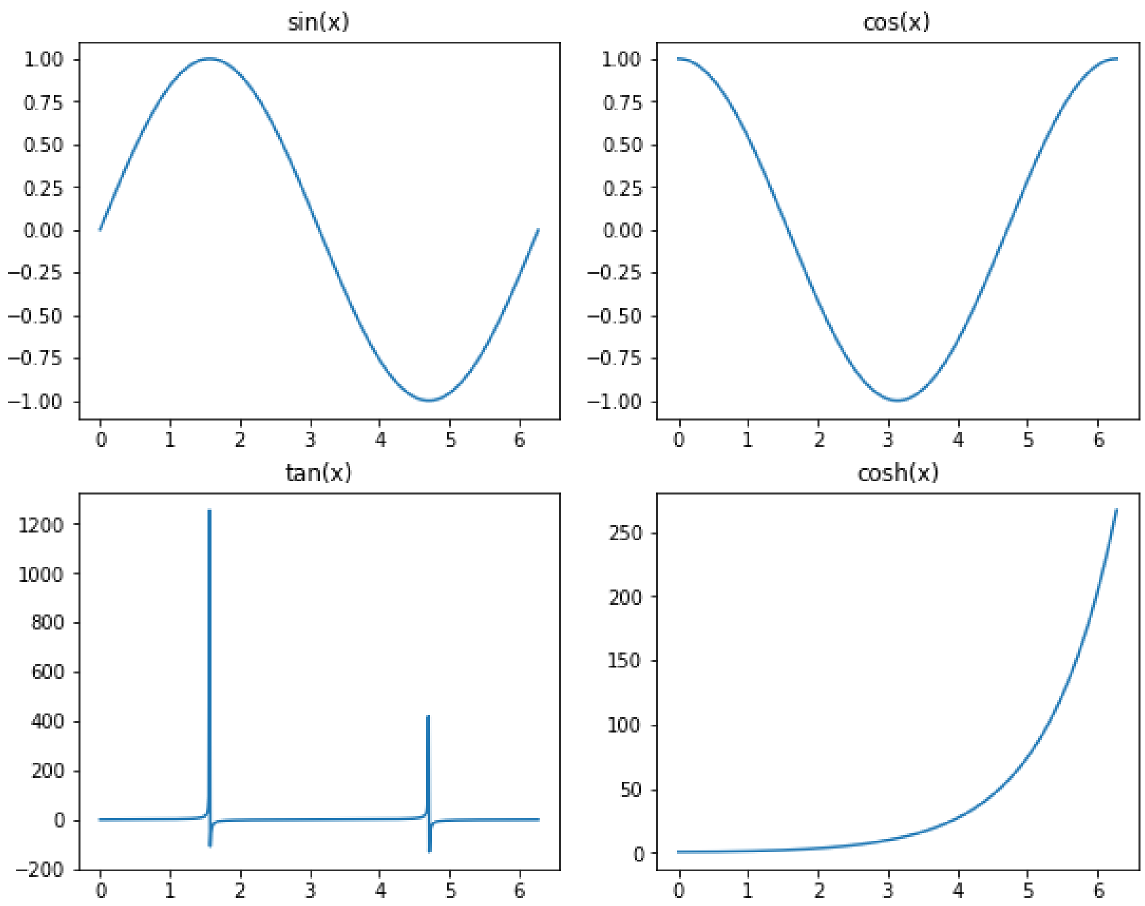
To further demonstrate, we plot a figure containing 4 sets of axes.

In [5]:

```python
fig = plt.figure(figsize=(10,8))    # using default size

ax1 = fig.add_subplot(2,2,1, title='sin(x)')
ax1.plot(X, Y)

ax2 = fig.add_subplot(2,2,2, title='cos(x)')
Y2 = np.cos(X)
ax2.plot(X, Y2)

ax3 = fig.add_subplot(2,2,3, title='tan(x)')
Y3 = np.tan(X)
ax3.plot(X, Y3)

ax4 = fig.add_subplot(2,2,4, title='cosh(x)')
Y4 = np.cosh(X)
ax4.plot(X, Y4)
```

Out[5]:

```
[<matplotlib.lines.Line2D at 0x11852d190>]
```



`plt.show()` in a Python script would then render all the graphics objects (referred to as *Artists* in `matplotlib` language) and produce the graphics.

However, as a side note, in Jupyter Notebook, `plt.show()` is redundant, as it is automatically assumed at the end of each code cell.

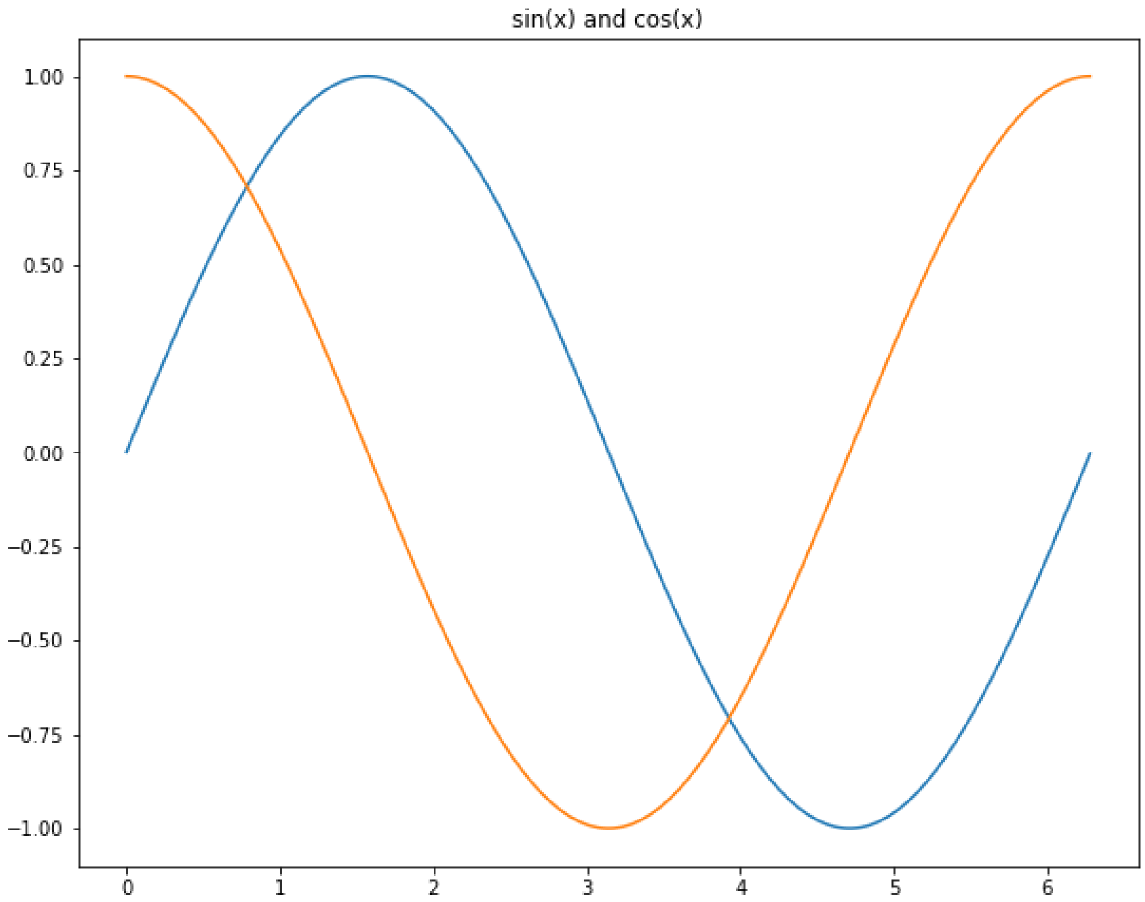
We can keep on adding plots to the same axes:

In [6]:

```
# It is a Jupyter notebook bug that for multiple plots
# in a same axes and figure, all matplotlib code
# must be in the same cell for the plot to show
# correctly

fig = plt.figure(figsize=(10,8))   # using default size
ax = fig.add_subplot(1,1,1, title='sin(x) and cos(x)')
ax.plot(X, Y)

Y2 = np.cos(X)
ax.plot(X, Y2)
```

Out[6]:

```
[<matplotlib.lines.Line2D at 0x11865f9d0>]
```

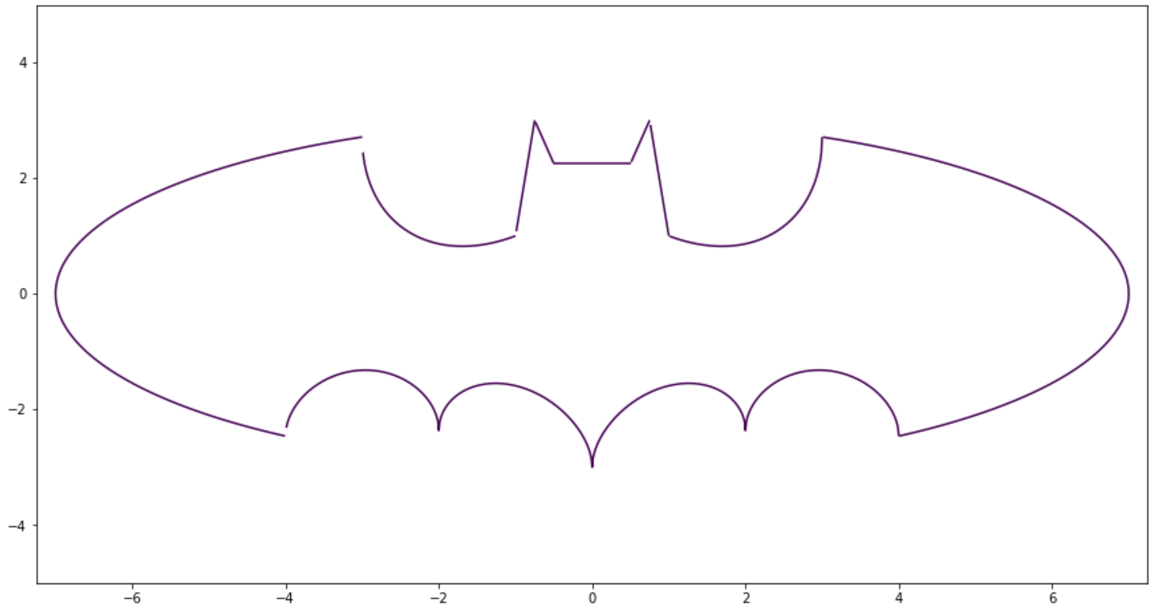


A slightly more interesting example, and certainly far more complicated one, is the plotting of the "Batman Curve" below.

In [7]:

```
from numpy import sqrt #originally had from scipy import sqrt
from numpy import meshgrid
from numpy import arange

xs = arange(-7.25, 7.25, 0.01)
ys = arange(-5, 5, 0.01)
x, y = meshgrid(xs, ys)

eq1 = ((x/7)**2*sqrt(abs(abs(x)-3)/(abs(x)-3))+(y/3)**2*sqrt(abs(y+3/7*sqrt(33))/(y+3/7*sqrt(33)))-1)
eq2 = (abs(x/2)-((3*sqrt(33)-7)/112)*x**2-3+sqrt(1-(abs(abs(x)-2)-1)**2)-y)
eq3 = (9*sqrt(abs((abs(x)-1)*(abs(x)-.75))/((1-abs(x))*(abs(x)-.75)))-8*abs(x)-y)
eq4 = (3*abs(x)+.75*sqrt(abs((abs(x)-.75)*(abs(x)-.5))/((.75-abs(x))*(abs(x)-.5)))-y)
eq5 = (2.25*sqrt(abs((x-.5)*(x+.5))/((.5-x)*(.5+x)))-y)
eq6 = (6*sqrt(10)/7+(1.5-.5*abs(x))*sqrt(abs(abs(x)-1)/(abs(x)-1))-(6*sqrt(10)/14)*sqrt(4-(abs(x)-1)**2)-y)

#eq1 = ((x/7.0)**2.0*sqrt(abs(abs(x)-3.0)/(abs(x)-3.0))+(y/3.0)**2.0*sqrt(abs(y+3.0/7.0*sqrt(33.0))/(y+3.0/7.0*sqrt(33.0)))-1.0)

plt.figure(figsize=(15,8))
for f in [eq1,eq2,eq3,eq4,eq5,eq6]:
    plt.contour(x, y, f, [0])

plt.show()
```

```
/anaconda3/envs/QA_PML/lib/python3.7/site-packages/ipykernel_launche
r.py:9: RuntimeWarning: invalid value encountered in sqrt
  if __name__ == '__main__':
/anaconda3/envs/QA_PML/lib/python3.7/site-packages/ipykernel_launche
r.py:10: RuntimeWarning: invalid value encountered in sqrt
  # Remove the CWD from sys.path while we load stuff.
/anaconda3/envs/QA_PML/lib/python3.7/site-packages/ipykernel_launche
r.py:11: RuntimeWarning: invalid value encountered in sqrt
  # This is added back by InteractiveShellApp.init_path()
/anaconda3/envs/QA_PML/lib/python3.7/site-packages/ipykernel_launche
r.py:12: RuntimeWarning: invalid value encountered in sqrt
  if sys.path[0] == '':
/anaconda3/envs/QA_PML/lib/python3.7/site-packages/ipykernel_launche
r.py:13: RuntimeWarning: invalid value encountered in sqrt
  del sys.path[0]
/anaconda3/envs/QA_PML/lib/python3.7/site-packages/ipykernel_launche
r.py:14: RuntimeWarning: invalid value encountered in sqrt
```



Below, we will be demonstrate the use of the OO API on a real data set, containing survey responses on the lifestyle/music prerferences of Slovakian youths.

First, we print the data set to get an idea of what it contains.

In [8]:

```
df_resp = pd.read_csv('./responses.csv')
for columns in df_resp.columns:
    print(columns)
```

Music
Slow songs or fast songs
Dance
Folk
Country
Classical music
Musical
Pop
Rock
Metal or Hardrock
Punk
Hiphop, Rap
Reggae, Ska
Swing, Jazz
Rock n roll
Alternative
Latino
Techno, Trance
Opera
Movies
Horror
Thriller
Comedy
Romantic
Sci-fi
War
Fantasy/Fairy tales
Animated
Documentary
Western
Action
History
Psychology
Politics
Mathematics
Physics
Internet
PC
Economy Management
Biology
Chemistry
Reading
Geography
Foreign languages
Medicine
Law
Cars
Art exhibitions
Religion
Countryside, outdoors
Dancing
Musical instruments
Writing
Passive sport
Active sport
Gardening
Celebrities
Shopping
Science and technology
Theatre
Fun with friends

Adrenaline sports
Pets
Flying
Storm
Darkness
Heights
Spiders
Snakes
Rats
Ageing
Dangerous dogs
Fear of public speaking
Smoking
Alcohol
Healthy eating
Daily events
Prioritising workload
Writing notes
Workaholism
Thinking ahead
Final judgement
Reliability
Keeping promises
Loss of interest
Friends versus money
Funniness
Fake
Criminal damage
Decision making
Elections
Self-criticism
Judgment calls
Hypochondria
Empathy
Eating to survive
Giving
Compassion to animals
Borrowed stuff
Loneliness
Cheating in school
Health
Changing the past
God
Dreams
Charity
Number of friends
Punctuality
Lying
Waiting
New environment
Mood swings
Appearence and gestures
Socializing
Achievements
Responding to a serious letter
Children
Assertiveness
Getting angry
Knowing the right people
Public speaking
Unpopularity

```
Life struggles
Happiness in life
Energy levels
Small - big dogs
Personality
Finding lost valuables
Getting up
Interests or hobbies
Parents' advice
Questionnaires or polls
Internet usage
Finances
Shopping centres
Branded clothing
Entertainment spending
Spending on looks
Spending on gadgets
Spending on healthy eating
Age
Height
Weight
Number of siblings
Gender
Left - right handed
Education
Only child
Village - town
House - block of flats
```

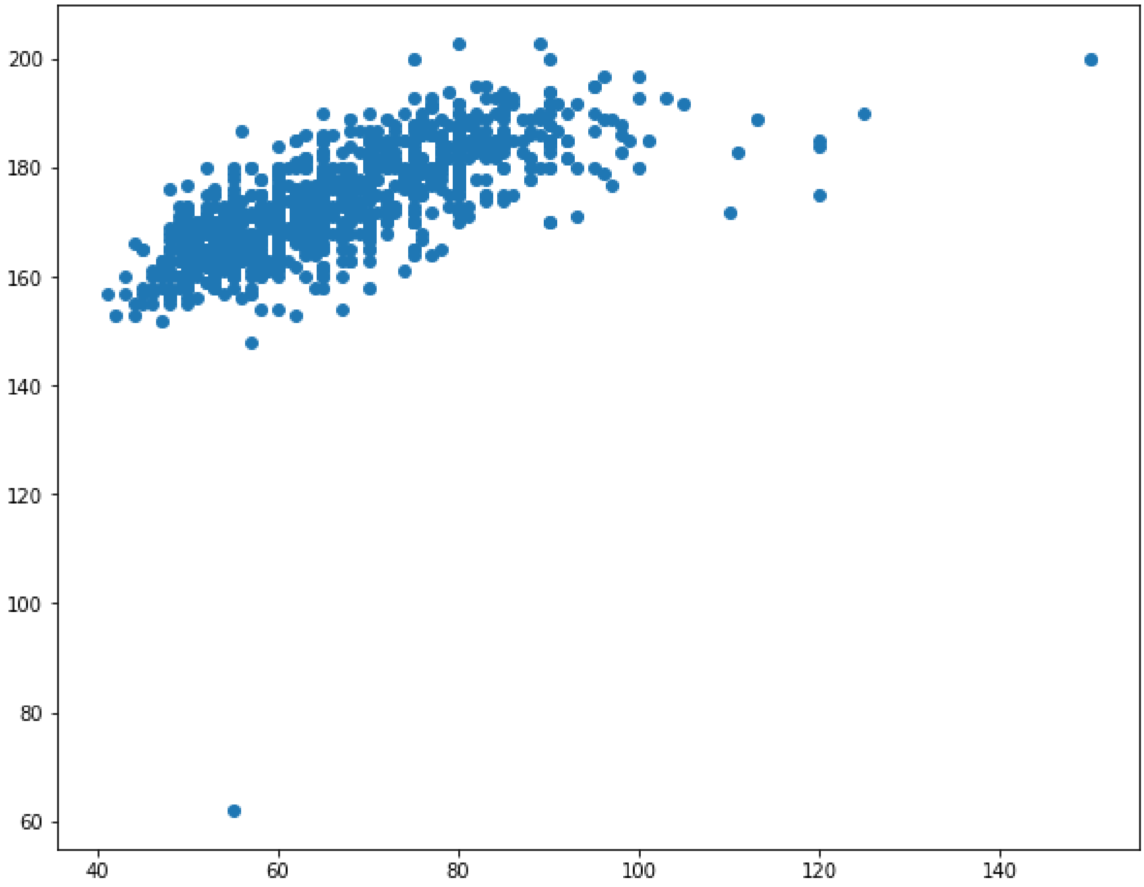
Initially, we will simply plot the height of each participant agains their weight.

In [9]:

```
x = df_resp['Weight']
y = df_resp['Height']

fig = plt.figure(figsize=(10,8))
ax = fig.add_subplot(1,1,1)

ax.scatter(x, y, marker = 'o')

ax.plot()
```

Out[9]:

```
[]
```



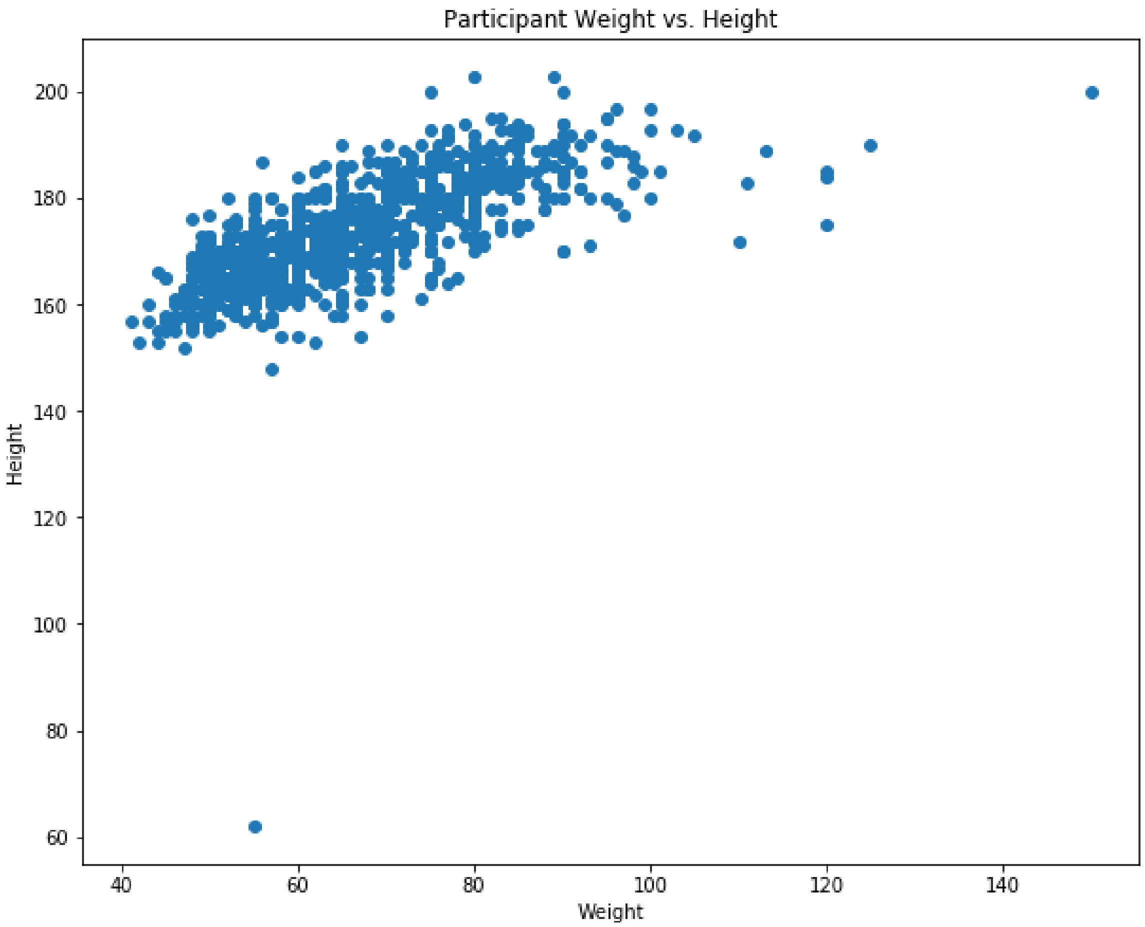
To make the graphic easier to read, we add X and Y axis lables, as well as adding a title to the graph.

In [10]:

```
x = df_resp['Weight']
y = df_resp['Height']

fig = plt.figure(figsize=(10,8))
ax = fig.add_subplot(1,1,1, title = 'Participant Weight vs. Height')

ax.scatter(x, y, marker = 'o')

#plt.title('Height and Weight')
ax.set_xlabel('Weight')
ax.set_ylabel('Height')

ax.plot()
```

Out[10]:

```
[]
```



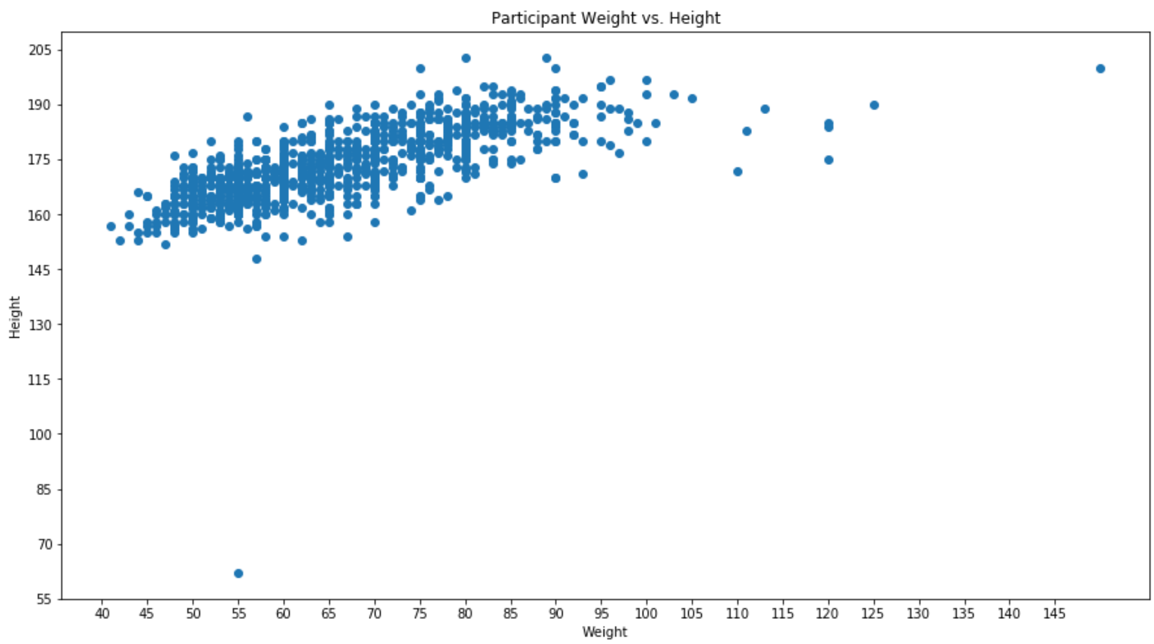
In order to make the graph easier to understand, we add our own scaled axes and increase the overall size of the graph.

In [11]:

```
x = df_resp['Weight']
y = df_resp['Height']

fig = plt.figure(figsize = (15,8))
ax = fig.add_subplot(1,1,1, title = 'Participant Weight vs. Height')

ax.scatter(x, y, marker = 'o')

#plt.title('Height and Weight')
ax.set_xlabel('Weight')
ax.set_ylabel('Height')

ax.set_xticks(np.arange(20, 150, 5))
ax.set_yticks(np.arange(55, 220+1, 15))

ax.plot()
```

Out[11]:

```
[]
```



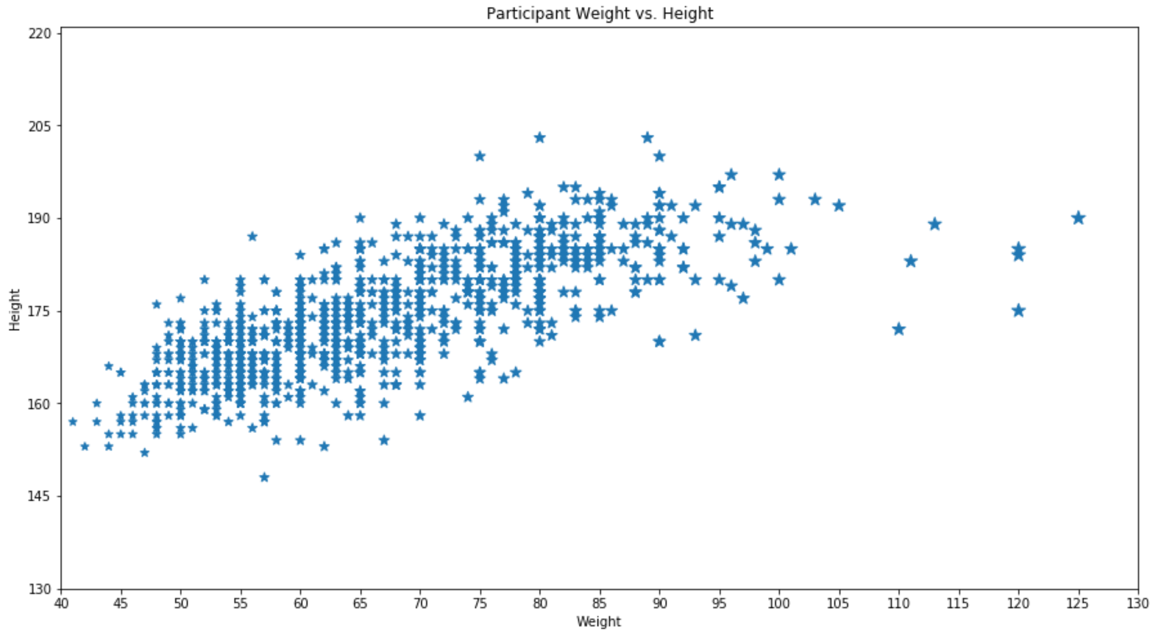
If we wished to show only the bulk of the data, we could rescale the axes. If we wanted to add a bit of pizzazz, we could represent each data point with a star.

In [12]:

```
x = df_resp['Weight']
y = df_resp['Height']

fig = plt.figure(figsize = (15,8))
ax = fig.add_subplot(1,1,1, title = 'Participant Weight vs. Height')

ax.scatter(x, y, marker = '*', s = df_resp['Weight'])
# Swe can immediately filter by colour as Gender is a string not an int 0 or 1


#plt.title('Height and Weight')
ax.set_xlabel('Weight')
ax.set_ylabel('Height')

ax.set_xticks(np.arange(20, 150+1, 5)) #+1 so we can see the whole plot
ax.set_yticks(np.arange(55, 220+1, 15))

ax.set_xlim(40,130, 5)
ax.set_ylim(130, 220+1, 15)

ax.plot()
```

Out[12]:

```
[]
```



In order to glean more information from the graphic, we are going to colour the chart by gender. To do this, we first need to enumerate the gender column, replacing male with 0 and female with 1.

After performing this operation, we can pass the gender column as the `c` argument (short for colour), producing a colour coded graphic.

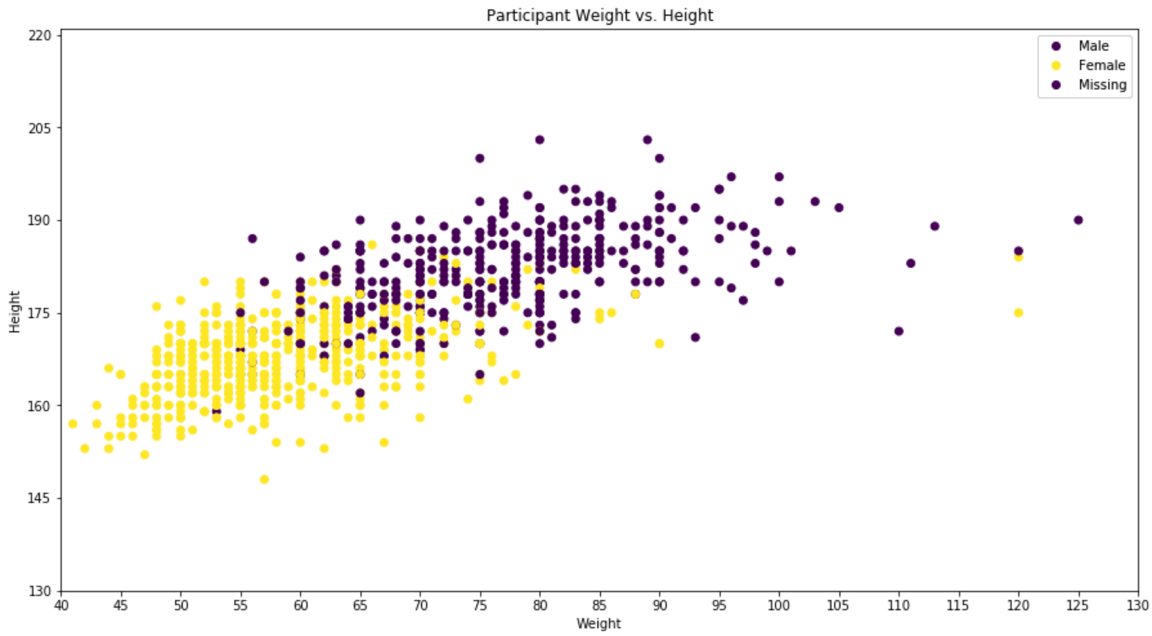
In [13]:

```
x = df_resp['Weight']
y = df_resp['Height']

fig, ax = plt.subplots(figsize = (15,8))
#ax = fig.add_subplot(1,1,1, )

scatter = ax.scatter(x, y, marker = 'o', c=df_resp['Gender'].replace(['male', 'f
emale'], [0,1]))
# We cannot immediately filter by colour as Gender is a string not an int 0 or 1

legend1 = ax.legend(*(scatter.legend_elements()[0],['Male', 'Female', 'Missing'
]))

ax.add_artist(legend1)

ax.set_title('Participant Weight vs. Height')

#plt.title('Height and Weight')
ax.set_xlabel('Weight')
ax.set_ylabel('Height')

ax.set_xticks(np.arange(20, 150+1, 5)) #+1 so we can see the whole plot
ax.set_yticks(np.arange(55, 220+1, 15))

ax.set_xlim(40,130, 5)
ax.set_ylim(130, 220+1, 15)

#scatter.
ax.plot()
```

```
/anaconda3/envs/QA_PML/lib/python3.7/site-packages/matplotlib/color
s.py:885: UserWarning: Warning: converting a masked element to nan.
  dtype = np.min_scalar_type(value)
/anaconda3/envs/QA_PML/lib/python3.7/site-packages/numpy/ma/core.py:
713: UserWarning: Warning: converting a masked element to nan.
  data = np.array(a, copy=False, subok=subok)
/anaconda3/envs/QA_PML/lib/python3.7/site-packages/matplotlib/ticke
r.py:589: UserWarning: Warning: converting a masked element to nan.
  s = self.format % xp
```

Out[13]:

```
[]
```



Finally, if we wish to save the graphic we have created, we use the method `.savefig(<filename>)`

In [14]:

```
fig.savefig('WeightVsHeight_c_Gender.png')
```

Exercise

*** Write a function which given x, will compute x^2**

*** Generate a Numpy array which goes from -2 to 2 in steps of 0.1**

*** Plot y = x^2 using the Matplotlib OO API and add an appropriate title.**

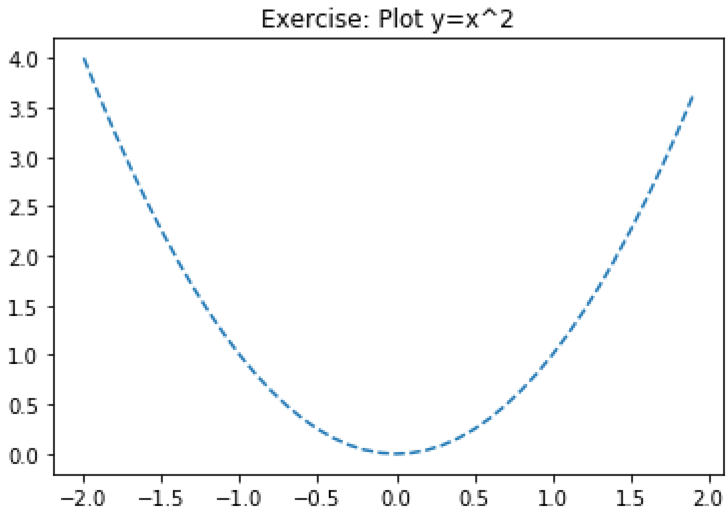
Solution

In [15]:

```
def y(x):
    return x**2

x = np.arange(-2, 2, 0.1)
y = y(x)

df = pd.DataFrame(np.array([x,y]).T, columns=['x','y'])

fig = plt.figure()
ax = fig.add_subplot(1,1,1, title = 'Exercise: Plot y=x^2')
ax.plot(x,y,linestyle = "--")
```

Out[15]:

```
[<matplotlib.lines.Line2D at 0x11e88a990>]
```

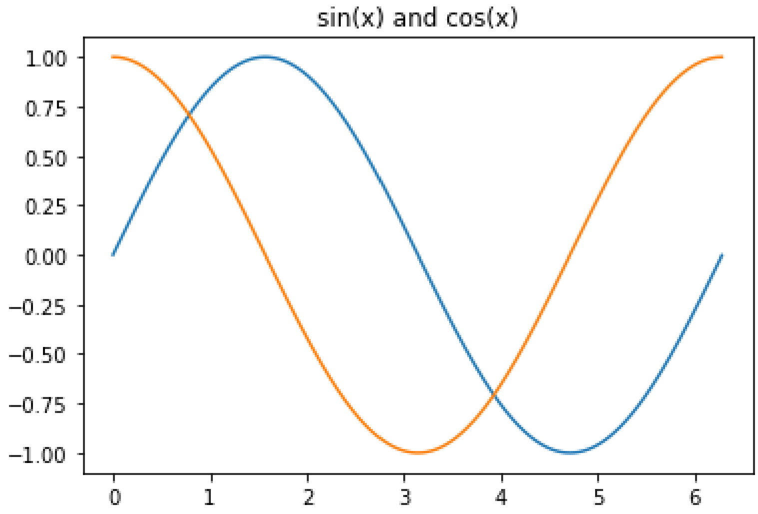


Simple Plots (MATLAB API)

The MATLAB API is much easier to use, but offers less flexibility than the OO API.

In [16]:

```
plt.plot(X, Y)
plt.plot(X, Y2)
plt.title('sin(x) and cos(x)')
plt.show()
```



Behind the scenes, the module still works in Object-Oriented mode, the call `plt.plot()` will by default create a new figure of a default size and one axes that fills up the entire figure.

This can be configured through various `pyplot` functions.

Writing to files

We can save files using both apis by using the `savefig()` method, specifying the filetype.

Object-Oriented API

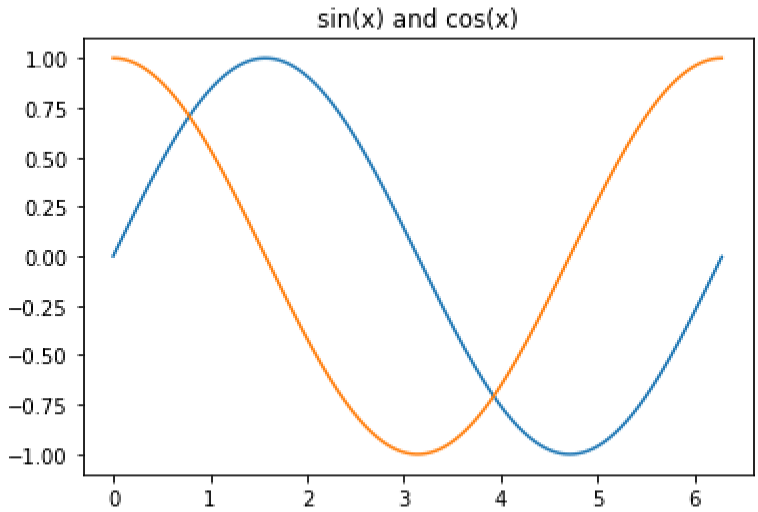
In [17]:

```
fig.savefig('example_graphics.png')
```

MATLAB-like API

In [18]:

```
# again Jupyter notebook requires all plots to
# be performed within one cell
plt.plot(X, Y)
plt.plot(X, Y2)
plt.title('sin(x) and cos(x)')

plt.savefig('example_graphics2.png')
```



Changing Appearances

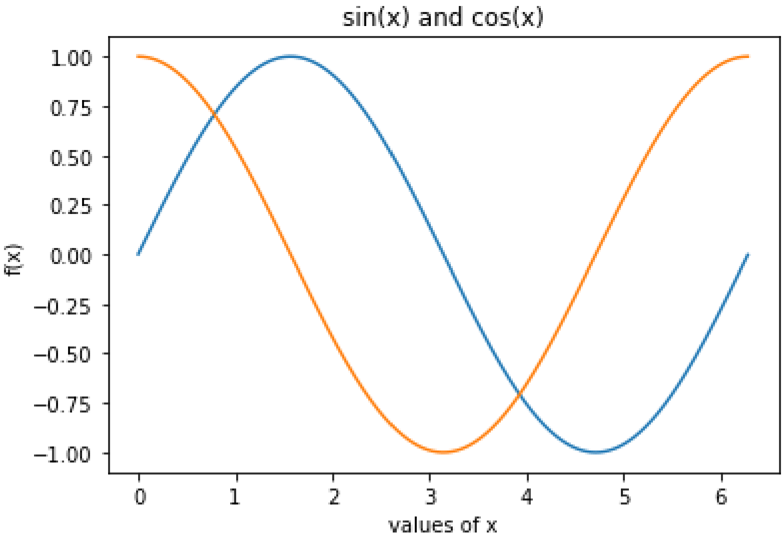
From this pont on we are going to be using the MATLAB like api. Below, we demonstrate the ways in which we can alter/remove/add various attributes of plots.

Axis Labels

To alter the labels on each axis, we use the `ylabel()` and `xlabel()` commands

In [19]:

```
plt.plot(X, Y)
plt.plot(X, Y2)
plt.title('sin(x) and cos(x)')
# setting axis labels
plt.xlabel('values of x')
plt.ylabel('f(x)')

plt.show()
```

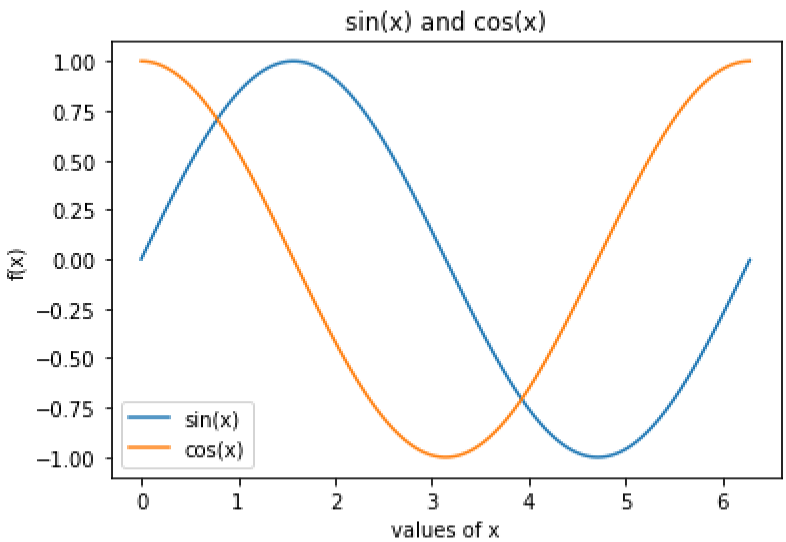


Adding Legends

If we wish to add a legend to the plot, we simply call the `legend()` command, which will auto-generate one using the axis labels the plot currently has.

In [20]:

```
plt.plot(X, Y, label='sin(x)')
plt.plot(X, Y2, label='cos(x)')
plt.title('sin(x) and cos(x)')
# setting axis labels
plt.xlabel('values of x')
plt.ylabel('f(x)')

plt.legend()

plt.show()
```

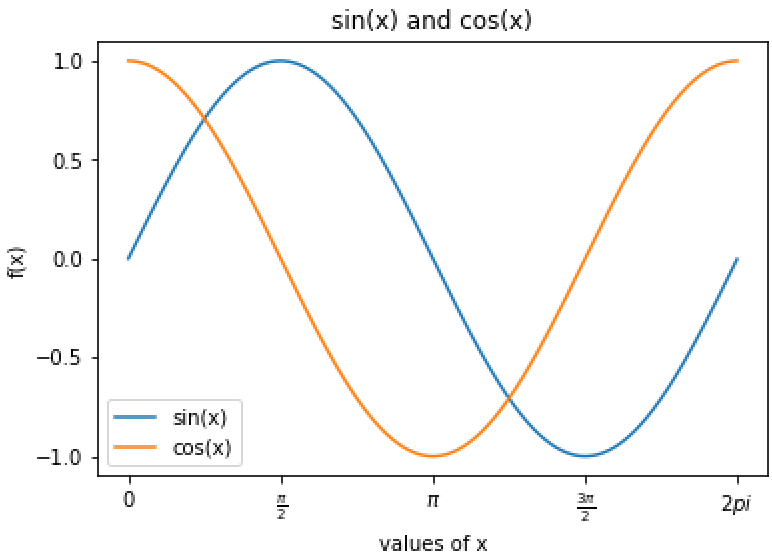


Axis ticks

If we wished to specify our own scale for the **existing** axes, we use the `xticks()` and `yticks()` commands, which take as input a range of values - usually in the form of a numpy range object.

In [21]:

```
plt.plot(X, Y, label='sin(x)')
plt.plot(X, Y2, label='cos(x)')
plt.title('sin(x) and cos(x)')
# setting axis labels
plt.xlabel('values of x')
plt.ylabel('f(x)')
plt.legend()

# setting up custom ticks in x-axis
plt.xticks(
    np.arange(0, 2*np.pi + 0.1, 0.5*np.pi),    # location of ticks
    ['0', '$\\frac{\pi}{2}$', '$\pi$', '$\\frac{3\pi}{2}$', '2$pi$']       # tex
t labels (optional)
)
# setting up custom ticks in y-axis
plt.yticks(
    np.arange(-1, 1 + 0.1, 0.5)    # location of ticks
)

plt.show()
```

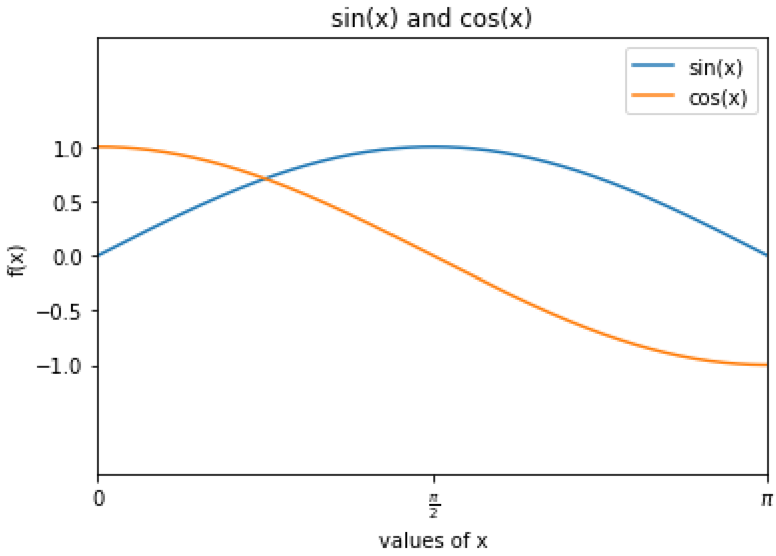


Axis limits

If we would like to specify our own range, or limits, for the axes, we can use the `xlim()` and `ylim()` commands. This takes a list object and fits the axes to it, scaling the graph appropriately.

In [22]:

```
plt.plot(X, Y, label='sin(x)')
plt.plot(X, Y2, label='cos(x)')
plt.title('sin(x) and cos(x)')
# setting axis labels
plt.xlabel('values of x')
plt.ylabel('f(x)')
plt.legend()

# setting up custom ticks in x-axis
plt.xticks(
    np.arange(0, 2*np.pi + 0.1, 0.5*np.pi),    # location of ticks
    ['0', '$\\frac{\pi}{2}$', '$\pi$', '$\\frac{3\pi}{2}$', '2$pi$']        # text labels (optional)
)
# setting up custom ticks in y-axis
plt.yticks(
    np.arange(-1, 1 + 0.1, 0.5)    # location of ticks
)

# setting x axis limits
plt.xlim([0, np.pi])
# setting y axis limits
plt.ylim([-2, 2])

plt.show()
```

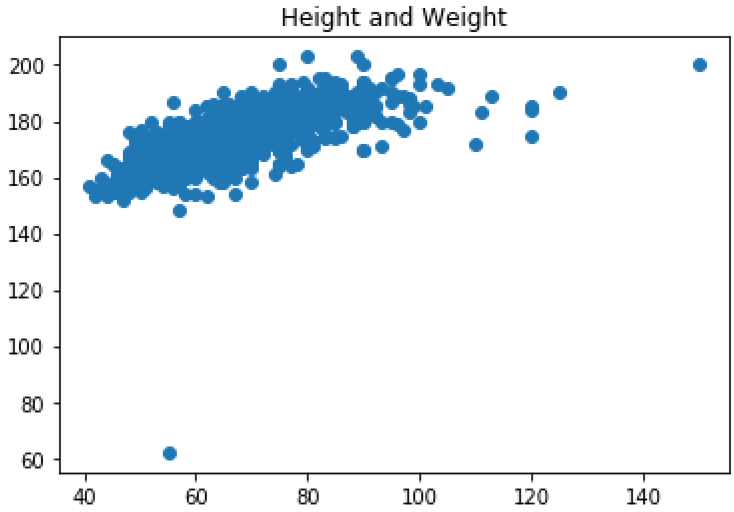


Below, we will be demonstrate the use of the MATLAB API on a real data set, containing survey responses on the lifestyle/music prerferences of Slovakian youths.

First, we create a scatter plot as we did with the OO API.

In [23]:

```
x = df_resp['Weight']
y = df_resp['Height']

plt.scatter(x,y)

plt.title('Height and Weight')

plt.savefig('SimpleScatter.png')
```



Next, we want to plot the the number of participants we have of each gender. To do this, we use the `value_counts()` function from `pandas` .

In [24]:

```
df_resp['Gender'].value_counts()
```

Out[24]:

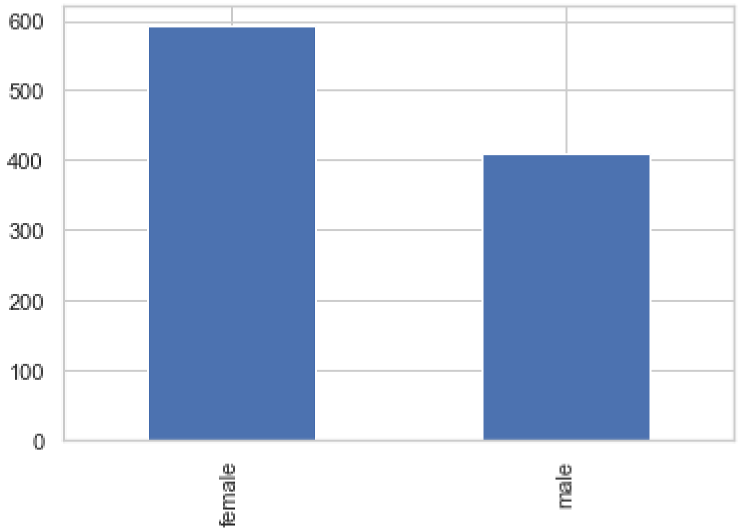
```
female    593
male      411
Name: Gender, dtype: int64
```

In [52]:

```
df_resp['Gender'].value_counts().plot(kind='bar')
```

Out[52]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x120545950>
```



Exercise

- #### Plot a graph of tan(x) between 0 and 2 π
- #### Add to this a graph of x^2
- #### Use dataframes for the above ^^^^^^
- #### Change the x axis to go up in intervals of $\pi/2$, and y axis to go up in 0.5.
- #### Generate a title and a legend

Seaborn

Seaborn is the data scientist's go to visualisation tool. Here, we will demonstrate some of it's features and properties. Seaborn uses Pandas objects to produce it's graphics.

In [26]:

```
import seaborn as sns
import pandas as pd
```

Let's read in some data:

The responses.csv file contains the results of a young persons survey contacted on a group on undergraduates in an Eastern European university.

We are only interested in Age, Gender, Height and Weight columns this time, so we will drop all other columns.

In [27]:

```
df = df_resp[['Age', 'Gender', 'Height', 'Weight']].copy()

df.head()
```

Out[27]:

	Age	Gender	Height	Weight
0	20.0	female	163.0	48.0
1	19.0	female	163.0	58.0
2	20.0	female	176.0	67.0
3	22.0	female	172.0	59.0
4	20.0	female	170.0	59.0

Let's first clean the data. At the moment, we will consider cleaning the data to be removing those rows which have empty/null values.

In [28]:

```
df = df.dropna()
df.head()
```

Out[28]:

	Age	Gender	Height	Weight
0	20.0	female	163.0	48.0
1	19.0	female	163.0	58.0
2	20.0	female	176.0	67.0
3	22.0	female	172.0	59.0
4	20.0	female	170.0	59.0

Plot distributions

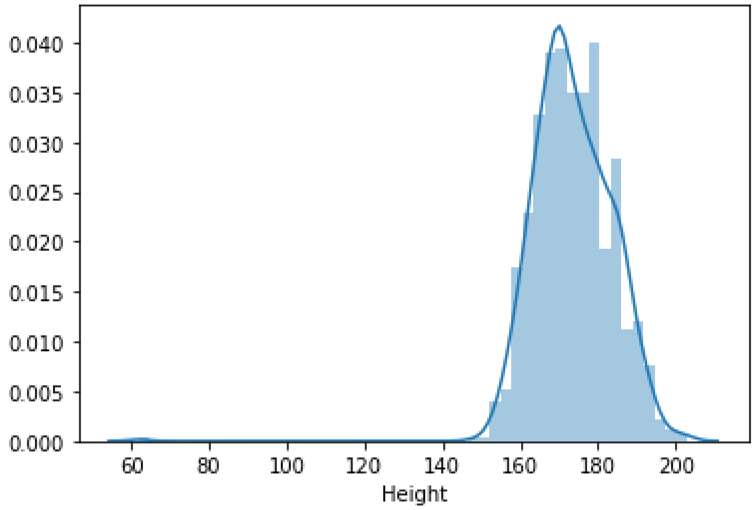
To plot the distribution of a column in our dataframe, we can use `distplot()` . This produces a plot conaining a generated bar graph, with a line plot overlayed to help ascertain the statistical distribution.

In [29]:

```
sns.distplot(df['Height'])
```

Out[29]:

<matplotlib.axes._subplots.AxesSubplot at 0x119ec27d0>

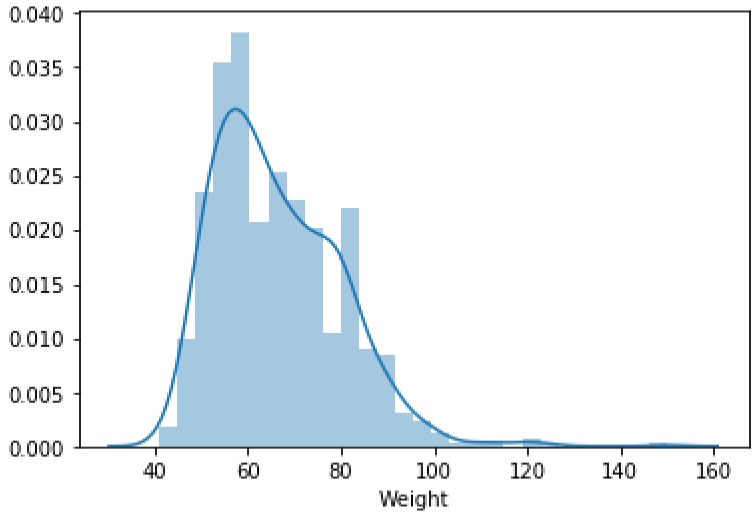


In [30]:

```
sns.distplot(df['Weight'])
```

Out[30]:

<matplotlib.axes._subplots.AxesSubplot at 0x118b29a50>



Box plots

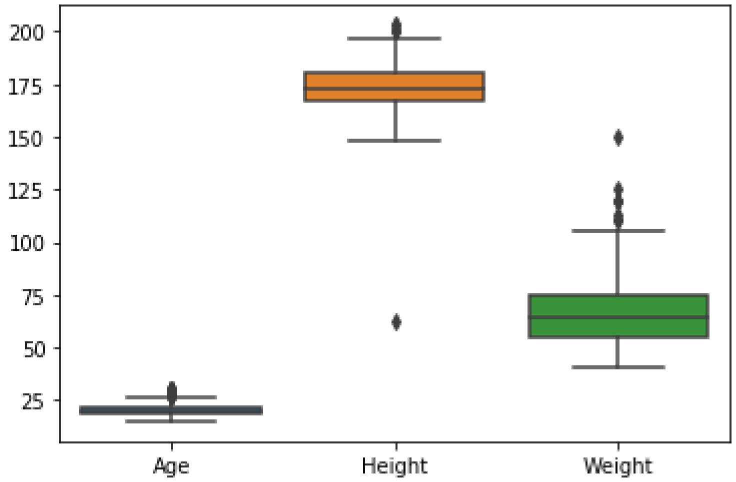
Box plots can be another useful tool in identifying the distribution and deviation of each column of the dataset. It also offers an idea of how many outlier datapoints there may be for each, the standard range of values, and the mean. We use the `boxplot()` command to generate one.

In [31]:

```
sns.boxplot(data=df)
```

Out[31]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x118aa2390>
```



Violin plots (hybrid box with distribution)

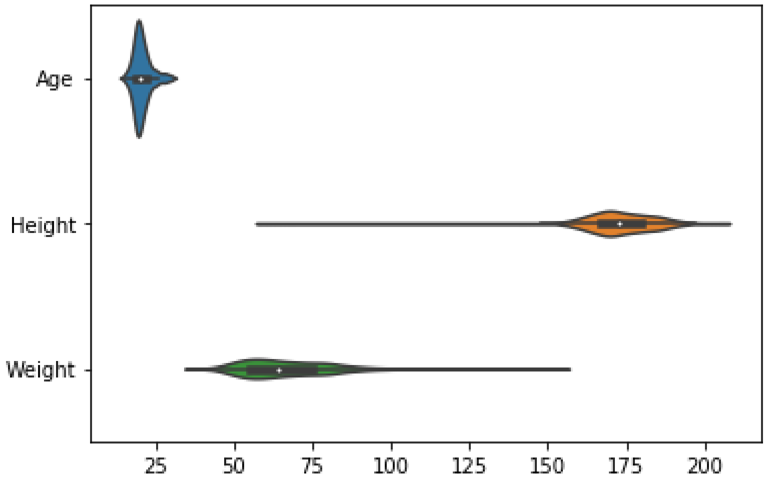
A voilin plot is essentially a combination of the two plots above. It displays both the distribution of the data as a symmetrical surface overlayed onto a boxplot. We use the command `violinplot()` to do this.

In [32]:

```
sns.violinplot(data=df, orient='h')  # plot in horizontal orientation
```

Out[32]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x118dbf7d0>
```



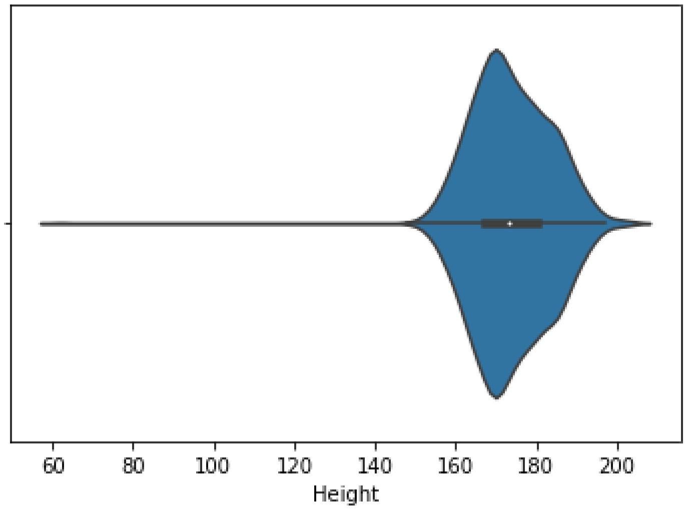
Focus on a single column

In [33]:

```
# choose a specific attribute
sns.violinplot(data=df, x='Height', orient='h')
```

Out[33]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x118ed7c90>
```



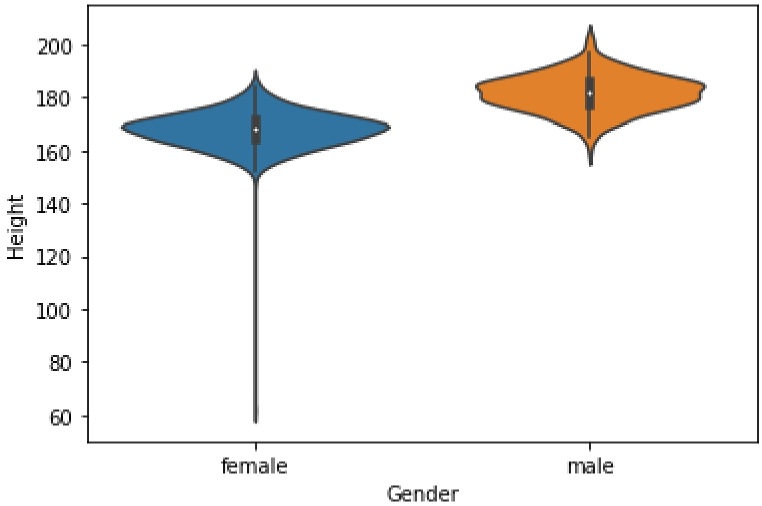
Compare two columns

In [34]:

```
# compare two attributes
sns.violinplot(data=df, x='Gender', y='Height')
```

Out[34]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x118e8d510>
```



We can also do a compare better by only plotting half of each violins.

Compare custom classifications

Suppose we want to see if the heights of people at different age bands differ.

The y axis of the violin plot does not have to be an existing column, it can be series, **providing** the series has the same index as the original data frame.

In the code below we create a new series (i.e. column) that computes the age bands:

In [35]:

```
def age_class(age):
    if age < 10:
        return 'under 10'
    elif age < 16:
        return 'teens under 16'
    elif age < 18:
        return 'teens over 16'
    else:
        return 'adult'
```

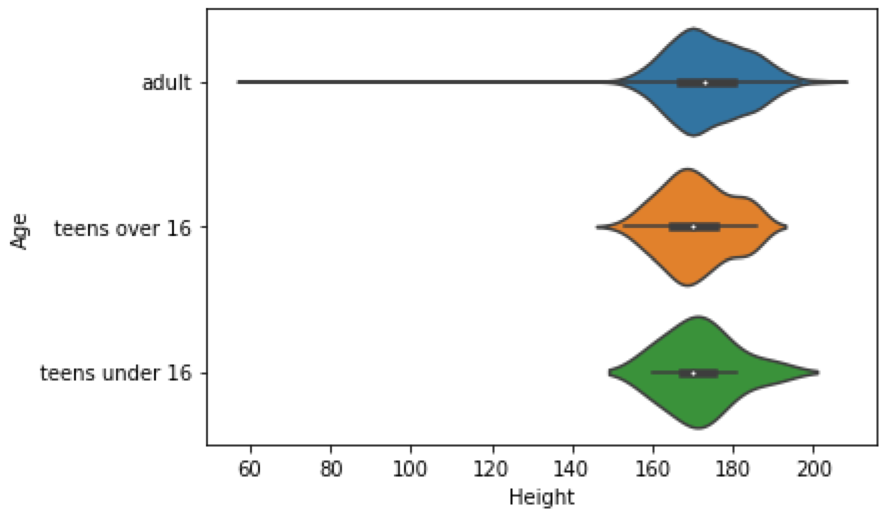
Now we can display a comparative plot for the distribution of height by age band.

In [36]:

```
sns.violinplot(data=df, x='Height', y=df['Age'].map(age_class))
```

Out[36]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x118fa96d0>
```



Exercise

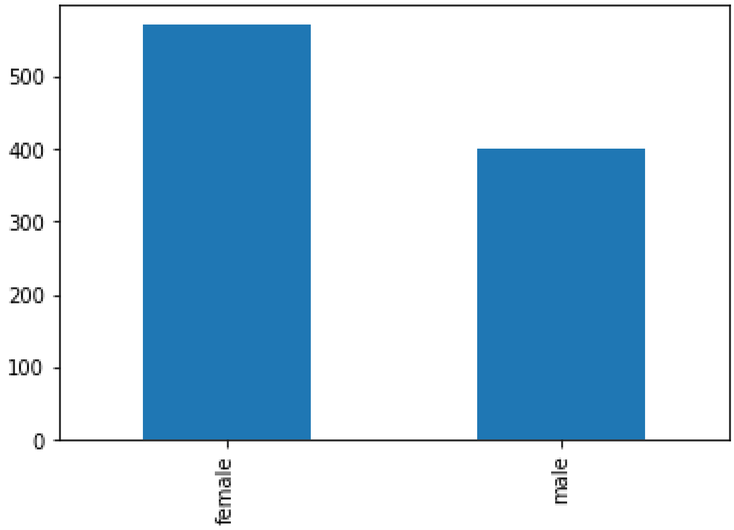
*** Plot the amount of men and women in the sample**

*** Create a Scatter plot of Height and weight**

*** Create a Regression plot of Height and Weight**

In [37]:

```
genders = df['Gender'].value_counts().plot(kind='bar')
#genders
```



In [38]:

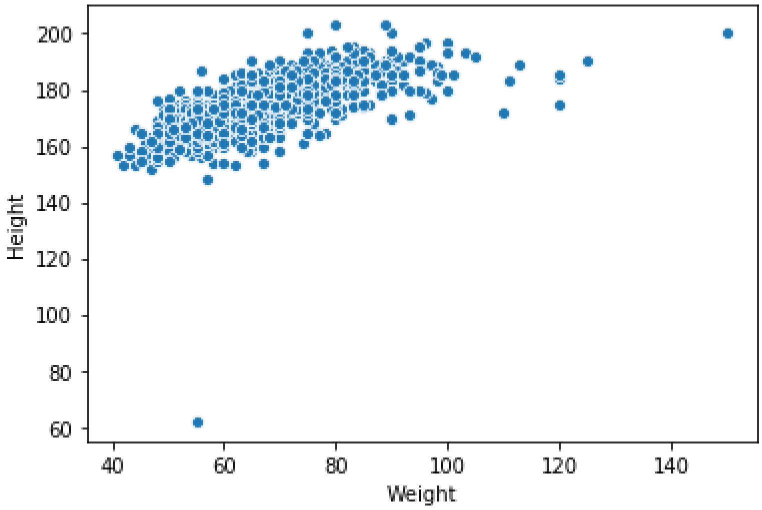
```
#sns.barplot(data=df, x='', y=df['Gender'].value_counts())
```

Scatter Plot

Probably the most common plot in statistics, it can be useful in both data exploration and the identification of trends.

In [39]:

```
# scatter plot
sns.scatterplot(data = df, x = 'Weight', y = 'Height')
plt.show()
```

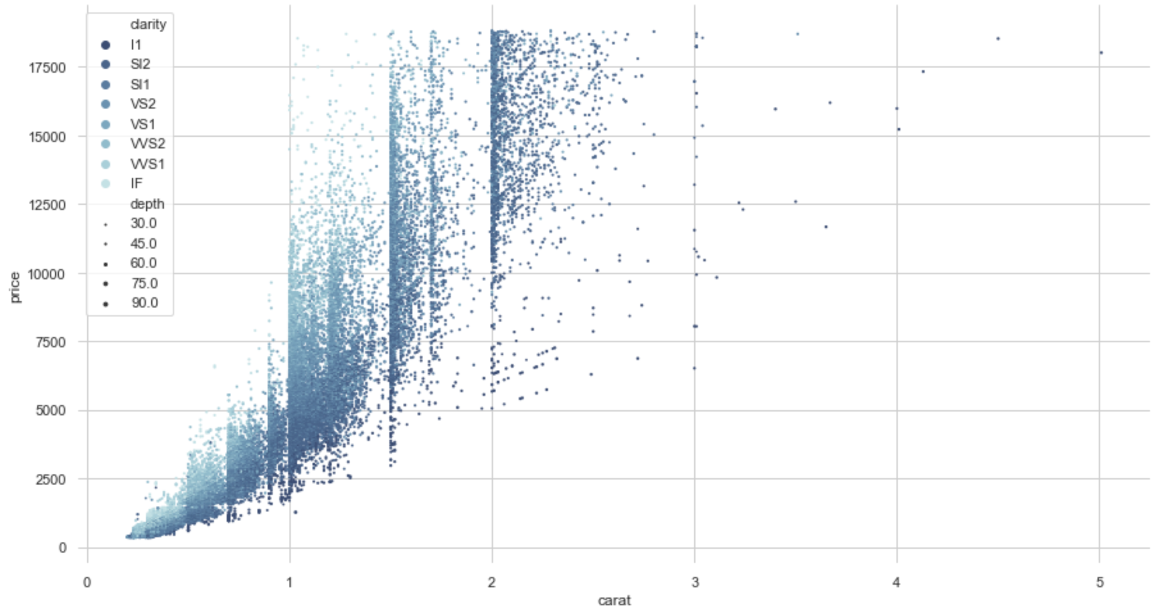


In [40]:

```
sns.set(style="whitegrid")

# Load the example iris dataset
diamonds = sns.load_dataset("diamonds")

# Draw a scatter plot while assigning point colors and sizes to different
# variables in the dataset
f, ax = plt.subplots(figsize=(15, 8))
sns.despine(f, left=True, bottom=True)
clarity_ranking = ["I1", "SI2", "SI1", "VS2", "VS1", "VVS2", "VVS1", "IF"]
sns.scatterplot(x="carat", y="price",
                hue="clarity", size="depth",
                palette="ch:r=-.2,d=.3_r",
                hue_order=clarity_ranking,
                sizes=(1, 8), linewidth=0,
                data=diamonds, ax=ax)
```

Out[40]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1193f4190>
```

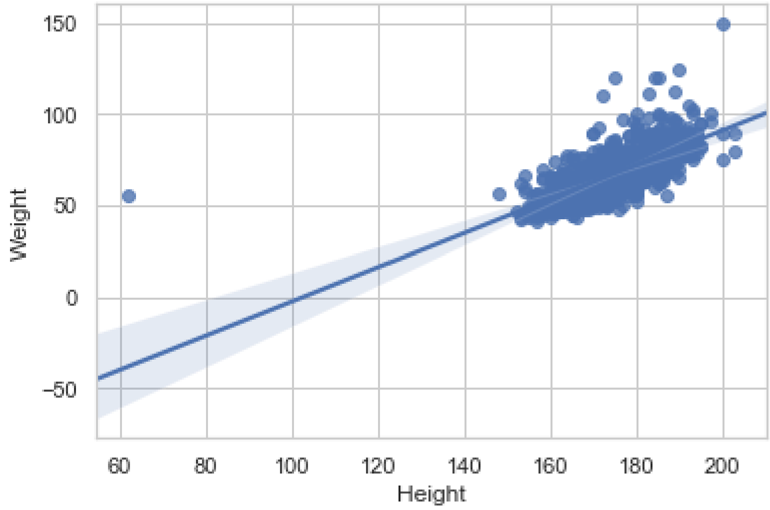


Regression Plot

Here, Seaborn essentially carries out machine learning for you. The regression plot is a scatterplot with a line of best fit inserted, and a shaded area of uncertainty added around it. To do the we use the `regplot()` command.

In [41]:

```
sns.regplot(data = df, x = 'Height', y = 'Weight')
plt.show()
```



Plotting Correlation

Below, we demonstrate how to plot a heatmap of the correlation between variables. This is incredibly useful when you want to explore the relationship of many elements on one another.

We first take a few columns which we wish to visualise.

In [42]:

```
df_corr = df_resp[['Horror',
     'Thriller',
     'Comedy',
     'Romantic',
     'Sci-fi',
    ]].copy()
df_corr.head()
```

Out[42]:

	Horror	Thriller	Comedy	Romantic	Sci-fi
0	4.0	2.0	5.0	4.0	4.0
1	2.0	2.0	4.0	3.0	4.0
2	3.0	4.0	4.0	2.0	4.0
3	4.0	4.0	3.0	3.0	4.0
4	4.0	4.0	5.0	2.0	3.0

Following this, we compute the the correlation matrix using the `pandas` function `.corr()` .

In [43]:

```
corrs = df_corr.corr()
corrs
```

Out[43]:

	Horror	Thriller	Comedy	Romantic	Sci-fi
Horror	1.000000	0.505953	0.102308	-0.126763	0.168398
Thriller	0.505953	1.000000	-0.002359	-0.161722	0.233373
Comedy	0.102308	-0.002359	1.000000	0.283501	0.045954
Romantic	-0.126763	-0.161722	0.283501	1.000000	-0.093513
Sci-fi	0.168398	0.233373	0.045954	-0.093513	1.000000

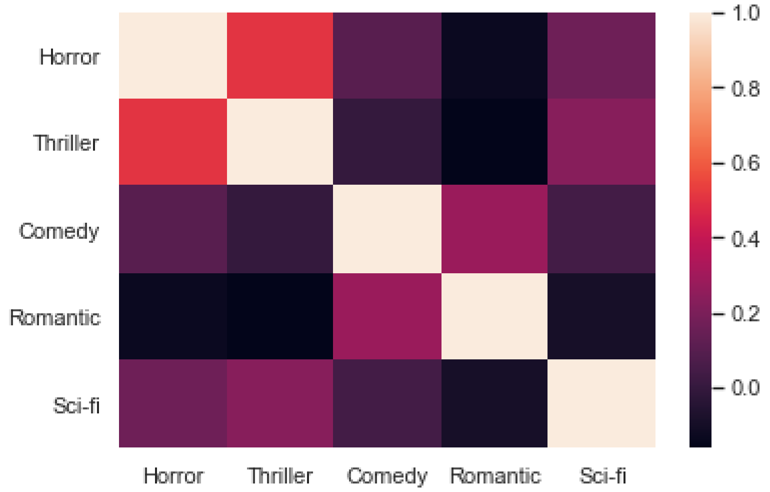
To generate the heatmap, we use the `seaborn` function `.heatmap(<dataframe>)`

In [44]:

```
sns.heatmap(corrs)
```

Out[44]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x11f892e10>
```



In order to colour the heatmap more appropriately, we can pass a center argument.

In [45]:

```
sns.heatmap(corrs, center=0.0)
```

Out[45]:

<matplotlib.axes._subplots.AxesSubplot at 0x11f850a50>

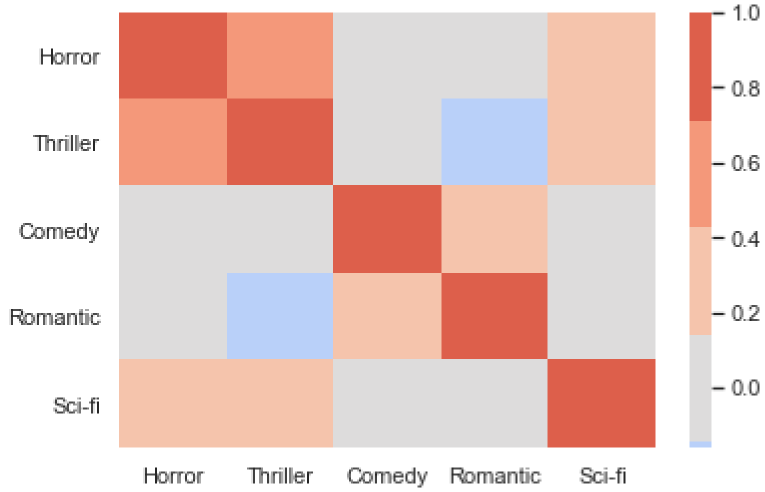
Finally, we may also change the colour palette to one of our choice.

In [46]:

```
sns.heatmap(corrs, center=0.0, cmap=sns.color_palette("coolwarm",7))
```

Out[46]:

<matplotlib.axes._subplots.AxesSubplot at 0x11e3dbe10>



Exercise

*** Choose 5 genres from the survey responses data set and load them into a DataFrame**

*** Create a correlation matrix**

*** Produce a heatmap of this matrix**

*** Colour the heatmap using a palette which is not coolwarm**

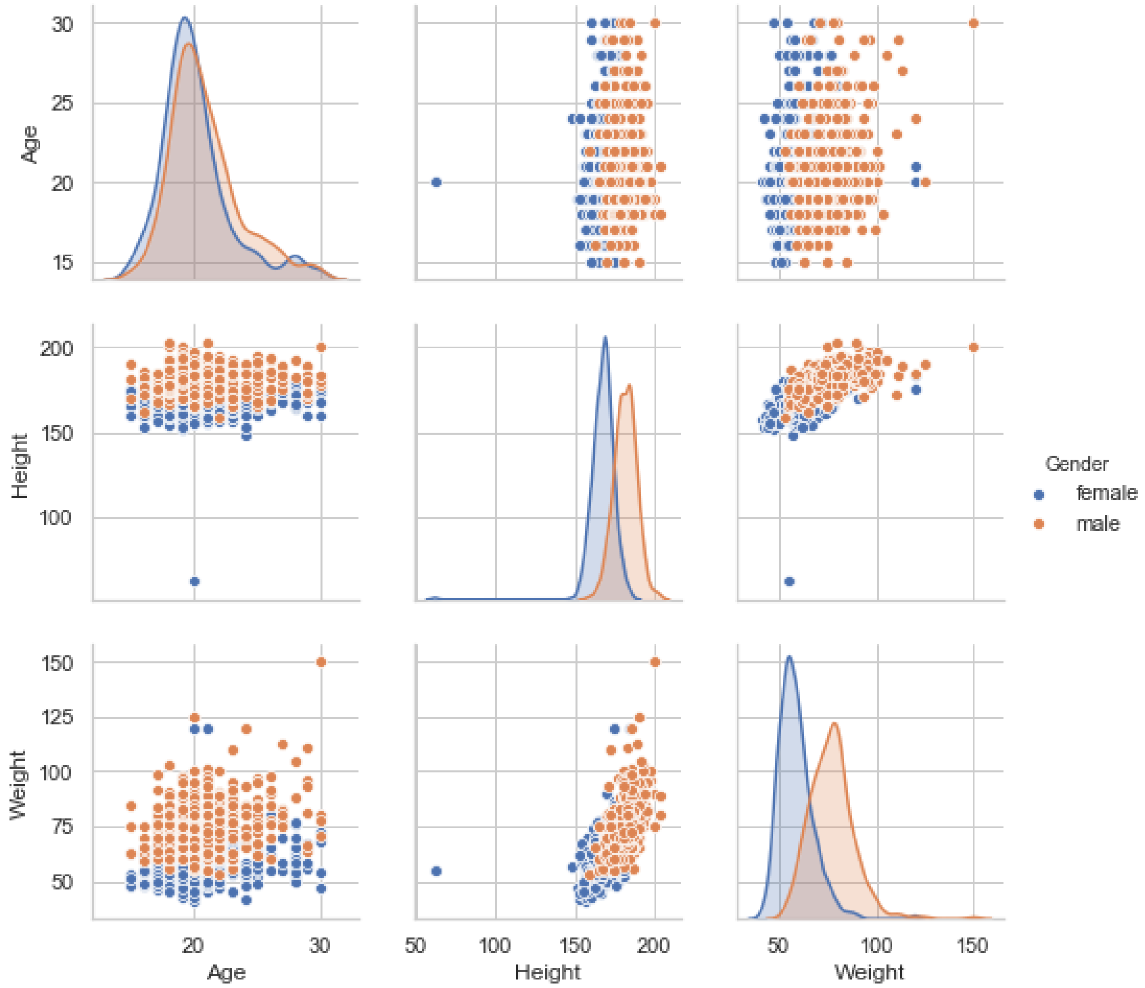
We can also easily produce multiple plots at one, using `pairplot()` and `gridplot()`

In [54]:

```
sns.pairplot(data=df, hue='Gender')
```

Out[54]:

```
<seaborn.axisgrid.PairGrid at 0x120573c50>
```

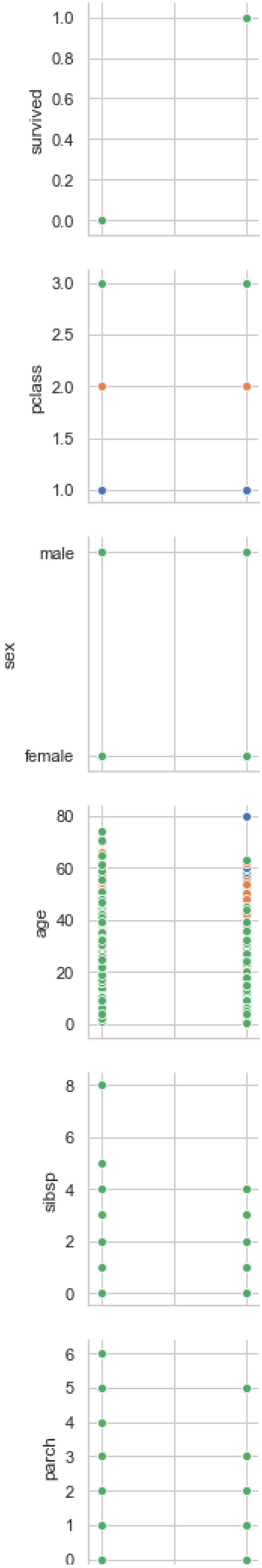


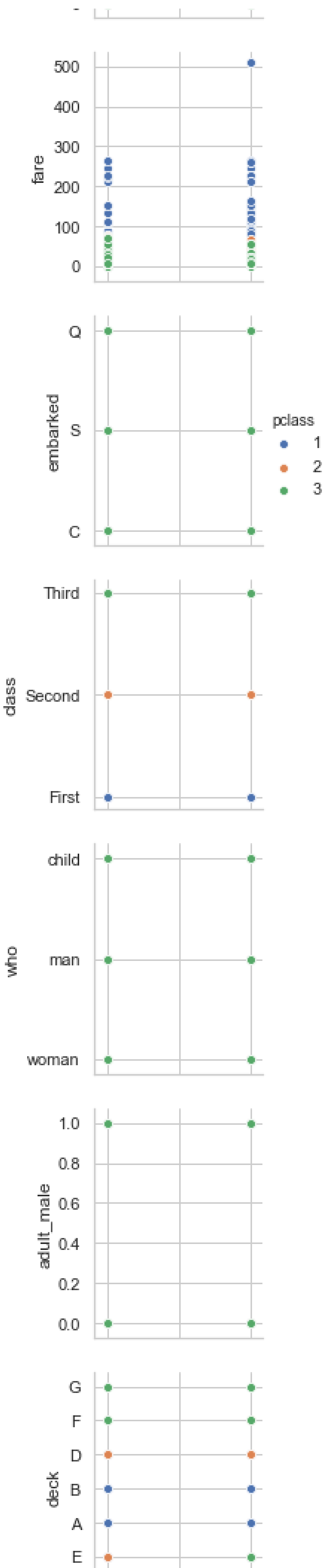
In [61]:

```
titanic = sns.load_dataset('Titanic')
sns.pairplot(data=titanic, hue='pclass', x_vars='survived', y_vars=titanic.colum
ns)
```

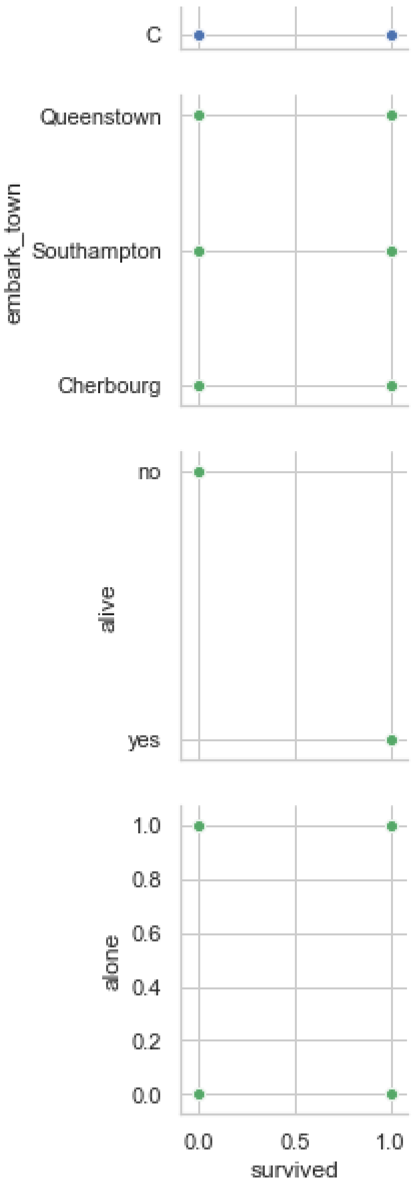
Out[61]:

```
<seaborn.axisgrid.PairGrid at 0x1a28f39c50>
```


survived
1.0
0.8
0.6
0.4
0.2
0.0
pclass
3.0
2.5
2.0
1.5
1.0
sex
male
female
age
80
60
40
20
0
sibsp
8
6
4
2
0
parch
6
5
4
3
2
1
0



500
400
300
fare
200
100
0
Q
embarked
S
C
pclass
1
2
3
Third
class
Second
First
child
who
man
woman
1.0
0.8
adult_male
0.6
0.4
0.2
0.0
G
F
D
deck
B
A
E

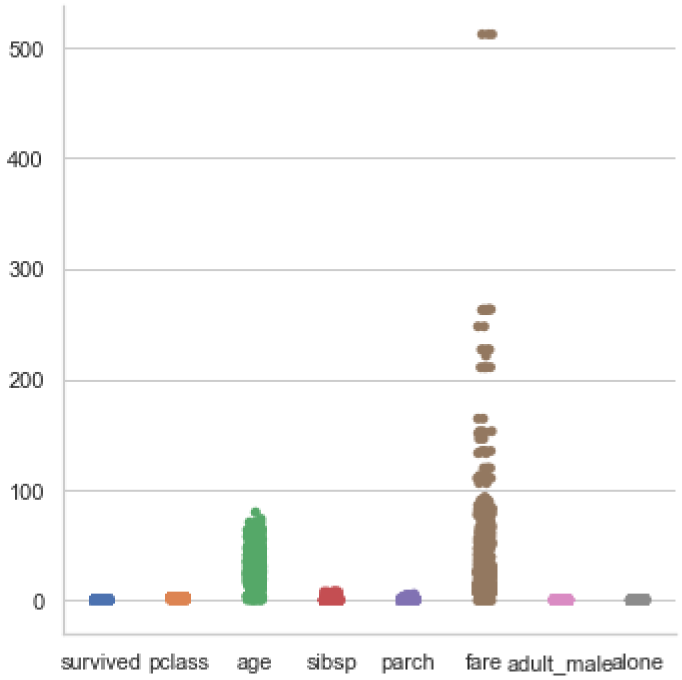

C
Queenstown
Southampton
Cherbourg
embark_town
no
alive
yes
1.0
0.8
0.6
0.4
0.2
0.0
alone
0.0
0.5
1.0
survived

In [63]:

```
sns.catplot(data=titanic)
```

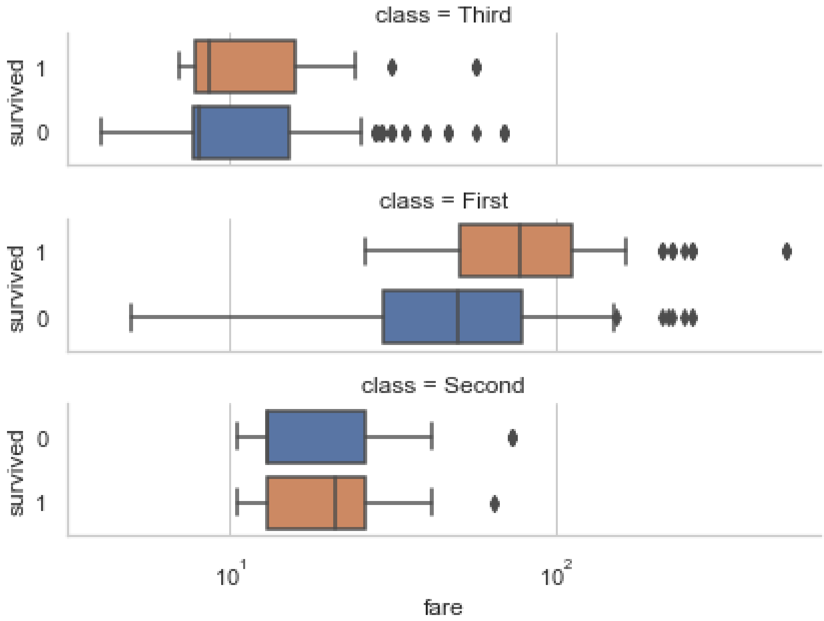
Out[63]:

```
<seaborn.axisgrid.FacetGrid at 0x120d1cb50>
```



In [62]:

```
g = sns.catplot(x="fare", y="survived", row="class",
                kind="box", orient="h", height=1.5, aspect=4,
                data=titanic.query("fare > 0"))
g.set(xscale="log");
```



In [47]:

```
norm_rock = df_resp["Rock"] / df_resp.groupby("Age")["Rock"].transform(sum)
norm_rock
```

Out[47]:

```
0       0.006793
1       0.006353
2       0.006793
3       0.006689
4       0.004076
          ...
1005    0.005435
1006    0.021739
1007    0.008511
1008    0.033333
1009    0.002062
Name: Rock, Length: 1010, dtype: float64
```

In [48]:

```
grouped_df = df_resp.groupby("Age")["Rock"]

for key, item in grouped_df:
    print(grouped_df.get_group(key), "\n\n")
```

```
128    5.0
132    4.0
158    4.0
202    2.0
263    4.0
265    2.0
306    4.0
312    4.0
338    5.0
748    4.0
950    5.0
Name: Rock, dtype: float64


119     2.0
134     3.0
190     4.0
235     5.0
293     5.0
309     4.0
318     1.0
321     5.0
333     4.0
341     2.0
348     2.0
351     5.0
396     3.0
409     4.0
490     2.0
509     1.0
551     2.0
556     1.0
560     3.0
565     2.0
730     5.0
770     1.0
787     2.0
882     4.0
895     3.0
935     2.0
940     3.0
947     3.0
1000    3.0
Name: Rock, dtype: float64


11     5.0
65     3.0
70     3.0
108    4.0
115    5.0
175    5.0
189    4.0
216    4.0
243    5.0
253    1.0
269    5.0
329    4.0
331    5.0
335    4.0
336    3.0
```

```
337    3.0
346    3.0
349    2.0
369    2.0
377    5.0
465    1.0
517    4.0
535    5.0
550    5.0
573    4.0
587    4.0
589    5.0
595    4.0
632    3.0
641    3.0
649    3.0
691    4.0
709    5.0
717    4.0
718    5.0
740    4.0
749    5.0
773    5.0
779    3.0
782    5.0
800    3.0
805    5.0
812    1.0
819    2.0
821    5.0
831    4.0
850    2.0
856    5.0
865    4.0
893    4.0
921    3.0
933    4.0
965    5.0
Name: Rock, dtype: float64


8       5.0
15      5.0
18      4.0
19      4.0
36      4.0
       ...
980     4.0
982     4.0
995     4.0
1001    3.0
1007    4.0
Name: Rock, Length: 123, dtype: float64


1      5.0
7      5.0
9      5.0
10     3.0
13     2.0
      ...
```

```
981    4.0
986    4.0
988    5.0
997    4.0
999    4.0
Name: Rock, Length: 210, dtype: float64


0       5.0
2       5.0
4       3.0
5       5.0
6       3.0
       ...
987     5.0
994     5.0
996     4.0
1003    5.0
1005    4.0
Name: Rock, Length: 194, dtype: float64


37      3.0
39      5.0
46      5.0
52      5.0
54      5.0
       ...
978     5.0
984     4.0
991     5.0
993     4.0
1009    1.0
Name: Rock, Length: 127, dtype: float64


3       2.0
14      5.0
22      NaN
26      4.0
31      3.0
       ...
958     1.0
970     4.0
983     3.0
1002    5.0
1004    3.0
Name: Rock, Length: 84, dtype: float64


38     4.0
55     3.0
60     4.0
61     4.0
62     4.0
118    4.0
120    5.0
122    5.0
181    4.0
210    5.0
254    3.0
```

```
270    5.0
296    3.0
360    5.0
403    4.0
419    5.0
520    2.0
527    4.0
568    3.0
570    3.0
594    4.0
612    4.0
618    5.0
645    2.0
648    3.0
667    2.0
670    3.0
678    5.0
695    4.0
703    4.0
708    3.0
738    5.0
739    5.0
761    1.0
794    2.0
817    5.0
839    3.0
846    2.0
866    5.0
879    4.0
885    4.0
897    3.0
905    4.0
911    5.0
923    4.0
963    4.0
975    5.0
Name: Rock, dtype: float64


12     5.0
21     5.0
67     1.0
78     3.0
86     5.0
93     3.0
105    4.0
154    5.0
204    5.0
222    5.0
271    5.0
334    3.0
344    3.0
365    5.0
391    3.0
423    4.0
636    4.0
654    3.0
657    3.0
733    4.0
762    3.0
793    2.0
```

```
818    4.0
840    5.0
842    5.0
904    3.0
942    2.0
959    3.0
Name: Rock, dtype: float64


30      4.0
44      4.0
152     4.0
156     5.0
206     4.0
226     5.0
303     5.0
379     4.0
385     5.0
387     2.0
388     4.0
402     5.0
472     3.0
496     3.0
537     4.0
543     4.0
608     1.0
635     4.0
663     3.0
675     4.0
680     5.0
714     5.0
745     4.0
756     5.0
760     5.0
766     4.0
776     5.0
867     3.0
870     3.0
1008    4.0
Name: Rock, dtype: float64


40     1.0
157    4.0
174    1.0
177    5.0
193    5.0
219    1.0
447    1.0
539    5.0
700    4.0
742    4.0
754    3.0
755    5.0
768    3.0
878    4.0
951    3.0
Name: Rock, dtype: float64


33      3.0
```

```
45       4.0
215      1.0
246      5.0
286      4.0
394      4.0
546      5.0
619      3.0
668      5.0
796      1.0
803      5.0
854      3.0
929      2.0
1006     1.0
Name: Rock, dtype: float64
```

```
72      5.0
422     1.0
452     3.0
486     2.0
503     4.0
525     5.0
530     2.0
584     3.0
590     5.0
664     2.0
672     4.0
677     3.0
681     5.0
684     4.0
898     5.0
909     3.0
998     5.0
Name: Rock, dtype: float64
```

```
186     2.0
191     2.0
343     4.0
461     3.0
473     4.0
542     4.0
637     5.0
715     5.0
716     3.0
824     5.0
990     4.0
Name: Rock, dtype: float64
```

```
221     5.0
392     3.0
478     4.0
683     5.0
795     5.0
801     5.0
844     3.0
853     3.0
989     5.0
992     4.0
Name: Rock, dtype: float64
```