Portfolio 1 - Analysis of Cycling Data

In [1]:
```
import sys
sys.path.append("used_functions/")
import custom_functions as custfun
import importlib
importlib.reload(custfun)
import pandas as pd
from pandas.plotting import scatter_matrix
import numpy as np
from matplotlib import pyplot as plt
import matplotlib as mpl
import matplotlib.dates as mdates
from datetime import timedelta
import re
import warnings
%matplotlib inline
plt.style.use("ggplot")
```

Loading Data

The first dataset is an export of my ride data from Strava (https://strava.com/), an online social network site for cycling and other sports. This data is a log of every ride since the start of 2018 and contains summary data like the distance and average speed. It was exported using the script `stravaget.py` which uses the stravalib module to read data. Some details of the fields exported by that script can be seen in the documentation for stravalib (https://pythonhosted.org/stravalib/api.html#stravalib.model.Activity).

The exported data is a CSV file so that's easy to read, however the date information in the file is recorded in a different timezone (UTC) so we need to do a bit of conversion. In reading the data I'm setting the index of the data frame to be the datetime of the ride.

In [2]:
```
# Loading strava_exoport.csv
strava = pd.read_csv('Portfolio1/data/strava_export.csv', index_col = 'date', parse_dates = True)

# Converting the timezone to Aus/Syd
strava.index = strava.index.tz_localize('Australia/Sydney')

print("Shape of the dataframe(row, column) : ", strava.shape)
strava.head()
```

```
Shape of the dataframe(row, column) :  (268, 10)
```

Out[2]:

date	average_heartrate	average_temp	average_watts	device_watts	distance	elapsed_time	elevation_gain	kudos	moving
2018-01-02 20:47:51+11:00	100.6	21.0	73.8	False	15.2	94	316.00 m	10	
2018-01-04 01:36:53+11:00	NaN	24.0	131.7	False	18.0	52	236.00 m	5	
2018-01-04 02:56:00+11:00	83.1	25.0	13.8	False	0.0	3	0.00 m	2	
2018-01-04 05:37:04+11:00	110.1	24.0	113.6	False	22.9	77	246.00 m	8	
2018-01-05 19:22:46+11:00	110.9	20.0	147.7	True	58.4	189	676.00 m	12	

The second dataset comes from an application called GoldenCheetah (https://www.goldencheetah.org/) which provides some analytics services over ride data. This has some of the same fields but adds a lot of analysis of the power, speed and heart rate data in each ride. This data overlaps with the Strava data but doesn't include all of the same rides.

Again we create an index using the datetime for each ride, this time combining two columns in the data (date and time) and localising to Sydney so that the times match those for the Strava data.

In [3]:
```
# Loading the data from cheetah.csv
cheetah = pd.read_csv('Portfolio1/data/cheetah.csv', skipinitialspace = True)

# Setting combination of time and date as index
cheetah.index = pd.to_datetime(cheetah['date'] + ' ' + cheetah['time'])

# Converting the timezone to Aus/Syd
cheetah.index = cheetah.index.tz_localize('Australia/Sydney')

print("Shape of the dataframe(row, column) : ", cheetah.shape)
cheetah.head()
```

```
Shape of the dataframe(row, column) :  (251, 362)
```

Out[3]:

	date	time	filename	axPower	aPower Relative Intensity	aBikeScore	Skiba aVI	aPower Response Index	alsoPower	aIF
2018-01-28 06:39:49+11:00	01/28/18	06:39:49	2018_01_28_06_39_49.json	202.211	0.75452	16.6520	1.31920	1.67755	223.621	0.83441
2018-01-28 07:01:32+11:00	01/28/18	07:01:32	2018_01_28_07_01_32.json	226.039	0.84343	80.2669	1.21137	1.54250	246.185	0.91860
2018-02-01 08:13:34+11:00	02/01/18	08:13:34	2018_02_01_08_13_34.json	0.000	0.00000	0.0000	0.00000	0.00000	0.000	0.00000
2018-02-06 08:06:42+11:00	02/06/18	08:06:42	2018_02_06_08_06_42.json	221.672	0.82714	78.8866	1.35775	1.86002	254.409	0.94929
2018-02-07 17:59:05+11:00	02/07/18	17:59:05	2018_02_07_17_59_05.json	218.211	0.81422	159.4590	1.47188	1.74658	233.780	0.87231

5 rows × 362 columns

The GoldenCheetah data contains many many variables (columns) and I won't go into all of them here. Some that are of particular interest for the analysis below are:

Here are definitions of some of the more important fields in the data. Capitalised fields come from the GoldenCheetah data while lowercase_fields come from Strava. There are many cases where fields are duplicated and in this case the values should be the same, although there is room for variation as the algorithm used to calculate them could be different in each case.

- Duration - overall duration of the ride, should be same as elapsed_time
- Time Moving - time spent moving (not resting or waiting at lights), should be the same as moving_time
- Elevation Gain - metres climbed over the ride
- Average Speed - over the ride
- Average Power - average power in watts as measured by a power meter, relates to how much effort is being put in to the ride, should be the same as * average_watts' from Strava
- Nonzero Average Power - same as Average Power but excludes times when power is zero from the average
- Average Heart Rate - should be the same as average_heartrate
- Average Cadence - cadence is the rotations per minute of the pedals
- Average Temp - temperature in the environment as measured by the bike computer (should be same as average_temp)
- VAM - average ascent speed - speed up hills
- Calories (HR) - Calorie expendature as estimated from heart rate data
- 1 sec Peak Power - this and other 'Peak Power' measures give the maximum power output in the ride over this time period. Will be higher for shorter periods. High values in short periods would come from a very 'punchy' ride with sprints for example.
- 1 min Peak Hr - a similar measure relating to Heart Rate
- NP - Normalised Power, a smoothed average power measurement, generally higher than Average Power
- TSS - Training Stress Score, a measure of how hard a ride this was
- device_watts - True if the power (watts) measures were from a power meter, False if they were estimated
- distance - distance travelled in Km
- kudos - likes from other Strava users (social network)
- workout_type - one of 'Race', 'Workout' or 'Ride'

Some of the GoldenCheetah parameters are defined in their documentation (https://github.com/GoldenCheetah/GoldenCheetah/wiki/UG_Glossary).

Your Tasks

Your first task is to combine these two data frames using the `join` method of Pandas (https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html#joining-on-index). The goal is to keep only those rows of data that appear in **both** data frames so that we have complete data for every row.

In [4]:
```
# Carrying out inner join on both dataframe
## strava_plus_cheetah = strava.join(cheetah, how = "inner")

## Join is not working due to some problems so reading csv file
strava_plus_cheetah = pd.read_csv("Portfolio1/data/strava_plus_cheetah.csv", parse_dates = True)

# Setting an index
strava_plus_cheetah.index = strava_plus_cheetah["index"]

# Deleting index column and name of index of df
del strava_plus_cheetah["index"]
del strava_plus_cheetah.index.name

print("Shape of the dataframe(row, column) :", strava_plus_cheetah.shape)
strava_plus_cheetah.head()
```

```
Shape of the dataframe(row, column) : (243, 372)
```

Out[4]:

	average_heartrate	average_temp	average_watts	device_watts	distance	elapsed_time	elevation_gain	kudos	moving
2018-01-28 06:39:49+11:00	120.6	21.0	153.4	True	7.6	17	95.00 m	4	
2018-01-28 07:01:32+11:00	146.9	22.0	187.7	True	38.6	67	449.00 m	19	
2018-02-01 08:13:34+11:00	109.8	19.0	143.0	False	26.3	649	612.00 m	6	
2018-02-06 08:06:42+11:00	119.3	19.0	165.9	True	24.3	69	439.00 m	6	
2018-02-07 17:59:05+11:00	124.8	20.0	151.0	True	47.1	144	890.00 m	10	

5 rows × 372 columns

Required Analysis

1. Remove rides with no measured power (where device_watts is False) - these are commutes or MTB rides

In [5]:
```
# Creating a boolean flag based on condition devise_watts is false
flag = strava_plus_cheetah["device_watts"] != False

# Filtering data based on boolean flag
strava_plus_cheetah = strava_plus_cheetah[flag]

print("Shape of the dataframe(row, column) :", strava_plus_cheetah.shape)

## strava_plus_cheetah = strava_plus_cheetah.drop((strava_plus_cheetah[not flag]).index) # Another way to achieve above task
```

```
Shape of the dataframe(row, column) : (209, 372)
```

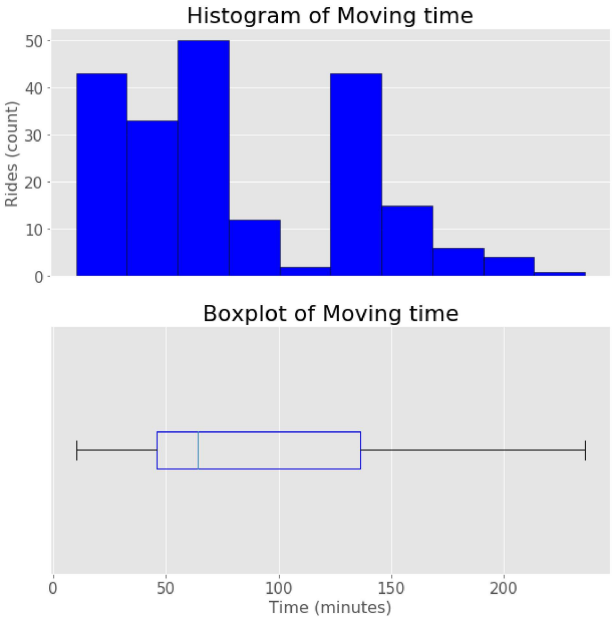
2. Look at the distributions of some key variables: time, distance, average speed, average power, TSS. Are they normally distributed? Skewed?

2.0 Setting matplotlib parameters

In [6]:
```
mpl.rcParams['axes.titlesize'] = 22                                    # -Change title size of a plot
mpl.rcParams['axes.labelsize'] = 16                                    # -Change label size(x and y) of a plot
mpl.rcParams['xtick.labelsize'] = 15                                   # -Change xticks size of a plot
mpl.rcParams['ytick.labelsize'] = 15                                   # -Change yticks size of a plot
```

2.1 moving_time

In [7]:
```
# For more detials about this function check used_function/custom_functions.py
custfun.create_histogram_plus_boxplot(strava_plus_cheetah["moving_time"], "Moving time", "blue",
                                      "Rides (count)", "Time (minutes)", (10, 10))
```



Graph Explanation

moving_time : Time spent moving on a bike(in Minutes)

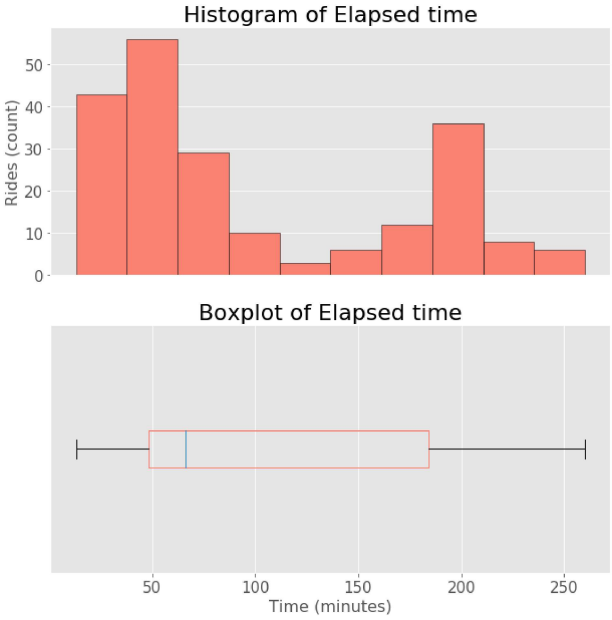
By looking at the histogram and boxplot, One can say that the distribution is not normal. In fact it is **right skewed**(Right box is longer than left box in boxplot). Generally such type of distribution is called **bimodal**(check density plot below) as you can see two peaks in histogram(one at median value approx, 65 minutes and another peak at approx. 135 minutes). We can also figure out following from the boxplot. **Boxplots are also called 5 number summary.**

- Median(Q2) ~= 65 mintues (50% of the rides have moving time less than 65 minutes **OR** 50% of the rides have moving time greater than 65 minutes)
- 1st Quartile(Q1) ~= 45 minutes (25% of the rides have moving time below 45 minutes)
- 3rd Quartile(Q3) ~= 135 minutes (75% of the rides have moving time below 135 minutes **OR** 25% of the rides have moving time above 135 minutes)
- Minimun ~= 10 minutes (Minimun moving time amongst all the rides - left end of the whisker)
- Maximum ~= 240 minutes (Maximum moving time amongst all the rides **-** right end of the whisker)

No outliers are detected

2.2 elapsed_time

In [8]:
```
# For more detials about this function check used_function/custom_functions.py
custfun.create_histogram_plus_boxplot(strava_plus_cheetah["elapsed_time"], "Elapsed time", "salmon",
                                      "Rides (count)", "Time (minutes)", (10, 10))
```



Graph Explanation

elapsed_time : Total time spent on a bike including standing at traffic lights and resting (in Minutes)

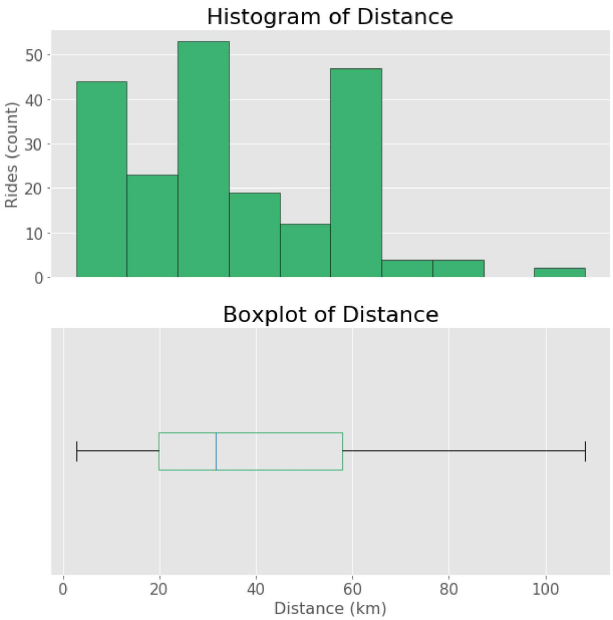
Distribution is not normal. In fact it is **right skewed**(Right box is longer than left box in boxplot). Generally such type of distribution is called **bimodal**(check density plot below) as you can see two peaks in histogram(one at median value approx, 70 minutes and another peak at approx. 200 minutes). We can also figure out following from the boxplot. **Boxplots are also called 5 number summary.**

- Median(Q2) ~= 70 mintues (50% of the rides have elapsed time below 70 minutes)
- 1st Quartile(Q1) ~= 50 minutes (25% of the rides have elapsed time below 50 minutes)
- 3rd Quartile(Q3) ~= 180 minutes (75% of the rides have elapsed time below 180 minutes **OR** 25% of the rides have moving time above 180 minutes)
- Minimun ~= 13 minutes (Minimun elapsed time amongst all the rides - left end of the whisker)
- Maximum ~= 265 minutes (Maximum elapsed time amongst all the rides - right end of the whisker)

No outliers are detected

2.3 distance

In [9]:
```
# Removing the rows where distance = 0 km
strava_plus_cheetah = strava_plus_cheetah[strava_plus_cheetah["distance"] != 0]

# For more detials about this function check used_function/custom_functions.py
custfun.create_histogram_plus_boxplot(strava_plus_cheetah["distance"], "Distance", "mediumseagreen",
                                      "Rides (count)", "Distance (km)", (10, 10))
```



Graph Explanation

Distance : Total Distance travelled on a bike (in KM)

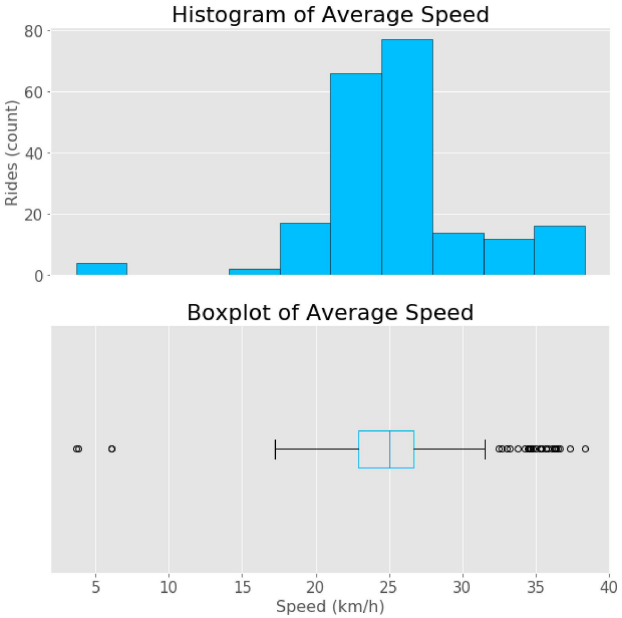
Distribution is **right skewed**(Right box is longer than left box in boxplot). Generally such type of distribution is called **bimodal**(check density plot below) as you can see two peaks in histogram(one at median value approx, 30 km and another peak at approx. 60 km). We can also figure out following from the boxplot. **Boxplots are also called 5 number summary.**

- Median(Q2) ~= 30 km (50% of the rides have travelled distance below 30 km)
- 1st Quartile(Q1) ~= 20 km (25% of the rides have travelled distance below 20 km)
- 3rd Quartile(Q3) ~= 58 km (75% of the rides have travelled distance below 58 km **OR** 25% of the rides have travelled distance above 58 km)
- Minimun ~= 3 km (Minimun distance travelled amongst all the rides - left end of the whisker)
- Maximum ~= 110 km (Maximum distance travelled amongst all the rides **-** right end of the whisker)

No outliers are detected

2.4 average speed

In [10]:
```
# For more detials about this function check used_function/custom_functions.py
custfun.create_histogram_plus_boxplot(strava_plus_cheetah["Average Speed"], "Average Speed", "deepskyblue",
                                      "Rides (count)", "Speed (km/h)", (10, 10))
```



Graph Explanation

Average Speed : Average Speed of a bike (in km/h)

Distribution is **nearly normal**(Right box has almost equal length as left box in boxplot)(check density plot below). We can also figure out following from the boxplot. **Boxplots are also called 5 number summary.**

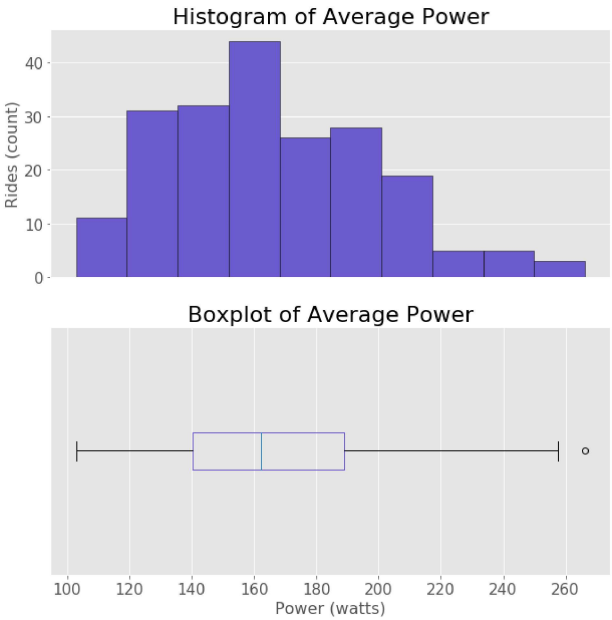
- Median(Q2) ~= 25 km/h (50% of the rides have speed below 25 km/h)
- 1st Quartile(Q1) ~= 23 km/h (25% of the rides have speed below 23 km/h)
- 3rd Quartile(Q3) ~= 26 km/h (75% of the rides have speed below 26 km/h **OR** 25% of the rides have speed above 26 km/h)
- Minimun ~= 18 km/h (Minimun average speed amongst all the rides - left end of the whisker)
- Maximum ~= 32 km/h (Maximum average speed amongst all the rides - right end of the whisker)

Outliers detected

- Outliers less than 7 km/h and and more than 32 km/h

2.5 average power

In [11]:
```
# Removing rows where Average power = 0
strava_plus_cheetah = strava_plus_cheetah[strava_plus_cheetah["Average Power"] != 0]

# For more detials about this function check used_function/custom_functions.py
custfun.create_histogram_plus_boxplot(strava_plus_cheetah["Average Power"], "Average Power", "slateblue",
                                      "Rides (count)", "Power (watts)", (10, 10))
```



Graph Explanation

Average Power : Average Power of a bike (in watts)

Distribution is **normal**(check density plot below). We can also figure out following from the boxplot. **Boxplots are also called 5 number summary.**

- Median(Q2) ~= 155 watts (50% of the rides generated power below 155 watts)
- 1st Quartile(Q1) ~= 125 watts (25% of the rides generated power below 125 watts)
- 3rd Quartile(Q3) ~= 180 watts (75% of the rides generated power below 180 watts **OR** 25% of the rides have speed above 180 watts)
- Minimun ~= 100 watts (Minimun average power amongst all the rides - left end of the whisker)
- Maximum ~= 260 watts (Maximum average power amongst all the rides **-** right end of the whisker)

Outliers detected

- Outliers having average power more than 260 watts

2.6 TSS

In [12]:
```
# For more detials about this function check used_function/custom_functions.py
custfun.create_histogram_plus_boxplot(strava_plus_cheetah["TSS"], "Training stress score", "mediumorchid",
                                      "Rides (count)", "Training Stress Score", (10, 10))
```

Graph Explanation

Training Stress Score

Distribution is **Right skewed**(Right box has more length than left box in boxplot)(check density plot below). We can also figure out following from the boxplot. **Boxplots are also called 5 number summary.**

- Median(Q2) ~= 80 (50% of the rides have TSS below 80)
- 1st Quartile(Q1) ~= 30 (25% of the rides have TSS below 30)
- 3rd Quartile(Q3) ~= 150 (75% of the rides have TSS below 150 **OR** 25% of the rides have TSS above 150)
- Minimun ~= 0 (Minimun TSS amongst all the rides - left end of the whisker)
- Maximum ~= 280 (Maximum TSS amongst all the rides - right end of the whisker)

Outliers detected

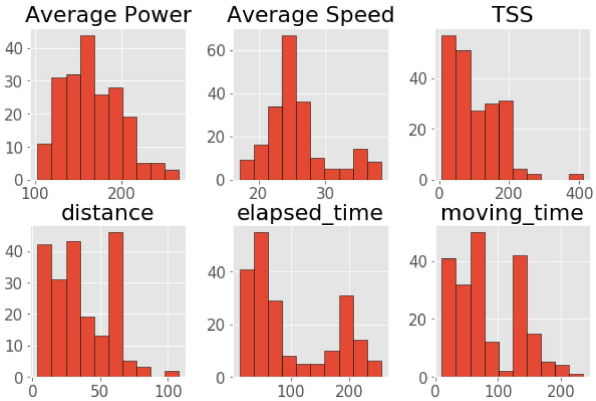
- Outliers detected around 400 TSS.

Other ways to plot the graphs(Rather than plotting all of them individually you can use below code)

Plotting them together helps us to compare the characterstics of variables with other variables

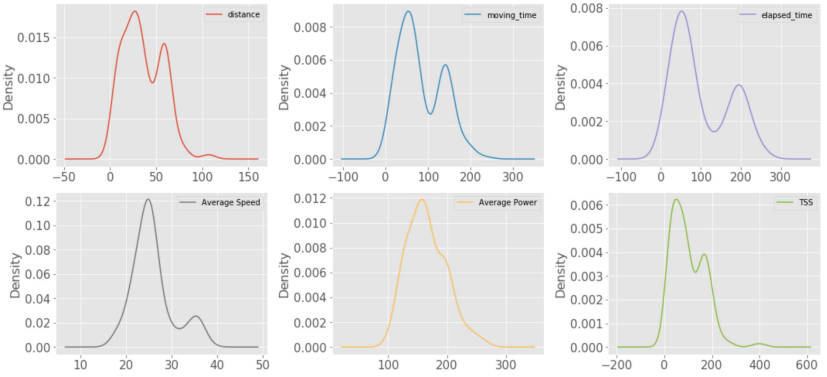
A. Histograms

In [13]:
```
## Histograms

# Selecting requied columns
new_dataframe = strava_plus_cheetah[["distance", "moving_time", "elapsed_time" ,  "Average Speed",
                                     "Average Power", "TSS"]]

# Plotting a histogram
new_dataframe.hist(figsize = (10, 10), edgecolor = "black", layout = (3, 3))
plt.show()

## We are only interesed in distributions of the variables that's why labels and units are not displayed on
the x-axis.
```



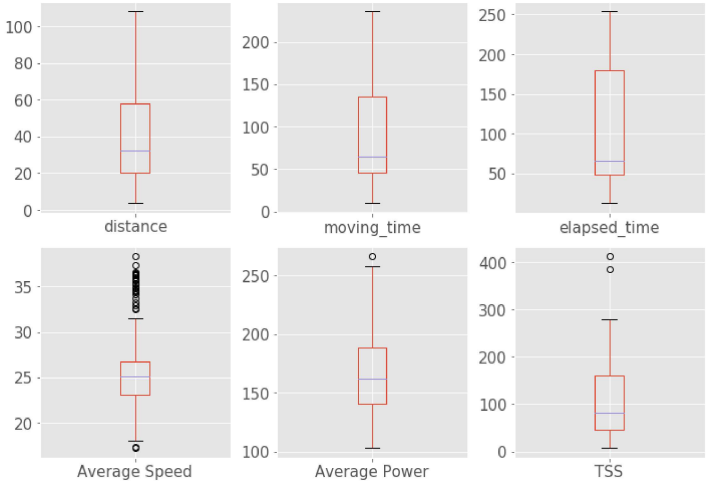
B. Density plots

In [14]:
```
## Density plot shows density curves. It can be useful to easily identify the distribution of the variable
s.

# Plotting density curves
new_dataframe.plot(kind='density', subplots = True, layout = (3, 3),
                   figsize = (15, 10), sharex = False, sharey = False)

# Spaces between subplots
plt.tight_layout()

plt.show()

## We are only interesed in distributions of the variables that's why labels and units are not displayed on
the x-axis.
```



C. Boxplots

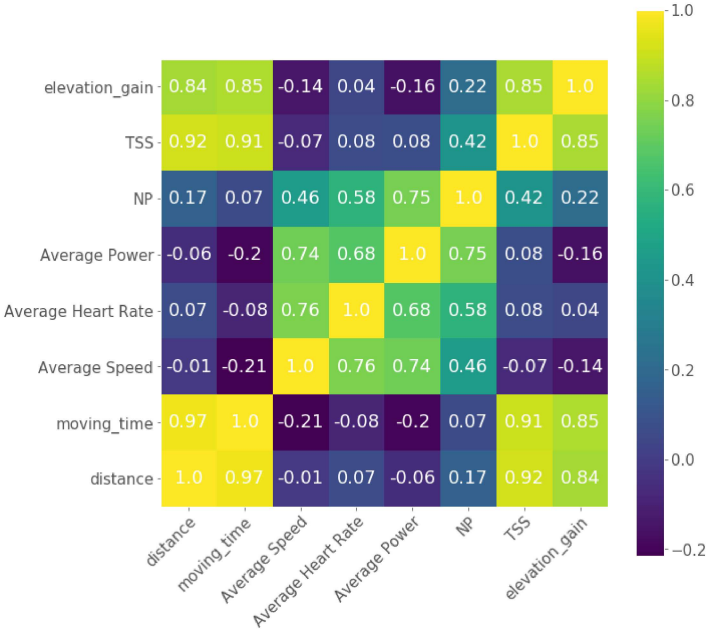
In [15]:
```
## Boxplots are also useful to identify how much variation a variable has.

# Plotting the boxplots
new_dataframe.plot(kind='box', subplots = True, layout = (3, 3),
                   figsize = (10,10), sharex = False, sharey = False)

# Spaces between subplots
plt.tight_layout()

plt.show()

## We are only interesed in variation of the variables that's why labels and units are not displayed on the
y-axis.
```



3. Explore the relationships between the following variables. Are any of them corrolated with each other (do they vary together in a predictable way)? Can you explain any relationships you observe?

- Distance
- Moving Time
- Average Speed
- Heart Rate
- Power (watts)
- Normalised power (NP)
- Training Stress Score
- Elevation Gain

In [16]:
```
# Ignoring the simple warning for this cell
warnings.simplefilter(action = 'ignore')

# Removing rows where Avg heart rate = 0
strava_plus_cheetah = strava_plus_cheetah[strava_plus_cheetah["Average Heart Rate"] != 0.0]

# Getting required variables
variables_in_que3 = strava_plus_cheetah[["distance", "moving_time", "Average Speed", "Average Heart Rate",
                                         "Average Power","NP", "TSS", "elevation_gain"]]

# Cleaning elevation_gain
# For more detials about this function check used_function/custom_functions.py
variables_in_que3["elevation_gain"] = custfun.clean_elevation_gain(variables_in_que3["elevation_gain"])

# Creating correlation matrix
# For more detials about this function check used_function/custom_functions.py
custfun.crete_correlogram(df = variables_in_que3, size = (10, 10), xlab_rotation = 45)
```



Correlogram

- Correlogram - Shows correlation between multiple numerical variables.
- Correlation is a measure that determines the strength of linear relation between two numerical variables.
- The value for correlation ranges between -1 and 1.
- 0 correlation means variables are independent of each other.
- The closer the value of correlation between variables to -1 and 1 the stronger the relationship between them.
- Negative correlation between x and y means if the value of x increases the value of y decreases and vice versa.
- Positive correlation between x and y means if the value of x increases the value of y increases and vice versa.

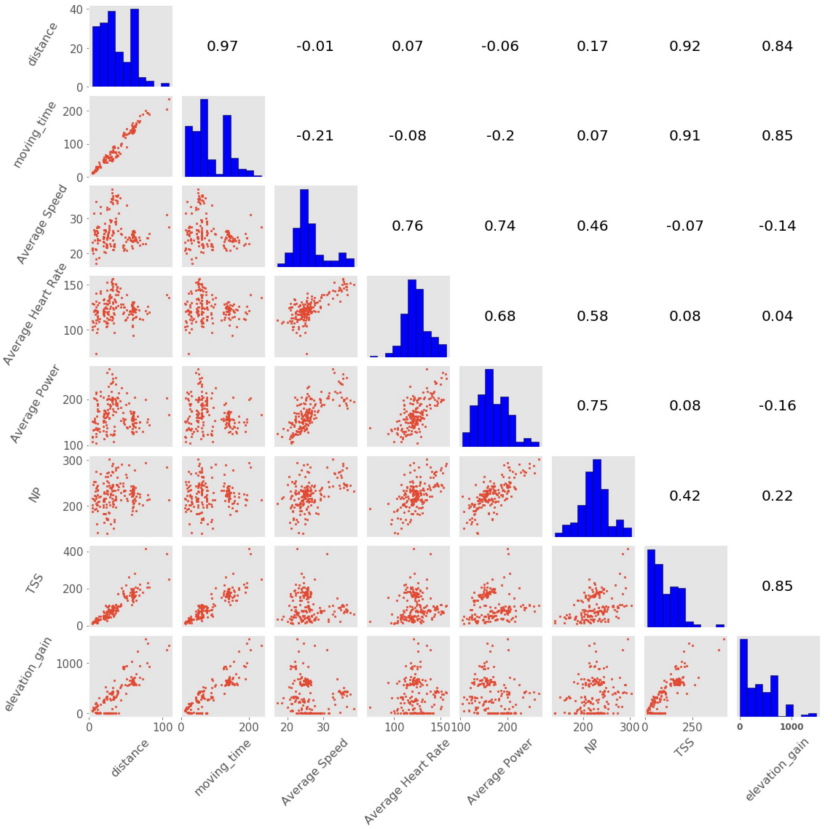
Graph Explanation

- Distance is strongly related to moving time(obvious), TSS and elevation gain(0.97, 0.92, 0.84). With the increase/decrease in Distance there will be strong increase/decrease in all three variables. And also it is weakly related to all other variables. We can also see that Distance is negatively weakly correlated with Average Speed. That makes sense as distance increases the speed will be sligly descreased. That also explains -0.06 correlation with average power(speed decreases, power generated by bike decreases).
- High correlation of distance with Elevation gain means the rider rides his/her bike where there are more hills(**This suggests that Australia has more** Undulate (https://english.stackexchange.com/questions/469106/what-do-you-call-a-road-that-goes-up-and-down) **roads)**. And due to hills rider's body is going to get under more stress due to riding up the hills(suggests more TSS).
- Moving time is also highly correlated with TSS and elevation gain(0.91, 0.85). More moving time -> more distance -> more stress on body(due to travelling long distances in hill areas).
- Average Speed is highly correlated with Average Power and NP(0.76, 0.74) for obvious reasons, moderately correlated Average Heart Rate(0.46). More speed -> more heart rate -> more power generated by bike. Average speed is weakly negatively correlated with moving time. It is obvious that if speed is increased rider needs less time to cover the distance.
- NP is just smoothened power so it is believable that it is strongly correlated(0.75) with Average Power. And it will aslo be highly correlated with variables which were highly correlated with Average Power(**confounding**).
- High correlation between elevation gain and moving time can be explained as more the ascent height, less the speed and more moving time.

Other ways to observe any relation between variable

1. Scatterplot matrix

In [17]:
```
# For more detials about this function check used_function/custom_functions.py
custfun.create_scatter_matrix(variables_in_que3, xlab_rotation = 45, ylab_rotation = 60,
                              xticklab_fontsize = 12, yticklab_fontsize = 12)
```



Graph Explanation

- You can see how two variables are varying(increasing/decreasing) together by looking at their scatter plot. Let's take an example of **distance** and **moving time.** By looking at the *first row second column* we can see that for high value of distance moving time is also high. The points are not scattered vertically much - this suggests that they are highly correlated. *You can observe other graphs just like that. The diagonal represents the histogram of respective variable*

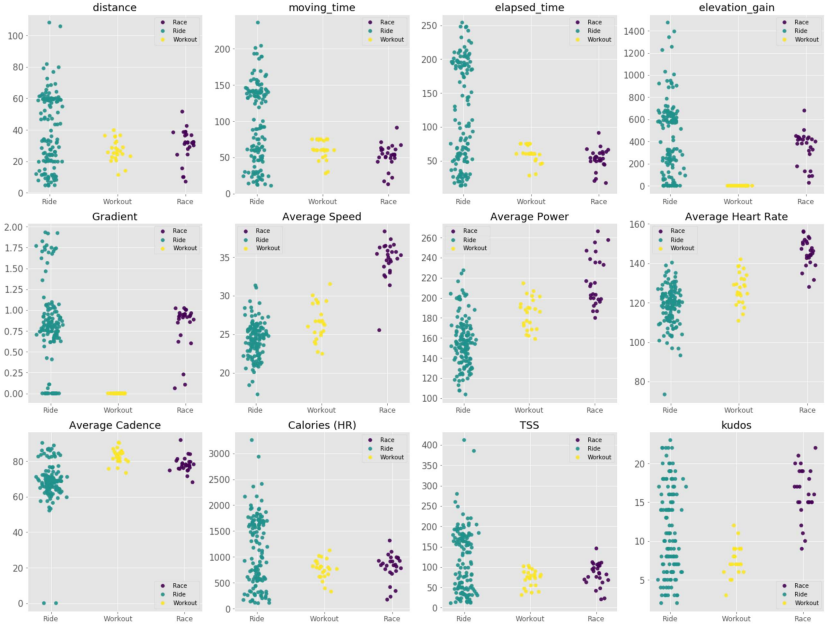
4. We want to explore the differences between the three categories: Race, Workout and Ride.

0. Some operations on dataframe before plotting the graphs

In [18]:
```
# Selecting important variables
important_dataframe = strava_plus_cheetah[["distance", "moving_time", "elapsed_time",
                                           "elevation_gain", "Gradient", "Average Speed", "Average Power",
"Average Heart Rate",
                                           "Average Cadence", "Calories (HR)", "TSS", "kudos", "workout_typ
e"]]

# Removing rows where heart rate is 0
important_dataframe = important_dataframe[important_dataframe["Average Heart Rate"] != 0]

# Cleaning elevation_gain
# For more detials about this function check used_function/custom_functions.py
important_dataframe["elevation_gain"] = custfun.clean_elevation_gain(important_dataframe["elevation_gain"])
# -See documentation above



def process_workout_type():
    """
    Function that will create new list based on workout_type.
    New list will be containing integers 1, 2 and 3.
    where,
    1 : Ride
    2 : Workout
    3 : Race
    This is done to help plot graphs based on category.
    """
    temp_list = []
    for i in important_dataframe["workout_type"]:
        if i == "Ride":
            temp_list.append(1)
        elif i == "Workout":
            temp_list.append(2)
        else:
            temp_list.append(3)

    return temp_list

important_dataframe["workout_type_int"] = process_workout_type()
```

A. Use scatter plots with different colours for each category to explore how these categories differ.

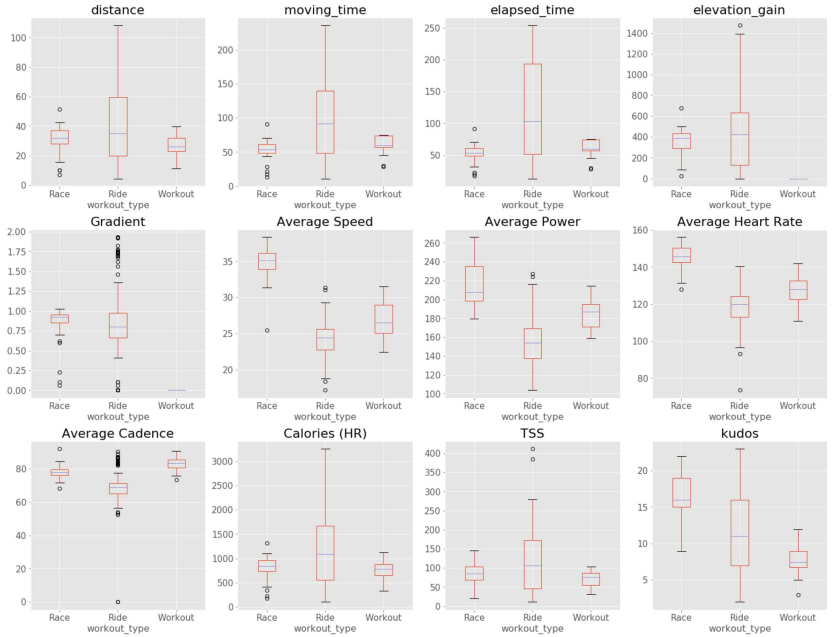
In [19]:
```
# For more detials about this function check used_function/custom_functions.py
custfun.plots_with_workout(df = important_dataframe, kind = "scatter")
```



Graph Explanation

- **Distance, Moving time, Elapsed time -** We can see that generally distance for *Race* and *Workout* is lower than distance for general *Ride*. Due to distance moving time and elapsed time are also low for *Race* and *Workout*.
- **Elevation gain, Gradient -** Elevation gain for *Workout* is 0(Obvious - stationary equipment). Elevation gain for *Ride* is high compare to *Race*. (Maybe race tracks have even ground[not much up and down roads]). Elevation gain and Gradient has the same distribution across workout types.
- **Average Speed, Average Heart Rate, Average Power -** Avg speed for *Race* is the highest(Obvious - Race has limited distance track, and contenstants are riding their bike fast in order to win), followed by *Workout*(due to less distance) and the least speed for *Ride*(Obvious - casual rides, no competition, more distance). **More Speed -> More Heart Rate -> More Power.**
- **Average Cadence -** Cadence is speed at which rider turns the pedal. It is strange that it is simillar across various workout types.
- **Calories, TSS -** These two variables show same relation to workout types as Distance. **More Distance -> More Calories burnt -> More Stress on Body**
- **Kudos -** More kudos to *Race* and *Ride* compare to *Workout*. We can't be sure of it. Maybe it is because of how kudos are higly correlated with distance.

In [20]:
```
# For more detials about this function check used_function/custom_functions.py
custfun.plots_with_workout(df = important_dataframe, kind = "box")
```



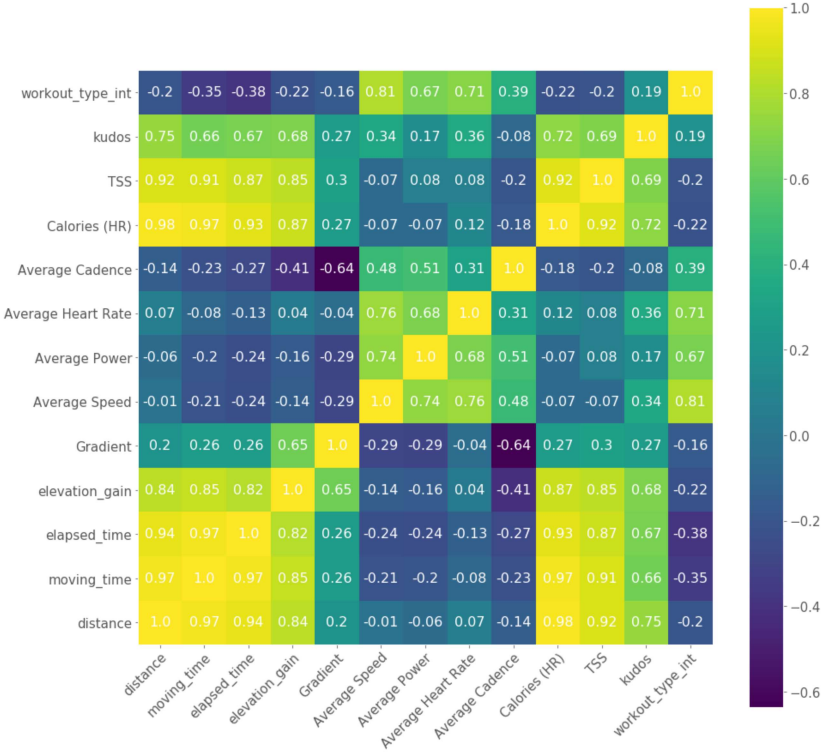
Graph Explanation

- **Distance, Moving time, Elapsed time -** We can see that generally boxplot of distance for *Race* and *Workout* is simillar. While *Ride* shows high variation in all of three variables. (Similar outlier points in all three variables)
- **Elevation gain, Gradient -** We can see that boxplots of Elevation gain and Gradient for *Race* and *Workout* is simillar. But for *Ride*, Elevation gain has high variation compare to Gradient(More outliers in Gradient for *Ride*).
- **Average Speed, Average Heart Rate, Average Power -** Variation in the categories *Race*, *Ride* and *Workout* in Average Speed is almost same. This also applies to Average Heart Rate and Average Power.
- **Average Cadence** - It is strange that it is simillar across various workout types. Though *Ride* has more outliers.
- **Calories, TSS -** We can see that boxplots of Calories and TSS for *Race* and *Workout* is simillar. But for *Ride*, Calories has high variation compare to Gradient(More outliers in Gradient for *Ride*).
- **Kudos -** More kudos to *Race* and *Ride* compare to *Workout*. But more variation in *Ride* compare to other two categories.
- *We have seen all the variables, We noticed that **Ride** has more variation to it. Maybe it is because people are generally attending more Ride compare to other categories(Count of workout type **Ride** is higher compare to other categories) and also people are not paying more attention to things when they are just casually riding the bike. That explains more variations.

Challenge 1

What leads to more kudos? Is there anything to indicate which rides are more popular? Explore the relationship between the main variables and kudos. Show a plot and comment on any relationship you observe.

In [21]:
```
# Deleting the workout_type column
del important_dataframe["workout_type"]

# For more detials about this function check used_function/custom_functions.py
custfun.crete_correlogram(df = important_dataframe, size = (15, 15), corr_font_size = 16, xlab_rotation = 45)
```



Graph Explanation

- When we look at the kudos column we find that the correlation between **kudos** and **distance** is fairly strong(0.75). Now there are other variables which are highly correlated with kudos like **moving_time, elapsed_time, elevation_gain, Calories, TSS.** But we **cant** confidently make a **statement** here that **high numbers for these variables means more kudos.** Because distance has **strong correlation** with all those variable, So it might be a case of **confounding.**
- **Gradient, Average Speed, Average Power, Average Heart Rate** has weak correlation with Kudos.

Challenge 2

Generate a plot that summarises the number of km ridden each month over the period of the data. Overlay this with the sum of the Training Stress Score and the average of the Average Speed to generate an overall summary of activity.

In [22]:
```
# Creating a copy for a dataframe
temp = strava_plus_cheetah.copy()

# Creating a new column called date_new using index column of strava_plus_cheetah
temp["date_new"] = pd.to_datetime(strava_plus_cheetah.index)

# Grouping distance, TSS and Average speed by month
per_month_distance = temp.set_index('date_new').groupby(pd.Grouper(freq = 'M'))['distance'].sum()
per_month_tss = temp.set_index('date_new').groupby(pd.Grouper(freq = 'M'))['TSS'].sum()
per_month_avg_avg_speed = temp.set_index('date_new').groupby(pd.Grouper(freq = 'M'))['Average Speed'].mean
()

# Creating a new DataFrame
per_month = pd.DataFrame()
per_month["Date"] = per_month_distance.index
per_month["Distance"] = per_month_distance.values
per_month["TSS"] = per_month_tss.values
per_month["Avg Avg Speed"] = per_month_avg_avg_speed.values

# Rounding up whole dataframe to 2 digits
per_month = per_month.round(2)

# Removing the timezone detail
per_month["Date"] = per_month['Date'].dt.tz_localize(None)
```

Summary of distance, TSS and Average of Average Speed by months

In [23]:
```
per_month.iloc[:, :]
```

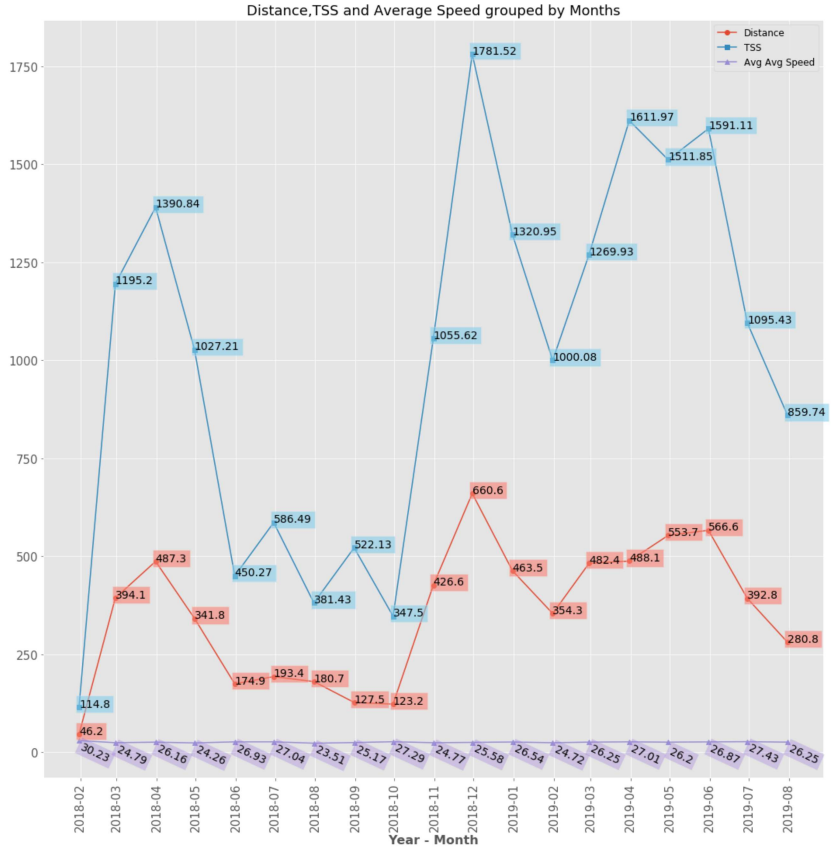
Out[23]:

	Date	Distance	TSS	Avg Avg Speed
0	2018-01-31	46.2	114.80	30.23
1	2018-02-28	394.1	1195.20	24.79
2	2018-03-31	487.3	1390.84	26.16
3	2018-04-30	341.8	1027.21	24.26
4	2018-05-31	174.9	450.27	26.93
5	2018-06-30	193.4	586.49	27.04
6	2018-07-31	180.7	381.43	23.51
7	2018-08-31	127.5	522.13	25.17
8	2018-09-30	123.2	347.50	27.29
9	2018-10-31	426.6	1055.62	24.77
10	2018-11-30	660.6	1781.52	25.58
11	2018-12-31	463.5	1320.95	26.54
12	2019-01-31	354.3	1000.08	24.72
13	2019-02-28	482.4	1269.93	26.25
14	2019-03-31	488.1	1611.97	27.01
15	2019-04-30	553.7	1511.85	26.20
16	2019-05-31	566.6	1591.11	26.87
17	2019-06-30	392.8	1095.43	27.43
18	2019-07-31	280.8	859.74	26.25

Plot Distance, TSS score and Average Speed grouped months

In [24]:

```
# Plotting an empty figure
plt.figure(figsize = (18, 18))

# Plotting three lines
p1, = plt.plot(per_month["Date"], per_month["Distance"], marker = 'o', label = "Distance" )
p2, = plt.plot(per_month["Date"], per_month["TSS"], marker = 's', label = "TSS")
p3, = plt.plot(per_month["Date"], per_month["Avg Avg Speed"], marker = '^', label = "Avg Avg Speed")

# Setting xLabel and title of the plot
plt.xlabel("Year - Month", fontsize = 16, fontweight = "bold")
plt.title("Distance,TSS and Average Speed grouped by Months", fontsize = 18)

# Setting x-axis locator, formatter and xticklabels
plt.gca().xaxis.set_major_locator(mdates.MonthLocator())
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m'))
plt.setp(plt.gca().get_xticklabels(),rotation = 90)

# Plotting Legends for Lines
lines = [p1, p2, p3]
plt.legend(lines, [l.get_label() for l in lines], fontsize = 12)

# Plotting values for every data point
for i in range(per_month.shape[0]):

    plt.text(per_month["Date"][i], per_month["Distance"][i], str(per_month["Distance"][i]),
             fontsize = 14, bbox = dict(facecolor = 'salmon', alpha = 0.6))

    plt.text(per_month["Date"][i], per_month["TSS"][i], str(per_month["TSS"][i]),
             fontsize = 14, bbox = dict(facecolor = 'skyblue', alpha = 0.6))

    # -55 -> just so that value label won't overlap with the other elements
    plt.text(per_month["Date"][i], per_month["Avg Avg Speed"][i] - 55, str(per_month["Avg Avg Speed"][i]),
             fontsize = 14, bbox = dict(facecolor = 'mediumpurple', alpha = 0.3), rotation = 335)

plt.show()
```


Distance,TSS and Average Speed grouped by Months
Distance
TSS
Avg Avg Speed
1750
1500
1250
1000
750
500
250
0
1781.52
1611.97
1591.11
1511.85
1390.84
1320.95
1269.93
1195.2
1095.43
1055.62
1027.21
1000.08
859.74
660.6
586.49
566.6
553.7
522.13
487.3
488.1
482.4
463.5
450.27
426.6
394.1
392.8
381.43
354.3
347.5
341.8
280.8
193.4
180.7
174.9
127.5
123.2
114.8
46.2
30.23
24.79
26.16
24.26
26.93
27.04
23.51
25.17
27.29
24.77
25.58
26.54
24.72
26.25
27.01
26.2
26.87
27.43
26.25
2018-02
2018-03
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
2018-10
2018-11
2018-12
2019-01
2019-02
2019-03
2019-04
2019-05
2019-06
2019-07
2019-08
Year - Month

OR

In [25]:
```
# For more detials about this function check used_function/custom_functions.py
custfun.create_lineplot_multiple_YAXIS(per_month, "Date", "Distance", "TSS", "Avg Avg Speed",
                                       "Year-Month", "Km", "score", "Km/h", c1 = "red", c2 = "royalblue", c
3 = "green")
```

Graph Explanation

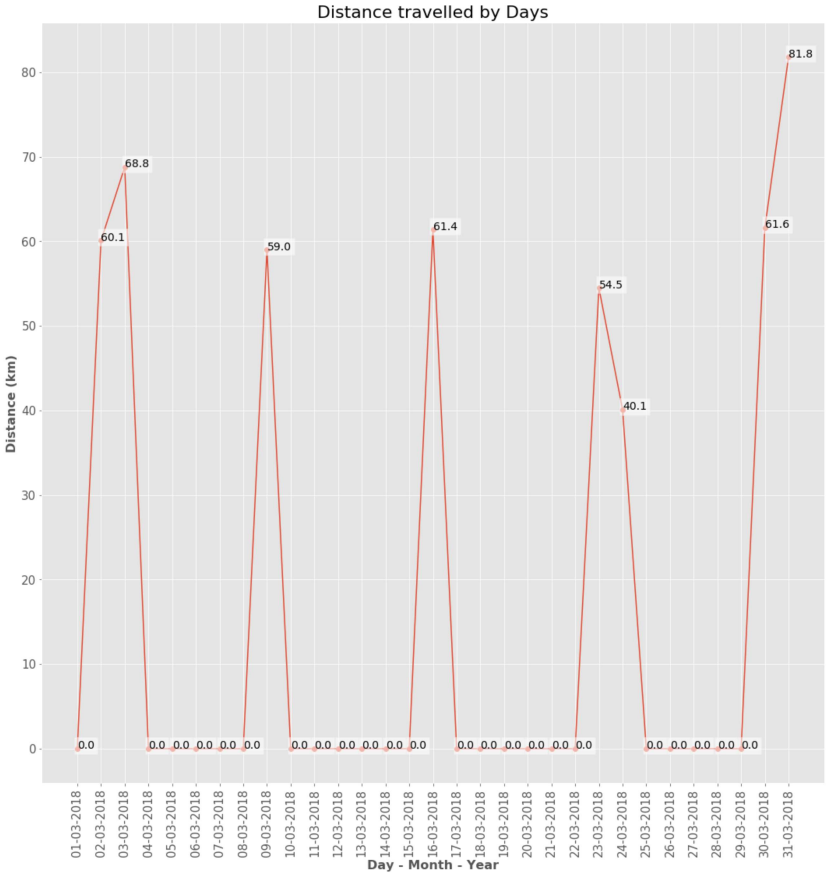
- Minimum total distance travelled was **46.2km in Febuary of 2018.**
- Maximum total distance travelled was **660.6km in December of 2018.**
- Minimum TSS score was **114.8 in Febuary of 2018.**
- Maximum TSS score was **1781.52 in December of 2018.**
- Minimum Average Average Speed was **30.2km/h in Febuary of 2018.**
- Maximum Average Average Speed was **23.51km/h in August of 2018.**
- We can observe following patterns here:-
  - 1. Line graph for TSS and Distance are almost the same.
  - 1. Average Average Speed is in inverse relationship with Distance(TSS).
  - 1. After **Febuary** of both year(2018 and 2019) total distance travelled grouped by months and TSS score is **increasing** up to **May** of both years.
  - 1. After **May** of both year(2018 and 2019) total distance travelled by grouped by months and TSS score is **decreasing** up to **August.**
  - 1. There is no observable pattern in Average Average Speed.

Challenge 3

Generate a similar graph but one that shows the activity over a given month, with the sum of the values for each day of the month shown. So, if there are two rides on a given day, the graph should show the sum of the distances etc for these rides.

In [26]:
```
# Creating a copy for a dataframe
temp = strava_plus_cheetah.copy()

# Creating a new column called date_new using index column of strava_plus_cheetah
temp["date_new"] = pd.to_datetime(strava_plus_cheetah.index)

# Grouping the distance by day
per_day_distance = temp.set_index('date_new').groupby(pd.Grouper(freq = 'D'))['distance'].sum()

# Creating a new dataframe
per_day = pd.DataFrame()

# Creating columns in new dataframe
per_day["date"] = per_day_distance.index
per_day["date"] = pd.to_datetime(per_day["date"])
per_day["distance"] = per_day_distance.values

# Rounding up whole dataframe to 2 digits
per_day = per_day.round(2)
```

In [27]:
```
# Distance travelled over given month

## Try it yourself
month = 3

## For more detials about this function check used_function/custom_functions.py
custfun.activityOverGivenMonth(per_day, 2018, month)
```



Portfolio 2 - Reproducing the existing work

In [28]:
```
# Loading the libraries
import sys
sys.path.append("used_functions/")
import custom_functions as custfun
import importlib
importlib.reload(custfun)
import pandas as pd
from pandas.plotting import scatter_matrix
import numpy as np
from matplotlib import pyplot as plt
plt.style.use("ggplot")
import matplotlib.dates as mdates
import seaborn as sns
import matplotlib as mpl
from sklearn import linear_model
from sklearn.model_selection import KFold, RepeatedKFold, cross_val_score
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.feature_selection import RFE
```

In [29]:
```
# Reading the data
training = pd.read_csv("Portfolio2/training.csv")
testing = pd.read_csv("Portfolio2/testing.csv")
training["date"] = pd.to_datetime(training["date"])
testing["date"] = pd.to_datetime(testing["date"])
```

1. EXPLORARION

In [30]:
```
print(training.shape)
print(testing.shape)
```

```
(14803, 32)
(4932, 32)
```

In [31]:
```
# % data in training
print("% of data in training dataset : ", str(round(training.shape[0]/(training.shape[0] + testing.shape[0
]) * 100))+"%")
# % data in testing
print("% of data in testing dataset : ", str(round(testing.shape[0]/(training.shape[0] + testing.shape[0])
* 100))+"%")
```

```
% of data in training dataset :  75%
% of data in testing dataset :  25%
```

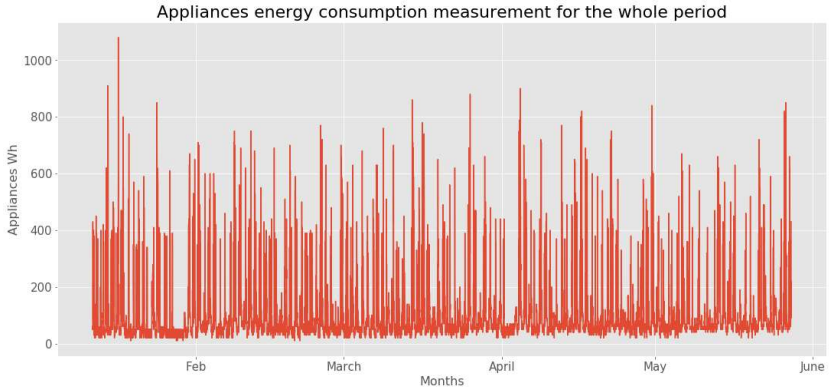
Data is partitioned in two parts [training->75\% and testing 25\%] as mentioned in the paper.

Setting matplotlib parameters

In [32]:
```
mpl.rcParams['axes.titlesize'] = 22                                    # -Change title size of
a plot
mpl.rcParams['axes.labelsize'] = 16                                    # -Change label size(x
 and y) of a plot
mpl.rcParams['xtick.labelsize'] = 15                                   # -Change xticks size o
f a plot
mpl.rcParams['ytick.labelsize'] = 15                                   # -Change yticks size o
f a plot
```

Graph 1

In [33]:
```
# Plotting the figure
plt.figure(figsize=(18,8))

# Adding the line plot to figure
plt.plot(training["date"], training["Appliances"])

# Setting plot parameters like xtick locators and labels
plt.gca().xaxis.set_major_locator(mdates.MonthLocator())
plt.gca().set_xticklabels(["Feb","March", "April", "May", "June"])
plt.setp(plt.gca().get_xticklabels(),rotation = 0)

# Setting x-axis, y-axis labels and title of the plot
plt.xlabel("Months")
plt.ylabel("Appliances Wh")
plt.title("Appliances energy consumption measurement for the whole period")

plt.show()
```

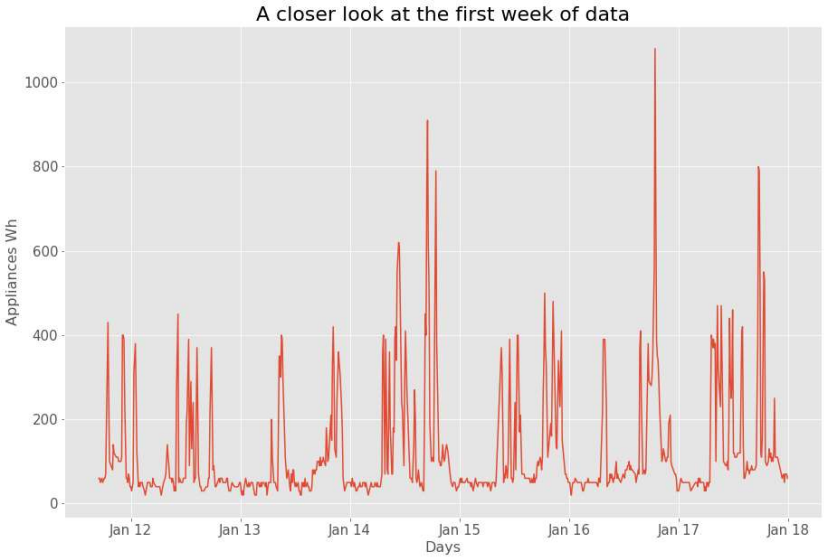


Graph Explanation

We can observe the Appliances' consumption is following some kind pattern. Like before the begining of February it show low consuption then variable consuption till April, after April it again shows low consumption and then pattern follows.

Graph 2

In [34]:
```
# Mapping string_to_int function to get weeks
data_weeks = pd.Series(map(custfun.string_to_int, training["date"].dt.strftime('%W')))

# Finding from which week data is starting
min_week = min(data_weeks)

# Filtering data of where week = min_week
flag_week1 = (data_weeks == min_week)
first_week = training.loc[flag_week1]

# Plotting the figure
plt.figure(figsize=(15,10))

# Adding the line plot to figure
plt.plot(first_week["date"], first_week["Appliances"])

# Setting plot parameters like xtick locators and labels
plt.gca().xaxis.set_major_locator(mdates.DayLocator())
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('Jan %d'))
plt.setp(plt.gca().get_xticklabels(),rotation = 0)

# Setting x-axis, y-axis labels and title of the plot
plt.xlabel("Days")
plt.ylabel("Appliances Wh")
plt.title("A closer look at the first week of data")

plt.show()
```

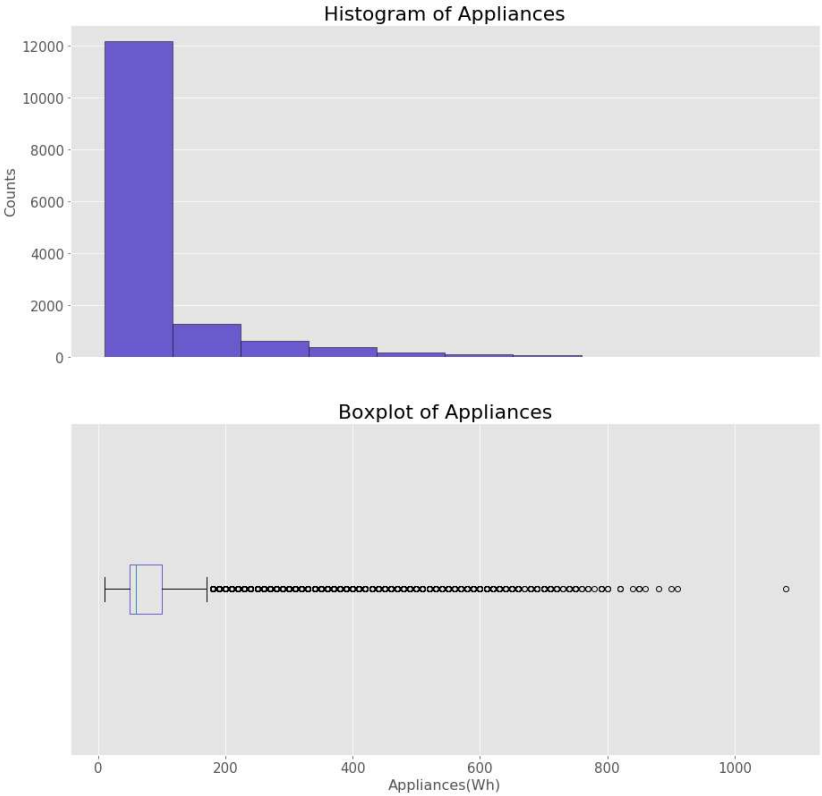


Graph Explanation

The appliances' consumption profile is highly variable as seen in above plot, with periods of almost constant demand followed by high spikes.

Graph 3

In [35]:
```
# For more detials about this function check used_function/custom_functions.py
custfun.create_histogram_plus_boxplot(training["Appliances"], "Appliances", "slateblue", "Counts", "Appliances(Wh)", (15,15))
```

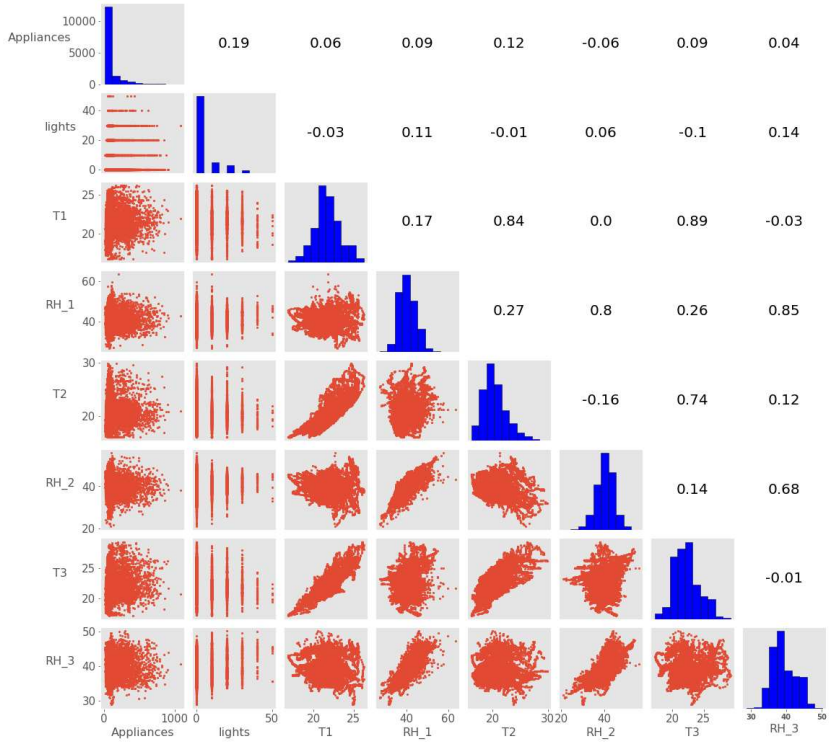


Graph Explanation

Above plot shows histogram and boxplot of the data. As can be seen the data distribution has a long tail. In the boxplot, the median is represented with a thick black line inside the blue rectangle, and has a value of 60Wh. The lower whisker has a value of 10Wh and the upper whisker has a value of 170Wh. It also shows that the data above the median is more dispersed and that there are several outliers (marked withthe round circles above the upper whisker)

Graph 4

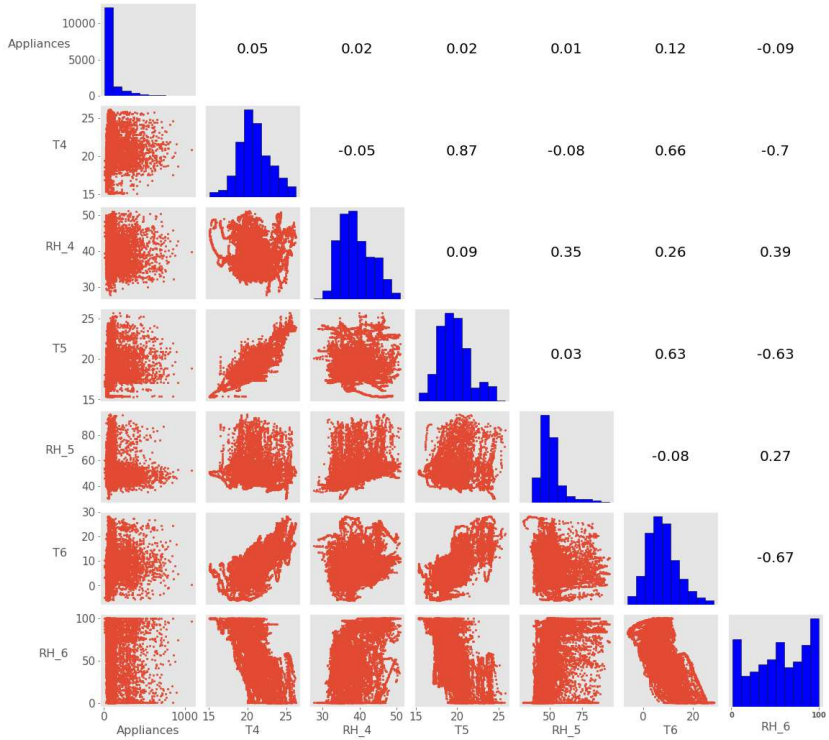
In [36]:
```
# For more detials about this function check used_function/custom_functions.py
custfun.create_scatter_matrix(training[["Appliances", "lights", "T1", "RH_1", "T2", "RH_2", "T3", "RH_3"]])
```



Graph Explanation

Plot shows that there is a positive correlation between the energy consumption of appliances and lights (0.19). The second largest correlation is between appliances and T2. For the indoor temperatures, the correlations are high as expected. For example, a positive correlation is found with T1 and T3.

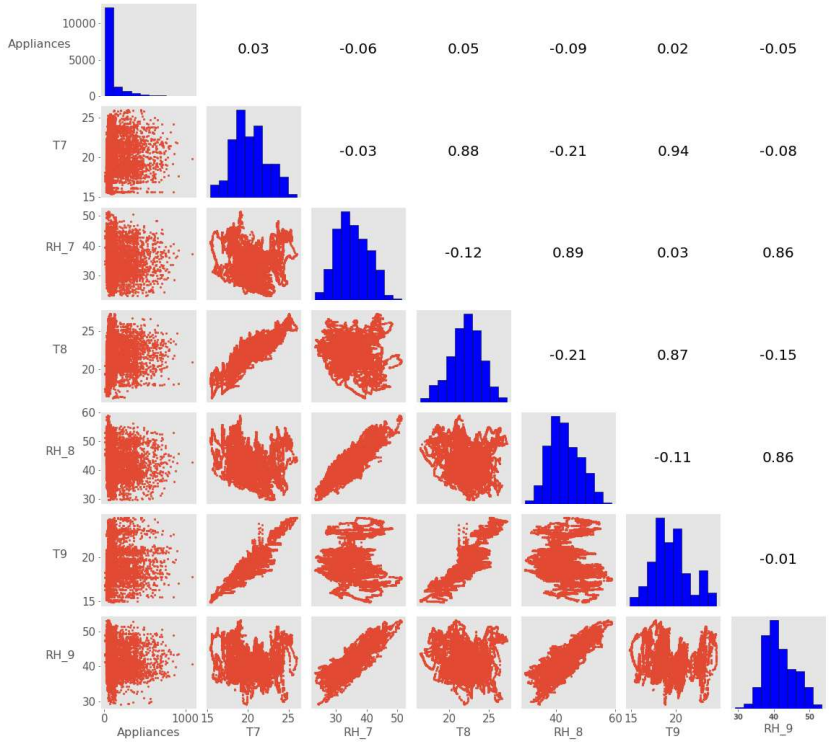
In [37]:
```
# For more detials about this function check used_function/custom_functions.py
custfun.create_scatter_matrix(training[["Appliances", "T4", "RH_4", "T5", "RH_5", "T6", "RH_6"]])
```



Graph Explanation

Plot shows that the highest correlation with the appliances is between the outdoor temperature (0.12). There is also a negative correlation between the appliances and outdoor humidity/RH6 (−0.09)

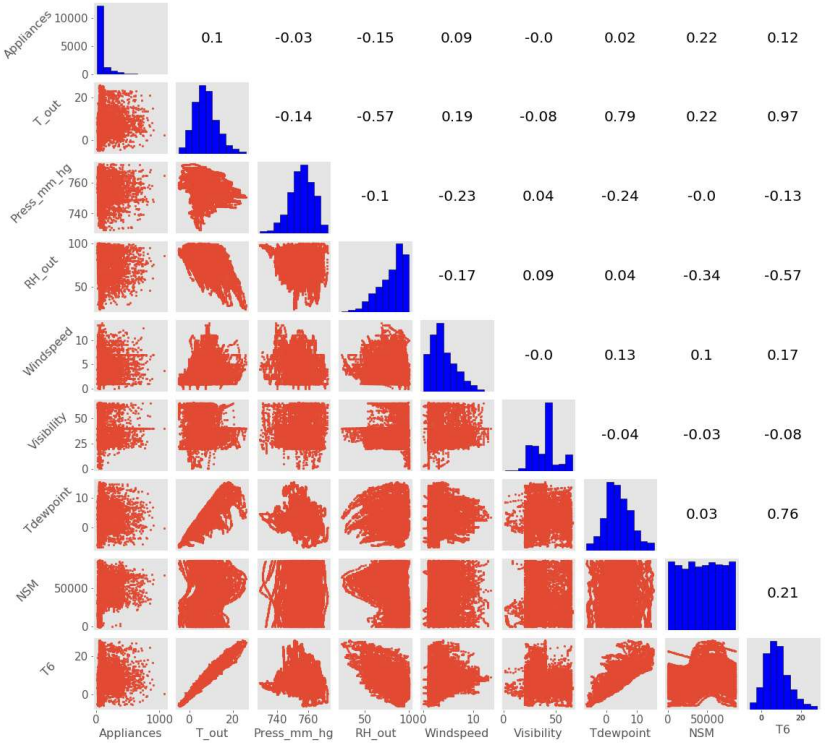
In [38]:
```
# For more detials about this function check used_function/custom_functions.py
custfun.create_scatter_matrix(training[["Appliances", "T7", "RH_7", "T8", "RH_8", "T9", "RH_9"]])
```



Graph Explanation

Plot shows positive correlations between the consumption of appliances and T7, T8 and T9 being 0.03, 0.05 and 0.02 respectively and negative correlation with RH_7(-0.06), RH_8(-0.09), RH_9(-0.05). The consumption of appliances have postive correlations with temperatures and negative correlation with Humidities.

In [39]:
```
# For more detials about this function check used_function/custom_functions.py
custfun.create_scatter_matrix(training[["Appliances", "T_out", "Press_mm_hg", "RH_out", "Windspeed",
       "Visibility", "Tdewpoint", "NSM", "T6"]], ylab_rotation = 45)
```



Graph Exaplanation

Plot shows the highest correlation between the energy consumption of appliances and NSM with a value of 0.22. A positive correlation of 0.10 is seen between appliances' consumption and outdoor temperature (Tout) that is, the higher temperatures, the higher the energy use by the appliances. Also there is a positive correlation with appliances' consumption and wind speed (0.09), higher wind speeds correlate with higher energy consumption by the appliances. A negative correlation of −0.15 was found with the RHout, and of −0.03 with pressure. Another important and interesting correlation is between the pressure and the wind speed. This relationship is negative (−0.23). The linear trend is with lower pressure the wind speed will be higher.

In [40]:
```
# Getting different dataframes for first four weeks from the data
## For more detials about this function check used_function/custom_functions.py
first_week_appliances_df = custfun.create_weekdata(training, 1)
second_week_appliances_df = custfun.create_weekdata(training, 2)
third_week_appliances_df = custfun.create_weekdata(training, 3)
forth_week_appliances_df = custfun.create_weekdata(training, 4)

# We don't have data for first week in January. Data starts from Monday from 17th hour.
first_week_appliances_df
```

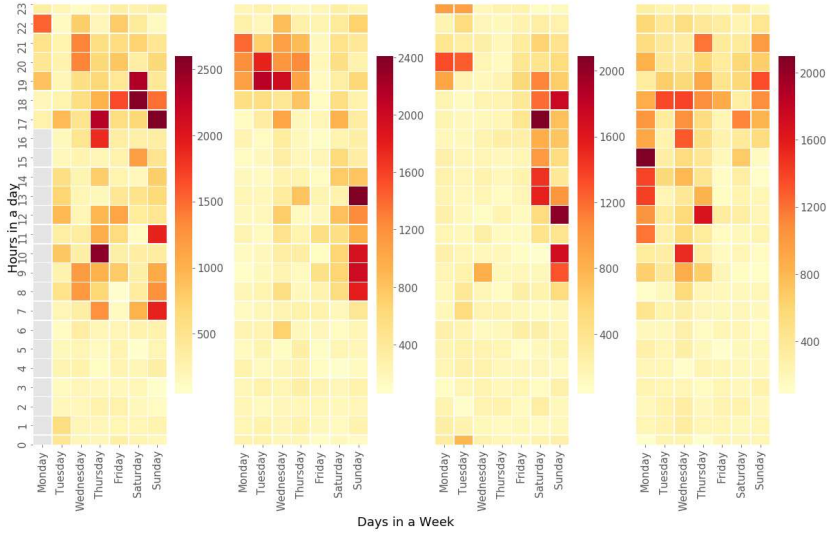
Out[40]:

date	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
date							
0	NaN	470.0	190.0	190.0	320.0	240.0	180.0
1	NaN	590.0	190.0	190.0	210.0	210.0	250.0
2	NaN	180.0	180.0	260.0	280.0	150.0	120.0
3	NaN	140.0	190.0	180.0	180.0	200.0	50.0
4	NaN	230.0	270.0	110.0	240.0	210.0	310.0
5	NaN	170.0	210.0	130.0	240.0	50.0	220.0
6	NaN	120.0	350.0	210.0	220.0	260.0	270.0
7	NaN	220.0	330.0	1290.0	140.0	1000.0	1920.0
8	NaN	520.0	1210.0	690.0	50.0	380.0	1270.0
9	NaN	260.0	1190.0	1010.0	800.0	310.0	1060.0
10	NaN	820.0	310.0	2540.0	310.0	350.0	360.0
11	NaN	330.0	360.0	1040.0	620.0	120.0	1910.0
12	NaN	950.0	210.0	960.0	1120.0	390.0	460.0
13	NaN	710.0	210.0	170.0	450.0	520.0	650.0
14	NaN	560.0	230.0	780.0	250.0	180.0	760.0
15	NaN	170.0	250.0	210.0	280.0	1180.0	490.0
16	NaN	150.0	480.0	1850.0	340.0	300.0	330.0
17	280.0	930.0	490.0	2320.0	590.0	670.0	2600.0
18	190.0	250.0	580.0	1000.0	1670.0	2570.0	1470.0
19	870.0	220.0	560.0	670.0	440.0	2350.0	420.0
20	450.0	260.0	1360.0	660.0	830.0	340.0	660.0
21	520.0	250.0	1360.0	570.0	610.0	720.0	470.0
22	1540.0	130.0	760.0	190.0	790.0	400.0	130.0
23	340.0	210.0	100.0	200.0	310.0	260.0	320.0

I have printed above dataframe just to show that there is no data before hour 17 on first day of the data

Graph 5

In [41]:
```
# Plotting the figure
fig = plt.figure(figsize = (20,20))

# Adding four subplots - each for each week
ax1 = fig.add_subplot(141)
ax2 = fig.add_subplot(142)
ax3 = fig.add_subplot(143)
ax4 = fig.add_subplot(144)

# Plotting heatmap for each week
sns.heatmap(first_week_appliances_df.sort_index(ascending = False), cmap = "YlOrRd", ax = ax1,
            square = True, cbar_kws = {"shrink" : 0.45}, linewidths = 0.5)
sns.heatmap(second_week_appliances_df.sort_index(ascending = False), cmap = "YlOrRd", ax = ax2,
            square = True, cbar_kws = {"shrink" : 0.45}, linewidths = 0.5)
sns.heatmap(third_week_appliances_df.sort_index(ascending = False), cmap = "YlOrRd", ax = ax3,
            square = True, cbar_kws = {"shrink" : 0.45}, linewidths = 0.5)
sns.heatmap(forth_week_appliances_df.sort_index(ascending = False), cmap = "YlOrRd", ax = ax4,
            square = True, cbar_kws = {"shrink" : 0.45}, linewidths = 0.5)

# Removing labels and ticks from the axes
ax1.set_ylabel('');ax1.set_xlabel('')

ax2.set_ylabel('');ax2.set_xlabel('');ax2.set_yticks([]);ax2.set_yticklabels([])

ax3.set_ylabel('');ax3.set_xlabel('');ax3.set_yticks([]);ax3.set_yticklabels([])

ax4.set_ylabel('');ax4.set_xlabel('');ax4.set_yticks([]);ax4.set_yticklabels([])

# Setting x-axis and y-axis label for plot
fig.text(0.5, 0.20, "Days in a Week", fontsize = 18, ha = 'center')
fig.text(0.1, 0.5, "Hours in a day", fontsize = 18, va = 'center', rotation = 'vertical')

plt.show()
```



Graph Explanation

The energy consumption starts to rise around 6 in the morning. Then around noon, there are energy load surges. The energy demand also increases around 6 pm. There is no clear pattern regarding the day of the week

Grey box means we don't have data for that hour. I can replace this NAN values with zero but Author of the paper didn't mention that in the paper.

2. MODELING

In [42]:
```
# Taking all the columns as mentioned in the paper
training_modified_1 = training.copy()
training_modified_1 = training_modified_1.iloc[:, 1:]
training_modified_1["WeekStatus"] = training_modified_1["WeekStatus"].astype("category")
training_modified_1["Day_of_week"] = training_modified_1["Day_of_week"].astype("category")

# Getting categorical columns and convert them to codes
categorical_cols = training_modified_1.select_dtypes(['category']).columns
training_modified_1[categorical_cols] = training_modified_1[categorical_cols].apply(lambda x: x.cat.codes)

# Taking all the columns as mentioned in the paper
testing_modified_1 = testing.copy()
testing_modified_1 = testing_modified_1.iloc[:, 1:]
testing_modified_1["WeekStatus"] = testing_modified_1["WeekStatus"].astype("category")
testing_modified_1["Day_of_week"] = testing_modified_1["Day_of_week"].astype("category")

# Getting categorical columns and convert them to codes
categorical_cols = testing_modified_1.select_dtypes(['category']).columns
testing_modified_1[categorical_cols] = testing_modified_1[categorical_cols].apply(lambda x: x.cat.codes)

# Taking x(independent) and y(dependent) variables for models, train for training and test for testing
x_train = training_modified_1.iloc[:, 1:]
y_train = training_modified_1.iloc[:, 0]
x_test = testing_modified_1.iloc[:, 1:]
y_test = testing_modified_1.iloc[:, 0]
```

In [43]:
```
# Creating the linear model
paper_model = linear_model.LinearRegression()

# Fitting the training data
paper_model.fit(x_train, y_train)
```

Out[43]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

Model Performance for Training

In [44]:
```
# Predicting the values for training data
predicted_train = paper_model.predict(x_train)

# Calculating various metrics for evaluation purposes
rmse_train = mean_squared_error(y_train, predicted_train) ** 0.5
mae_train = mean_absolute_error(y_train, predicted_train)
## For more detials about this function check used_function/custom_functions.py
mape_train = custfun.mean_absolute_percentage_error(y_train, predicted_train)
r2_train = r2_score(y_train, predicted_train)

# Printing metrics given in paper
print("RMSE(Root Mean Squared Error):", round(rmse_train, 2))
print("R Squared:", round(r2_train, 2))
print("MAE(Mean Absolute Error):", round(mae_train, 2))
print("MAPE(Mean Absolute Percentage Error):", round(mape_train, 2))
```

```
RMSE(Root Mean Squared Error): 93.29
R Squared: 0.18
MAE(Mean Absolute Error): 53.31
MAPE(Mean Absolute Percentage Error): 61.73
```

Got same results as in paper for training data.

Model Performance for Testing

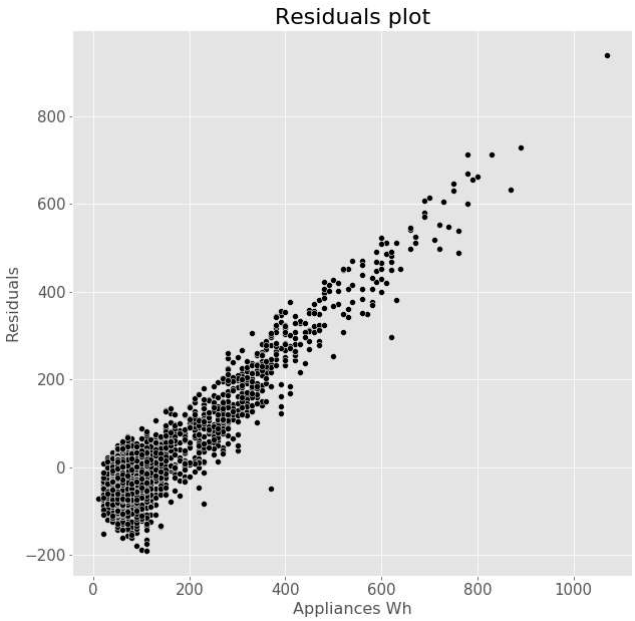
In [45]:
```
# Predicting the values for testing data
predicted_test = paper_model.predict(x_test)

# Calculating various metrix for evaluation purposes
rmse_test = mean_squared_error(y_test, predicted_test) ** 0.5
mae_test = mean_absolute_error(y_test, predicted_test)
## For more detials about this function check used_function/custom_functions.py
mape_test = custfun.mean_absolute_percentage_error(y_test, predicted_test)
r2_test = r2_score(y_test, predicted_test)

# Printing metrics
print("RMSE(Root Mean Squared Error):", round(rmse_test, 2))
print("R Squared:", round(r2_test, 2))
print("MAE(Mean Absolute Error):", round(mae_test, 2))
print("MAPE(Mean Absolute Percentage Error):", round(mape_test, 2))
```

```
RMSE(Root Mean Squared Error): 93.33
R Squared: 0.16
MAE(Mean Absolute Error): 52.16
MAPE(Mean Absolute Percentage Error): 60.32
```

Got same results as in paper for testing data.

Residuals plot

In [46]:
```
# Plotting the figure
plt.figure(figsize = (10,10))

# Adding the scatter plot of residuals vs original value
plt.scatter(y_test, (y_test - predicted_test), c = "black", edgecolors = "white", marker = "o")

# Setting x-axis, y-axis label and title of the plot
plt.xlabel("Appliances Wh")
plt.ylabel("Residuals")
plt.title("Residuals plot")

plt.show()
```



Graph Explanation

Plot shows a residual plot for the linear regression model. The residuals were computed as the difference between the real values and the predicted values. From plot above, it is obvious that the relationship between the variables and the energy consumption of appliances is not well represented by the linear model since the residuals are not normally distributed around the horizontal axis.

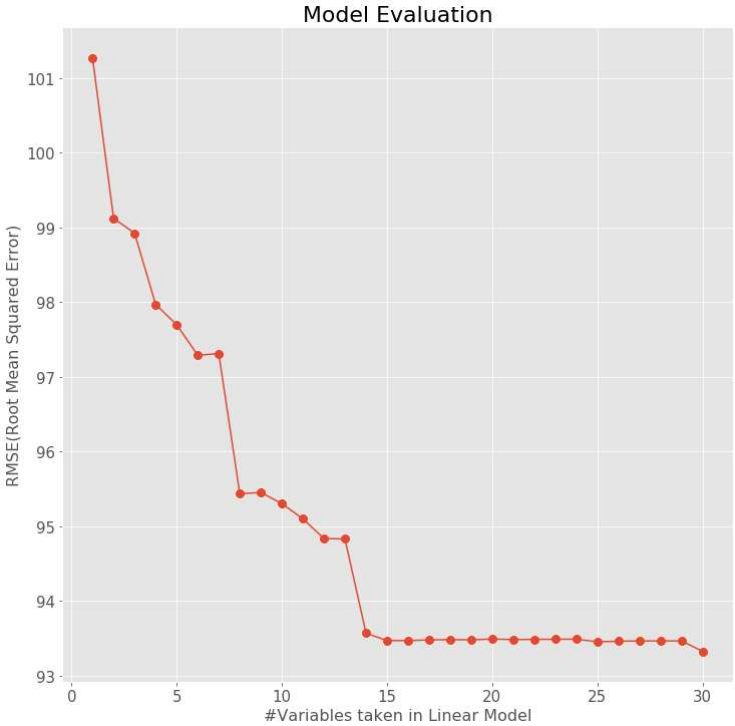
3. RFE

In [47]:
```
# Creating new dictionary for creating dataframe from it
dictionary = {}

# Some lists to create the columns in dataframe
variables_chosen, no_variables_chosen = [], []
rmse, mae, mape, r2 = [], [], [], []

# Looping over to get various #variable for RFE
for i in range(len(x_train.columns), 0, -1):

    # Creating the RFE
    rfe = RFE(paper_model, n_features_to_select = i, step = 1)

    # Fitting the RFE
    rfe_model = rfe.fit(x_train, y_train)

    # Predicting using rfe model
    predicted = rfe_model.predict(x_test)

    # Appending some information to the lists
    no_variables_chosen.append(i)
    variables_chosen.append(", ".join(x_train.columns.values[list(rfe_model.support_)]))
    rmse.append(mean_squared_error(y_test, predicted) ** 0.5)
    mae.append(mean_absolute_error(y_test, predicted))
    ## For more detials about this function check used_function/custom_functions.py
    mape.append(custfun.mean_absolute_percentage_error(y_test, predicted))
    r2.append(r2_score(y_test, predicted))

# Adding keys and values to dictionary
dictionary["variables_chosen"] = variables_chosen
dictionary["no_variables_chosen"] = no_variables_chosen
dictionary["rmse"] = rmse
dictionary["mae"] = mae
dictionary["mape"] = mape
dictionary["r2"] = r2
```

In [48]:
```
# Creating dataframe from the dictionary
model_information = pd.DataFrame.from_dict(dictionary)
```

In [49]:
```
# Sorting dataframe based on r2 values
model_information = model_information.sort_values("r2", ascending = False)
```

In [50]:
```
# Plotting RMSE vs No of variables
# For more detials about this function check used_function/custom_functions.py
custfun.plot_RMSE_VS_NoVar(df = model_information)
```



In [51]:
```
model_information.head()
```

Out[51]:

	variables_chosen	no_variables_chosen	rmse	mae	mape	r2
0	lights, T1, RH_1, T2, RH_2, T3, RH_3, T4, RH_4...	30	93.328781	52.157240	60.322326	0.156255
5	lights, T1, RH_1, T2, RH_2, T3, RH_3, T4, RH_4...	25	93.456502	52.270019	60.403636	0.153944
4	lights, T1, RH_1, T2, RH_2, T3, RH_3, T4, RH_4...	26	93.463714	52.242983	60.345830	0.153813
3	lights, T1, RH_1, T2, RH_2, T3, RH_3, T4, RH_4...	27	93.467029	52.209217	60.312623	0.153753
2	lights, T1, RH_1, T2, RH_2, T3, RH_3, T4, RH_4...	28	93.467705	52.222808	60.339592	0.153741

Experimenting with the Linear Model

Training data

In [52]:
```
# Creating new training dataframes for the experimental purposes
training_modified_2 = training.copy()

# Creating dummies instead of using categorical codes
dummies_weekstatus = pd.get_dummies(training_modified_2.loc[:, "WeekStatus"])
dummies_dow = pd.get_dummies(training_modified_2.loc[:, "Day_of_week"])

# Appending the dummies columns to dataframe
training_modified_2 = pd.concat([training_modified_2, dummies_weekstatus, dummies_dow], axis = 1)

# Dropping one dummy variable from each column as well as dropping categorical columns
training_modified_2 = training_modified_2.drop(["Weekend", "WeekStatus", "Day_of_week", "Monday"], axis = 1
)

# Creating new column for hour
training_modified_2["hour"] = training_modified_2["date"].dt.hour

training_modified_2.head()
```

Out[52]:

	date	Appliances	lights	T1	RH_1	T2	RH_2	T3	RH_3	T4	...	rv2	NSM	Weekday	Frida
0	2016-01-11 17:00:00	60	30	19.89	47.596667	19.2	44.790000	19.79	44.730000	19.000000	...	13.275433	61200	1	
1	2016-01-11 17:10:00	60	30	19.89	46.693333	19.2	44.722500	19.79	44.790000	19.000000	...	18.606195	61800	1	
2	2016-01-11 17:20:00	50	30	19.89	46.300000	19.2	44.626667	19.79	44.933333	18.926667	...	28.642668	62400	1	
3	2016-01-11 17:40:00	60	40	19.89	46.333333	19.2	44.530000	19.79	45.000000	18.890000	...	10.084097	63600	1	
4	2016-01-11 17:50:00	50	40	19.89	46.026667	19.2	44.500000	19.79	44.933333	18.890000	...	44.919484	64200	1	

5 rows × 38 columns

Testing data

In [53]:
```
# Creating new testing dataframes for the experimental purposes
testing_modified_2 = testing.copy()

# Creating dummies instead of using categorical codes
dummies_weekstatus = pd.get_dummies(testing_modified_2.loc[:, "WeekStatus"])
dummies_dow = pd.get_dummies(testing_modified_2.loc[:, "Day_of_week"])

# Appending the dummies columns to dataframe
testing_modified_2 = pd.concat([testing_modified_2, dummies_weekstatus, dummies_dow], axis = 1)

# Dropping one dummy variable from each column as well as dropping categorical columns
testing_modified_2 = testing_modified_2.drop(["Weekend", "WeekStatus", "Day_of_week", "Monday"], axis = 1)

# Creating new column for hour
testing_modified_2["hour"] = testing_modified_2["date"].dt.hour

testing_modified_2.head()
```

Out[53]:

	date	Appliances	lights	T1	RH_1	T2	RH_2	T3	RH_3	T4	...	rv2	NSM	Weekday
0	2016-01-11 17:30:00	50	40	19.890000	46.066667	19.200000	44.590000	19.79	45.000000	18.89	...	45.410389	63000	1
1	2016-01-11 18:00:00	60	50	19.890000	45.766667	19.200000	44.500000	19.79	44.900000	18.89	...	47.233763	64800	1
2	2016-01-11 18:40:00	230	70	19.926667	45.863333	19.356667	44.400000	19.79	44.900000	18.89	...	10.298729	67200	1
3	2016-01-11 18:50:00	580	60	20.066667	46.396667	19.426667	44.400000	19.79	44.826667	19.00	...	8.827838	67800	1
4	2016-01-11 19:30:00	100	10	20.566667	53.893333	20.033333	46.756667	20.10	48.466667	19.00	...	24.884962	70200	1

5 rows × 38 columns

Creating splits for the model

In [54]:
```
# Taking x(independent) and y(dependent) variables for models, train for training and test for testing
train_x = training_modified_2.iloc[:, 2:]
test_x = testing_modified_2.iloc[:, 2:]
train_y = training_modified_2.iloc[:, 1]
test_y = testing_modified_2.iloc[:, 1]
```

Modelling

In [55]:
```
# Creating linear regression model
tem_model = linear_model.LinearRegression()

# Fitting training data
tem_model.fit(train_x, train_y)
```

Out[55]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

Model Performance

In [56]:
```
# Predicting the values for testing data
predicted_test = tem_model.predict(test_x)

# Calculating various metrix for evaluation purposes
rmse_test = mean_squared_error(test_y, predicted_test) ** 0.5
mae_test = mean_absolute_error(test_y, predicted_test)
## For more detials about this function check used_function/custom_functions.py
mape_test = custfun.mean_absolute_percentage_error(test_y, predicted_test)
r2_test = r2_score(test_y, predicted_test)

# Printing metrics
print("RMSE(Root Mean Squared Error):", round(rmse_test, 2))
print("R Squared:", round(r2_test, 2))
print("MAE(Mean Absolute Error):", round(mae_test, 2))
print("MAPE(Mean Absolute Percentage Error):", round(mape_test, 2))
```

```
RMSE(Root Mean Squared Error): 93.16
R Squared: 0.16
MAE(Mean Absolute Error): 51.99
MAPE(Mean Absolute Percentage Error): 59.96
```

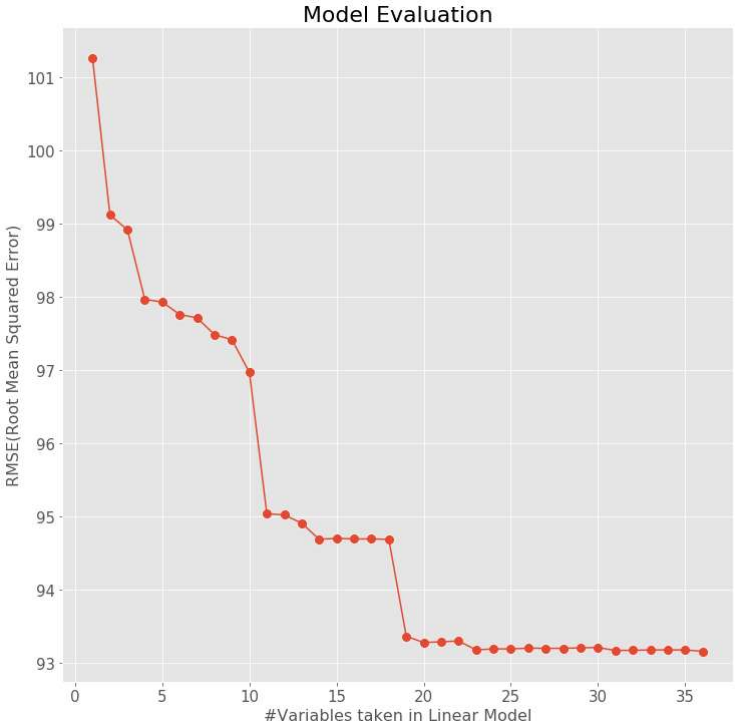
RFE

In [57]:
```
# Creating new dictionary for creating dataframe from it
dictionary = {}

# Some lists to create the columns in dataframe
variables_chosen, no_variables_chosen = [], []
rmse, mae, mape, r2 = [], [], [], []

# Looping over to get various #variable for RFE
for i in range(len(train_x.columns), 0, -1):

    # Creating the RFE
    rfe = RFE(tem_model, n_features_to_select = i, step = 1)

    # Fitting the RFE
    rfe_model = rfe.fit(train_x, train_y)

    # Predicting using rfe model
    predicted = rfe_model.predict(test_x)

    # Appending some information to the lists
    no_variables_chosen.append(i)
    variables_chosen.append(", ".join(train_x.columns.values[list(rfe_model.support_)]))
    rmse.append(mean_squared_error(test_y, predicted) ** 0.5)
    mae.append(mean_absolute_error(test_y, predicted))
    ## For more detials about this function check used_function/custom_functions.py
    mape.append(custfun.mean_absolute_percentage_error(test_y, predicted))
    r2.append(r2_score(test_y, predicted))

# Adding keys and values to dictionary
dictionary["variables_chosen"] = variables_chosen
dictionary["no_variables_chosen"] = no_variables_chosen
dictionary["rmse"] = rmse
dictionary["mae"] = mae
dictionary["mape"] = mape
dictionary["r2"] = r2
```

In [58]:
```
# Creating dataframe from the dictionary
model_information_2 = pd.DataFrame.from_dict(dictionary)
```

In [59]:
```
# Sorting dataframe based on r2 values
model_information_2 = model_information_2.sort_values("r2", ascending = False)
```

In [60]:
```
# Plotting RMSE vs No of variables
# For more detials about this function check used_function/custom_functions.py
custfun.plot_RMSE_VS_NoVar(df = model_information_2)
```



In [61]:
```
model_information_2.head()
```

Out[61]:

	variables_chosen	no_variables_chosen	rmse	mae	mape	r2
0	lights, T1, RH_1, T2, RH_2, T3, RH_3, T4, RH_4...	36	93.163470	51.985929	59.955435	0.159241
5	lights, T1, RH_1, T2, RH_2, T3, RH_3, T4, RH_4...	31	93.175087	52.018301	59.983445	0.159031
4	lights, T1, RH_1, T2, RH_2, T3, RH_3, T4, RH_4...	32	93.177736	51.987624	59.961395	0.158984
3	lights, T1, RH_1, T2, RH_2, T3, RH_3, T4, RH_4...	33	93.182177	51.975326	59.935284	0.158903
2	lights, T1, RH_1, T2, RH_2, T3, RH_3, T4, RH_4...	34	93.182521	51.985096	59.954496	0.158897

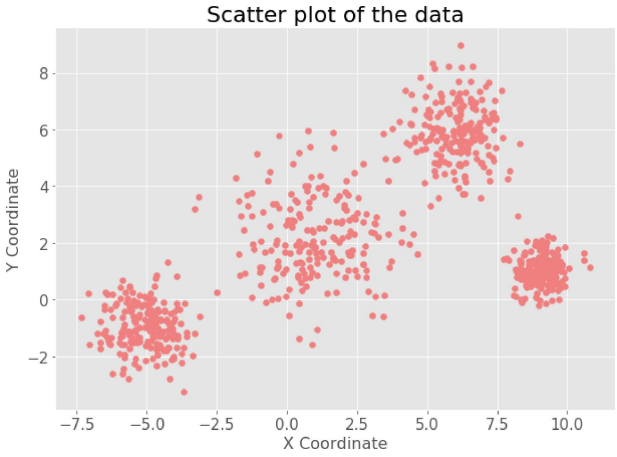
Portfolio 3 - Clustering Visualisation

K-means clustering is one of the simplest and popular unsupervised learning algorithms. Typically, unsupervised algorithms make inferences from datasets using only input vectors without referring to known, or labelled, outcomes. This notebook illustrates the process of K-means clustering by generating some random clusters of data and then showing the iterations of the algorithm as random cluster means are updated.

We first generate random data around 4 centers.

In [62]:
```
import sys
sys.path.append("used_functions/")
import importlib
import numpy as np
import pandas as pd
from tabulate import tabulate
import matplotlib as mpl
from matplotlib import pyplot as plt
plt.style.use("ggplot")
%matplotlib inline
import custom_functions as custfun
importlib.reload(custfun)
import warnings
warnings.filterwarnings('ignore')
import IPython
```

In [63]:
```
# Creating centers
center_1 = np.array([1,2])
center_2 = np.array([6,6])
center_3 = np.array([9,1])
center_4 = np.array([-5,-1])

# Generate random data and center it to the four centers each with a different variance
np.random.seed(5)

# Generating data randomly
data_1 = np.random.randn(200,2) * 1.5 + center_1
data_2 = np.random.randn(200,2) * 1 + center_2
data_3 = np.random.randn(200,2) * 0.5 + center_3
data_4 = np.random.randn(200,2) * 0.8 + center_4

# Concatenating the data arrays in one big array
data = np.concatenate((data_1, data_2, data_3, data_4), axis = 0)

# Plotting an empty figure
plt.figure(figsize = (10,7))

# Plotting data points
plt.scatter(data[:,0], data[:,1], c = "lightcoral")

# Plot title, xlabel and ylabel
plt.title("Scatter plot of the data")
plt.xlabel("X Coordinate")
plt.ylabel("Y Coordinate")

plt.show()
```



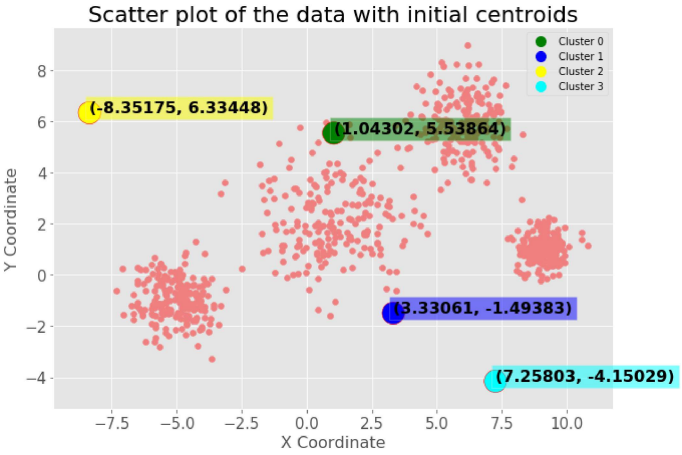
1. Generate random cluster centres

You need to generate four random centres.

This part of portfolio should contain at least:

- The number of clusters `k` is set to 4;
- Generate random centres via `centres = np.random.randn(k,c)*std + mean` where `std` and `mean` are the standard deviation and mean of the data. `c` represents the number of features in the data. Set the random seed to 6.
- Color the generated centers with `green`, `blue`, `yellow`, and `cyan`. Set the edgecolors to `red`.

In [64]:
```
# No of cluster are 4
k = 4

# No of features are 2
c = 2

# Calculating standard deviation and mean for the data
std = data.std()
mean = data.mean()

# Setting the random seed to 6 for reproducibility
np.random.seed(6)

# Creating centroids
centroids = np.random.randn(k,c) * std + mean

# Creating dataframe from centroids
centroid_df = pd.DataFrame(centroids)
centroid_df.index.name = "cluster"

# Colors for coordinates
colors = ["green", "blue", "yellow", "cyan"]

# Plotting an empty figure
plt.figure(figsize = (10,7))

# Plotting data points
plt.scatter(data[:,0], data[:,1], c = "lightcoral")

# Plotting cluster centroids
plt.scatter(centroids[:,0], centroids[:,1], c = colors, edgecolors = "red", s = 500)

# Putting centroid coordinates on the plot
custfun.put_coordinates(df = centroid_df, c = colors)

# Plotting the legend for the plot
patches = [ plt.plot([],[], marker = "o", ms = 10, ls = "", mec = None, color = colors[i],
            label = "Cluster " + str(i))[0]  for i in range(len(colors)) ]
plt.legend(handles = patches, loc = 'upper right', ncol = 1)

# Plot title, xlabel and ylabel
plt.title("Scatter plot of the data with initial centroids")
plt.xlabel("X Coordinate")
plt.ylabel("Y Coordinate")

# Saving plot for a slideshow
plt.savefig('Portfolio3/html/images/0.png', bbox_inches = "tight")

plt.show()
```



2. Visualise the clustering results in each iteration

How Kmeans works?

I. **Initialization -** First step is to randomly assign centroid for the clusters. This is typically done by randomly choosing K(here 4) points from the input set(we have created 4 centroids it using standard deviation and mean of the data in the above cell).

II. **Cluster Assignment -** Here each observation(each data point) is assigned to cluster centroid such that Within Cluster Sum of Squares is minimum or Between Cluster Sum of Square is maximum. Various metrics can be used to calculate similarities/dissimilarities between data points. This generally means assigning the data point to the closest centroid point. Here I have used Euclidean distance, which goes like following.

$$d(P, C) = \sqrt{(x_1 - X)^2 + (y_1 - Y)^2}$$

where (x_1, y_1) is an observation point(P) and (X, Y) is a centroid point(C).

III. **Centroid Update -** After all the points or observations are assigned to the centroids, each centroid is computed again. Here we are using Kmeans that's why new centroids will be computed by averaging(taking mean) the observation or data points which were assigned to the centroids in the **II** step. **[You can see that centroid are being pulled by data points]**

IV. **Convergence -** Step **II** and **III** are repeated until the algorithm converges or some criterion is reached. There are several ways to detect convergence. The most typical way is to run until none of the observations change cluster membership(used here).

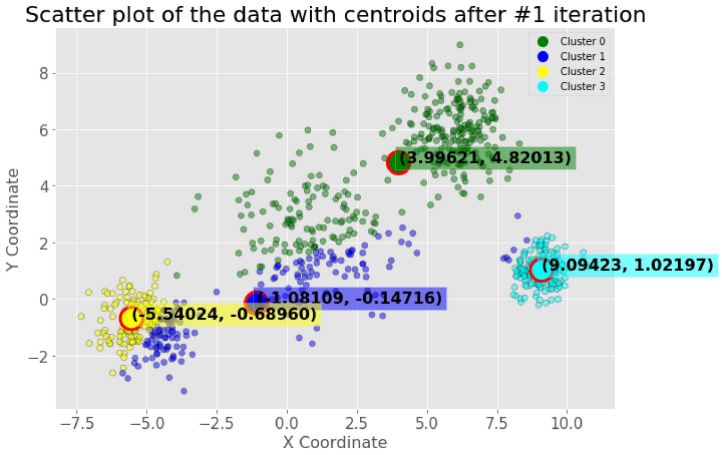
Extras - Kmeans often gives local minima(not the best solution/not the optimal solution). To overcome this, Kmeans is run for several times each time taking a different set of initial random centroids.

In [65]:
```
data_df = pd.DataFrame(data)
current_centroids = pd.DataFrame(centroids)
current_centroids.index.name = "cluster"
previous_centroids = pd.DataFrame()
clusters = pd.DataFrame()
```

First Iteration

In [66]:
```
# Checking if both centroids are same or not
print("Centroids are same ->", current_centroids.equals(previous_centroids))

# For more detials about this function check used_function/custom_functions.py
current_centroids, previous_centroids = custfun.kmean_iteration(data = data_df,
                                                                current_centroids_df = current_centroids,
                                                                previous_centroids_df = previous_centroids)
```

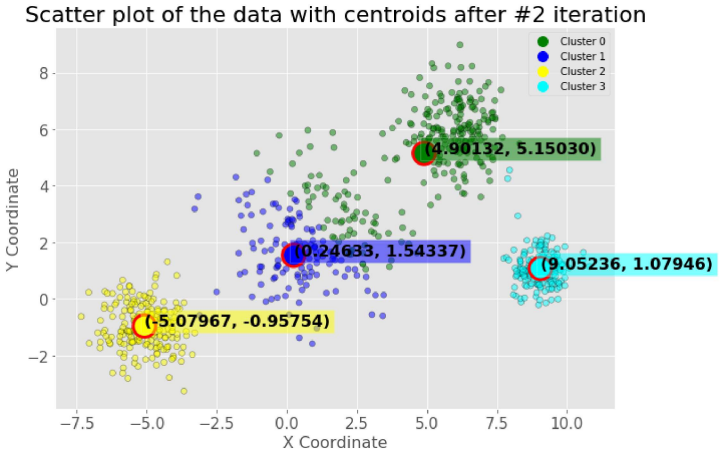
```
Centroids are same -> False
Previous centroids
+-----------+----------+----------+
|   cluster |        0 |        1 |
|-----------+----------+----------|
|         0 |  1.04302 |  5.53864 |
|         1 |  3.33061 | -1.49383 |
|         2 | -8.35175 |  6.33448 |
|         3 |  7.25803 | -4.15029 |
+-----------+----------+----------+
Current centroids
+-----------+----------+-----------+
|   cluster |        0 |         1 |
|-----------+----------+-----------|
|         0 |  3.99621 |  4.82013  |
|         1 | -1.08109 | -0.147156 |
|         2 | -5.54024 | -0.689604 |
|         3 |  9.09423 |  1.02197  |
+-----------+----------+-----------+
```



Second Iteration

In [67]:
```
# Checking if both centroids are same or not
print("Centroids are same ->", current_centroids.equals(previous_centroids))

# For more detials about this function check used_function/custom_functions.py
current_centroids, previous_centroids = custfun.kmean_iteration(data = data_df,
                                                                current_centroids_df = current_centroids,
                                                                previous_centroids_df = previous_centroids)
```

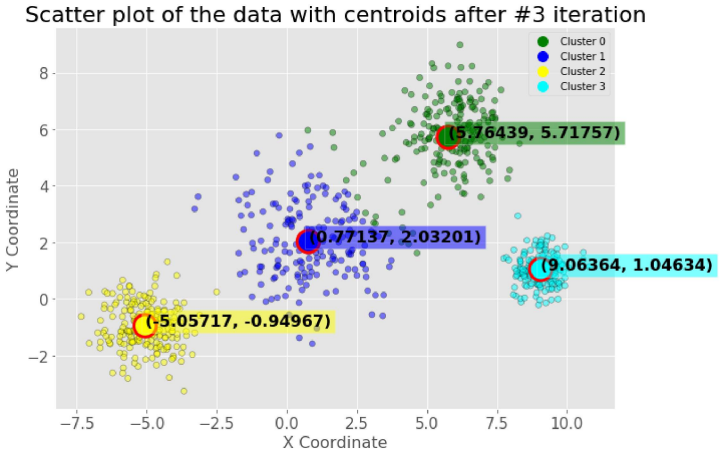
```
Centroids are same -> False
Previous centroids
+-----------+----------+-----------+
|   cluster |        0 |         1 |
|-----------+----------+-----------|
|         0 |  3.99621 |  4.82013  |
|         1 | -1.08109 | -0.147156 |
|         2 | -5.54024 | -0.689604 |
|         3 |  9.09423 |  1.02197  |
+-----------+----------+-----------+
Current centroids
+-----------+----------+-----------+
|   cluster |        0 |         1 |
|-----------+----------+-----------|
|         0 |  4.90132 |  5.1503   |
|         1 |  0.24633 |  1.54337  |
|         2 | -5.07967 | -0.957539 |
|         3 |  9.05236 |  1.07946  |
+-----------+----------+-----------+
```



Third Iteration

In [68]:
```
# Checking if both centroids are same or not
print("Centroids are same ->", current_centroids.equals(previous_centroids))

# For more detials about this function check used_function/custom_functions.py
current_centroids, previous_centroids = custfun.kmean_iteration(data = data_df,
                                                                current_centroids_df = current_centroids,
                                                                previous_centroids_df = previous_centroids)
```

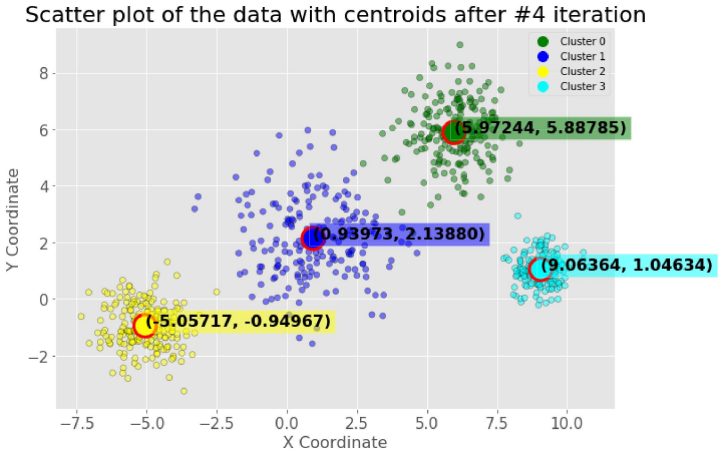
```
Centroids are same -> False
Previous centroids
+-----------+----------+-----------+
|   cluster |        0 |         1 |
|-----------+----------+-----------|
|         0 |  4.90132 |  5.1503   |
|         1 |  0.24633 |  1.54337  |
|         2 | -5.07967 | -0.957539 |
|         3 |  9.05236 |  1.07946  |
+-----------+----------+-----------+
Current centroids
+-----------+-----------+----------+
|   cluster |         0 |        1 |
|-----------+-----------+----------|
|         0 |  5.76439  |  5.71757 |
|         1 |  0.771375 |  2.03201 |
|         2 | -5.05717  | -0.94967 |
|         3 |  9.06364  |  1.04634 |
+-----------+-----------+----------+
```



Forth Iteration

In [69]:
```
# Checking if both centroids are same or not
print("Centroids are same ->", current_centroids.equals(previous_centroids))

# For more detials about this function check used_function/custom_functions.py
current_centroids, previous_centroids = custfun.kmean_iteration(data = data_df,
                                                                current_centroids_df = current_centroids,
                                                                previous_centroids_df = previous_centroids)
```

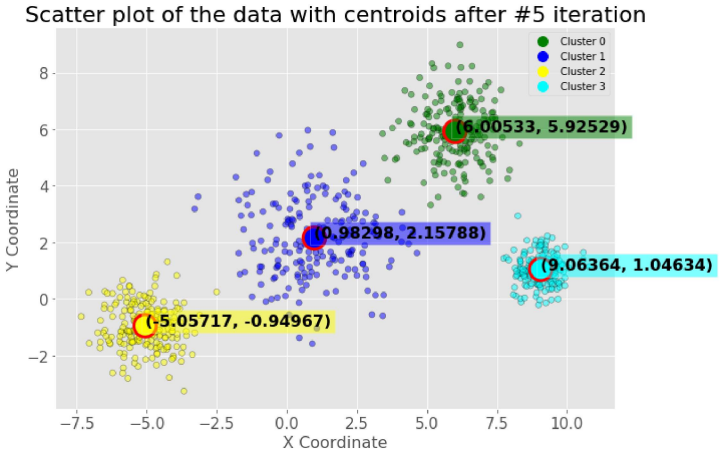
```
Centroids are same -> False
Previous centroids
+-----------+-----------+----------+
|   cluster |         0 |        1 |
|-----------+-----------+----------|
|         0 |  5.76439  |  5.71757 |
|         1 |  0.771375 |  2.03201 |
|         2 | -5.05717  | -0.94967 |
|         3 |  9.06364  |  1.04634 |
+-----------+-----------+----------+
Current centroids
+-----------+-----------+----------+
|   cluster |         0 |        1 |
|-----------+-----------+----------|
|         0 |  5.97244  |  5.88785 |
|         1 |  0.939731 |  2.1388  |
|         2 | -5.05717  | -0.94967 |
|         3 |  9.06364  |  1.04634 |
+-----------+-----------+----------+
```



Fifth Iteration

In [70]:
```
# Checking if both centroids are same or not
print("Centroids are same ->", current_centroids.equals(previous_centroids))

# For more detials about this function check used_function/custom_functions.py
current_centroids, previous_centroids = custfun.kmean_iteration(data = data_df,
                                                                current_centroids_df = current_centroids,
                                                                previous_centroids_df = previous_centroids)
```

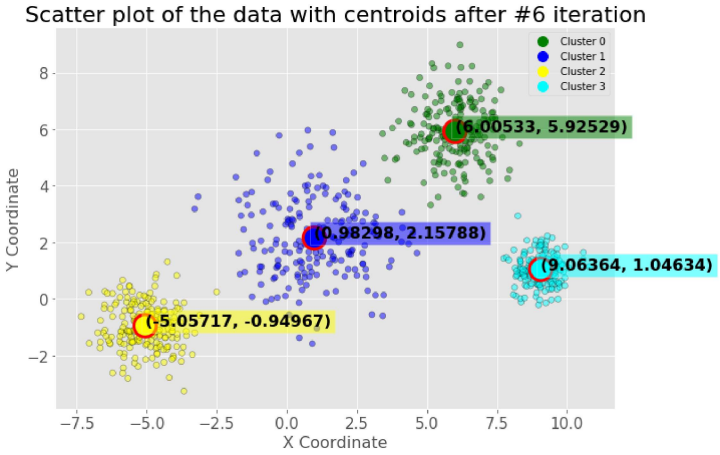
```
Centroids are same -> False
Previous centroids
+-----------+-----------+----------+
|   cluster |         0 |        1 |
|-----------+-----------+----------|
|         0 |  5.97244  |  5.88785 |
|         1 |  0.939731 |  2.1388  |
|         2 | -5.05717  | -0.94967 |
|         3 |  9.06364  |  1.04634 |
+-----------+-----------+----------+
Current centroids
+-----------+-----------+----------+
|   cluster |         0 |        1 |
|-----------+-----------+----------|
|         0 |  6.00533  |  5.92529 |
|         1 |  0.982978 |  2.15788 |
|         2 | -5.05717  | -0.94967 |
|         3 |  9.06364  |  1.04634 |
+-----------+-----------+----------+
```



Sixth Iteration

In [71]:
```
# Checking if both centroids are same or not
print("Centroids are same ->", current_centroids.equals(previous_centroids))

# For more detials about this function check used_function/custom_functions.py
current_centroids, previous_centroids = custfun.kmean_iteration(data = data_df,
                                                                current_centroids_df = current_centroids,
                                                                previous_centroids_df = previous_centroids)
```

```
Centroids are same -> False
Previous centroids
+-----------+-----------+----------+
|   cluster |         0 |        1 |
|-----------+-----------+----------|
|         0 |  6.00533  |  5.92529 |
|         1 |  0.982978 |  2.15788 |
|         2 | -5.05717  | -0.94967 |
|         3 |  9.06364  |  1.04634 |
+-----------+-----------+----------+
Current centroids
+-----------+-----------+----------+
|   cluster |         0 |        1 |
|-----------+-----------+----------|
|         0 |  6.00533  |  5.92529 |
|         1 |  0.982978 |  2.15788 |
|         2 | -5.05717  | -0.94967 |
|         3 |  9.06364  |  1.04634 |
+-----------+-----------+----------+
```



Seventh Iteration

Not needed as centroids are same

In [72]:
```
# Checking if both centroids are same or not
print("Centroids are same ->", current_centroids.equals(previous_centroids))
```

```
Centroids are same -> True
```

Look how centroids are changing in each iteration in the slideshow

In [73]:
```
# Find slideshow.html file in html folder of this portfolio
IPython.display.HTML(filename = 'slideshow.html')
```

Out[73]: