Data Visualization with Python - CET short course - 14 - 15 March 2019

March 6, 2019

1 Your first steps with Python

1.1 Introduction

Python is a general purpose programming language. It is used extensively for scientific computing, data analytics and visualization, web development and software development. It has a wide user base and excellent library support.

There are many ways to use and interact with the Python language. The first way is to access it directly from the command prompt and calling `python <script>.py`. This runs a script written in Python and does whatever you have programmed the computer to do. But scripts have to be written and how do we actually write Python scripts?

Actually Python scripts are just `.txt` files. So you could just open a `.txt` file and write a script, saving the file with a `.py` extension. The downsides of this approach is obvious to anyone working with Windows. Usually, Python source code is written-not with Microsoft Word- but with and **I**ntegrated **D**evelopment **E**nvironment. An IDE combines a text editor with a running Python console to test code and actually do work with Python without switching from one program to another. If you learnt the C, or C++ language, you will be familiar with Vim. Other popular IDE's for Python are Pycharm, Spyder and the Jupyter Notebook.

In this course, we will use the Jupyter Notebook as our IDE because of its ease of use ability to execute code cell by cell. It integrates with markdown so that one can annotate and document your code on the fly! All in all, it is an excellent tool for teaching and learning Python before one migrates to more advanced tools like Spyder for serious scripting and development work.

1.2 Your best friends.

In order to get the most from Python, your best source of reference is the Python documentation. Getting good at Python is a matter using it regularly and familiarizing yourself with the keywords, constructs and commonly used idioms.

Learn to use the Shift-Tab when coding. This activates a hovering tooltip that provides documentation for keywords, functions and even variables that you have declared in your environment. This convenient tooltip and be expanded into a pop-up window on your browser for easy reference. Use this often to reference function signatures, documentation and general help.

Jupyter notebook comes with Tab completion. This quality of life assists you in typing code by listing possible autocompletion options so that you don't have to type everything out! Use

Tab completion as often as you can. This makes coding faster and less tedious. Tab completion also allows you to check out various methods on classes which comes in handy when learning a library for the first time (like `matplotlib` or `seaborn`).

Finally ask Google. Once you have acquired enough "vocabulary", you can begin to query Google with your problem. And more often that not, somehow has experienced the same conundrum and left a message on Stackexchange. Browsing the solutions listed there is a powerful way to learn programming skills.

1.3 The learning objectives for this unit

The learning objectives of this first unit are:

- Getting around the Jupyter notebook.
- Learning how to `print("Hello world!")`
- Using and coding with basic Python objects: `int`, `str`, `float` and `bool`.
- What are variables and valid variable names.
- Using the `list` object and `list` methods.
- Learning how to access items in `list`. Slicing and indexing.
- To use the `dict` data structure and related methods.

2 Getting around the Jupyter notebook

2.1 Cells and colors, just remember, green is for go

All code is written in cells. Cells are where code blocks go. You execute a cell by pressing Shift-Enter or pressing the "play" button. Or you could just click on the drop down menu and select "Run cell" but who would want to do that!

In general, cells have two uses: One for writing "live" Python code which can be executed and one more to write documentation using markdown. To toggle between the two cell types, press Escape to exit from "edit" mode. The edges of the cell should turn blue. Now you are in "command" mode. Escape actually activates "command" mode. Enter activates "edit" mode. With the cell border coloured blue, press `M` to enter into markdown mode. You should see the `In [ ]:` prompt dissappear. Press Enter to change the border to green. This means you can now "edit" markdown. How does one change from markdown to a live coding cell? In "command" mode (remember blue border) press `Y`. Now the cell is "hot". When you Shift-Enter, you will execute code. If you happen to write markdown when in a "coding" cell, the Python kernel will shout at you. (Means raise an error message)

2.1.1 Practise makes perfect

Now its time for you to try. In the cell below, try switching to Markdown. Press Enter to activate "edit" mode and type some text in the cell. Press Shift-Enter and you should see the output rendered in html. Note that this is not coding yet

```
In [ ]: # change this cell into a Markdown cell.
        # Then type something here and execute it (Shift-Enter)
```

2.2 Your first script

It is a time honoured tradition that your very first program should be to print "Hello world!" How is this achieved in Python?

```
In [ ]: '''Make sure you are in "edit" mode and that this cell
        is for Coding ( You should see the In [ ]:)on the left
        of the cell.  '''

        print("Hello world!")
```

Notice that `Hello world!` is printed at the bottom of the cell as an output. In general, this is how output of a python code is displayed to you.

`print` is a special function in Python. It's purpose is to display output to the console. Notice that we pass an argument-in this case a string `"Hello world!"`- to the function. All arguments passed to the function must be enclosed in round brackets and this signals to the Python interpreter to execute a function named `print` with the argument `"Hello world!"`.

2.2.1 Self introductions

Your next exercise is to print your own name to the console. Remember to enclose your name in " "or ' '

```
In [ ]: # print your name in this cell.
```

2.3 Commenting

Commenting is a way to annotate and document code. There are two ways to do this: Inline using the # character or by using `''' <documentation block> '''`, the latter being multi-line and hence used mainly for documenting functions or classes. Comments enclosed using `'''  '''` style commenting are actually registed in Jupyter notebook and can be accessed from the Shift-Tab tooltip!

One should use # style commenting very sparingly. By right, code should be clear enough that # inline comments are not needed.

However, # has a very important function. It is used for debugging and trouble-shooting. This is because commented code sections are never executed when you execute a cell (Shift-Enter)

3 Python's building blocks

Python is an **O**bject **O**riented **P**rogramming language. That means to *all* of python is made out of objects which are instances of `classes`. The main point here is that I am going to introduce 4 basic objects of Python which form the backbone of any program or script.

- Integers or `int`.
- Strings or `str`. You've met one of these: `"Hello world!"`. For those who know about character encoding, it is highly encouraged to code Python with `UTF-8` encoding.
- Float or `float`. Basically the computer version of real numbers.
- Booleans or `bool`. In Python, true and false are indicated by the reserved keywords `True` and `False`. Take note of the capitalized first letter.

3.1 Numbers

You can't call yourself a scientific computing language without the ability to deal with numbers. The basic arithmetic operations for numbers are exactly as you expect it to be

```
In [ ]: # Addition
        5+3

In [ ]: # Subtraction
        8-9

In [ ]: # Multiplication
        3*12

In [ ]: # Division
        48/12
```

Note the floating point answer. In previous versions of Python, / meant floor division. This is no longer the case in Python 3

```
In [ ]: # Exponentiation. Limited precision though!

        16**0.5

In [ ]: # Residue class modulo n

        5%2
```

In the above `5%2` means return me the remainder after 5 is divided by 2 (which is indeed 1).

3.1.1 Variables

We can *assign* values to variables using the = operator.

Python is case sensitive. So a variable name `A` is different from `a`. Variables cannot begin with numbers and cannot have empty spaces between them. So `my variable` is not a valid variable. Instead write `my_variable`

```
In [ ]: # Assignment

        x=1
        y=2
```

```
In [ ]: x+y
```

```
In [ ]: x/y
```

Notice that after assignment, I can access the variables in a different cell. However, if you reassign a variable to a different number, the old values for that variable are overwritten.

```
In [ ]: x=5
        x+y-2
```

3.2 Strings

Strings are basically text. These are enclosed in `' '` or `" "`. The reason for having two ways of denoting strings is because we may need to nest a string within a string like in `'The quick brown fox "jumped" over the lazy old dog'`. This is especially useful when setting up database queries and the like.

```
In [ ]: # Noting the difference between printing
        # quoted variables (strings) and printing
        # the variable itself.
        x = 5

        print(x)
        print('x')
```

In the second print function, the text `'x'` is printed while in the first print function, it is the contents of `x` which is printed to the console.

3.2.1 String formatting

Strings can be assigned to variables just like numbers. And these can be recalled in a print function.

```
In [ ]: my_name = 'Tang U-Liang'
        print(my_name)
```

```
In [ ]: # String formatting: Using f-string syntax
        age = 35
        weight = 70.25
        print(f'Hello doctor, my name is {my_name}.'+
              f' I am {age} years old. I weigh {weight:.1f} kg')
```

3.2.2 Slicing

Slicing is a way of get specific subsets of the string. If you let x_n denote the $n+1$-th letter (note zero indexing) in a string (and by letter this includes whitespace characters as well!) then writing `my_string[i:j]` returns a subset

$$x_i, x_{i+1}, \ldots, x_{j-1}$$

of letters in a string. That means the slice `[i:j]` takes all subsets of letters starting from index `i` and stops *one index before* the index indicated by `j`.

0 indexing and stopping point convention frequently trips up first time users. So take special note of this convention. 0 indexing is used throughout Python especially in `matplotlib` and `pandas`.

```
In [ ]: favourite_drink = fruit+' '+drink
        print("Printing the first to 3rd letter.")
        print(favourite_drink[0:3])
        print("\nNow I want to print the second to seventh letter:")
        print(favourite_drink[1:7])
```

Notice the use of `\n` in the second `print` function. This is called a newline character which does exactly what its name says. Also in the third `print` function notice the seperation between `e` and `j`. It is actually not seperated. The sixth letter is a whitespace character `' '`.

Slicing also utilizes arithmetic progressions to return even more specific subsets of strings. So `[i:j:k]` means that the slice will return

$$x_i, x_{i+k}, x_{i+2k}, \ldots, x_{i+mk}$$

where m is the largest (resp. smallest) integer such that $i + mk \leq j - 1$ (resp $1 + mk \geq j + 1$ if $i \geq j$)

```
In [ ]: print(favourite_drink[0:7:2])
```

```
In [ ]: # Here's a trick, try this out
        print(favourite_drink[3:0:-1])
```

3.3 The `list` data structure.

`list` is a way to store multiple objects in an array like structure. Declaring a list is as simple as using square brackets `[ ]` to enclose a list of objects (or variables) seperated by commas.

```
In [ ]: # Here's a list called staff containing
        # his name, his age and current renumeration

        staff = ['Andy', 28, 980.15]
```

3.3.1 Properties of `list` objects and indexing

One of the fundamental properties we can ask about lists is how many objects they contain. We use the `len` (short for length) function to do that.

```
In [ ]: len(staff)
```

Perhaps you want to recover that staff's name. It's in the first position of the list.

```
In [ ]: staff[0]
```

Notice that Python still outputs to console even though we did not use the `print` function. Actually the print function prints a particularly "nice" string representation of the object, which is why `Andy` is printed without the quotation marks if `print` was used.

Can you find me `Andy`'s age now?

The same slicing rules for strings apply to lists as well. If we wanted Andy's age and wage, we would type `staff[1:3]`

```
In [ ]: staff[1:3]
```

This returns us a sub-list containing Andy's age and renumeration.

3.3.2 List methods

Right now, let us look at four very useful list methods. Methods are basically operations which modify lists. These are:

1. `pop` which allows us to remove an item in a list.

So for example if $x_0, x_1, \ldots, x_n$ are items in a list, calling `my_list.pop(r)` will modify the list so that it contains only

$$x_0, \ldots, x_{r-1}, x_{r+1}, \ldots, x_n$$

while returning the element x_r.

3. `append` which adds items to the *end* of the list.

Let's say x_{n+1} is the new object you wish to append to the end of the list. Calling the method `my_list.append(x_n+1)` will modify the list inplace so that the list will now contain

$$x_0, \ldots, x_n, x_{n+1}$$

Note that `append` does *not return any output*!

2. `insert` which as the name suggests, allows us to add items to a list *in a particular* index location

When using this, type `my_list.insert(r, x_{n+1})` with the second argument to the method the object you wish to insert and `r` the position (still 0 indexed) where this object ought to go in that list. This method modifies the list inplace and does not return any output. After calling the `insert` method, the list now contains

$$x_0, \ldots, x_{r-1}, x_{n+1}, x_r, \ldots, x_n$$

This means that `my_list[r] =` x_{n+1} while `my_list[r+1] =` x_r

1. `+` is used to concatenate two lists. If you have two lists and want to join them together producing a union of two (or more lists), use this binary operator.

This works by returning a union of two lists. So

$$[x_1, \ldots, x_n] + [y_1, \ldots, y_m]$$

is the list containing

$$x_1, \ldots, x_n, y_1, \ldots, y_m$$

This change is **not permanent** unless you assign the result of the operation to another variable.

```
In [ ]: # append

        staff.append('Finance')
        print(staff)
```

```
In [ ]: # pop away the information about his salary

        andys_salary = staff.pop(2)
        print(andys_salary)
        print(staff)

In [ ]: # oops, made a mistake, I want to reinsert
        # information about his salary

        staff.insert(3, andys_salary)
        print(staff)

In [ ]: contacts = [99993535, "andy@company.com"]

        # reassignment of the concatenated list back to staff

        staff = staff+contacts
        print(staff)
```

3.4 Dictionaries

Lists - while easy to create- have the weakness that one cannot easily retrieve data that has been already stored in it. Since the primary means of retrieving information in a list is via indexing and slicing, you need to know the exact integer positions of each data stored in the list. This can (and will often) lead to human errors in programming. Furthermore, an integer based recall is unenlightening. Other people who reads your code will find it difficult to understand what is being written.

To remedy this, Python has built into its base package a data structure called **dictionaries**. A dictionary is simply a key-value pairing like so

$$(key_1, value_1), (key_2, value_2) \ldots, (key_n, value_n)$$

where key_i are usually (but not always) strings and $value_i$ any Python object (`int`, `str`, `list` and even other `dict`!)

If `my_dictionary` is a dictionary. Then calling `my_dictionary[key_1]` will return you the value associated with `key_1` in `my_dictionary`, say `value_1`.

3.4.1 Creating dictionaries

Dictionaries are created using curly braces `{ }`. Inside the curly braces, we simply list down all the key-value pairs with a colon `:`. Different pairs are seperated by a `,`.

```
In [ ]: # creating a dictionary and assigning it to a variable

        staff = {'name': 'Andy', 'age': 28, 'email': 'andy@company.com' }

In [ ]: staff['name']

In [ ]: staff['age']
```

```
In [ ]: print(staff['email'])

In [ ]: # A dictionary is of class dict
        print(type(staff))

In [ ]: # list of all keys, note the brackets at the end.
        # .keys is a method associated to dictionaries

        staff.keys()

In [ ]: # list of all values, in no particular order
        staff.values()

In [ ]: # list all key-value pairings using .items

        staff.items()
```

3.4.2 Updating dictionaries

Very often, we need to change dictionary values and/or add more entries to our dictionary.

```
In [ ]: # Hey, Andy mistakenly keyed in his age.
        # He is actually 29 years old!

        staff['age'] = 29
        print(staff)

In [ ]: # HR wants us to record down his staff ID.

        staff['id'] = 12345
        print(staff)

In [ ]: # Let's check the list of keys
        staff.keys()
```

3.4.3 Using the `.update` method

To combine two dictionaries, we use the `.update` method.

```
In [ ]: staff.update({'salary': 980.15,
                       'department':'finance',
                       'colleagues': ['George', 'Liz']})
```

4 The `pandas` library

Vectorized operations and array support is provided in Python via the `pandas` library. In this section, we will learn about `Series` and how to manipulate it.

A `Series` is an indexed data structure which stores data in an array. It is analogous to a single column in Excel. Data stored in a `Series` can be accessed quickly using its index and allows us to perform "vectorised" operations.

Contents:

- Series Structure
- Basic Operations
- Indexing and Selecting Data
- Boolean Masking
- Missing Values

The `numpy` (stands for **num**erical **py**thon) library and `pandas` (short for **pan**el **da**ta) provide us with array support and `Series` data structure.

```
In [ ]: #import statement for the numpy and pandas libraries
        import numpy as np
        import pandas as pd
```

4.1 Series

In base Python, we can store data using a list - which stores it in an array like structure. We access each item we want by using a *positional* index. But sometimes, a positional index can be inconvenient. It is desireable that we have a flexibility of defining a custom index so that retrieving data is made much more convenient.

At this juncture, it is sufficient to understand series as a list (usually of objects with the same `dtype`) given together with an *index*.

To create a series, we may call the `Series` class and initiate and instance by passing an iterable (a `list` or a numpy `array`) upon initialization.

```
In [ ]: # Create a series from a list
        s = pd.Series([2,3,5,7,11])
        # Display the series
        s
```

The contents of the series is stored as an *attribute* of the series and is accessed with `s.values`. `s.values` is an *array* and not a normal list of objects. The index itself can be accessed with `s.index`.

```
In [ ]: # Get the series values
        print(s.values)

        # Get the series index
        print(s.index)
```

```
In [ ]: # Series index may be reassigned
        s.index = ["a", "b", "c", "d", "e"]

        # Display
        s
```

4.1.1 Accessing series contents: slicing

We may use basic slicing operations to access objects stored in series.

```
In [ ]: # Accessing series by position
        s[0]
```

```
In [ ]: # Accessing series by index
        s["b"]
```

In this regard, we may think of a series as a dictionary. However there is an important difference. Dictionaries do not support slicing. With series, we can perform slicing with the index entries.

```
In [ ]: # Slicing using index. Slicing using index right inclusive.
        s["a": "d"]
```

```
In [ ]: # Contrast this with slicing using positional entries.
        s[0: 3]
```

4.1.2 `.loc` and `.iloc` selection based slicers

If the index is assigned numerical labels, we can recover the right inclusive behaviour using the `.loc` property

```
In [ ]: s.index = range(1,6)
        # Series defaults to positional slicing when
        # numbers are used to contruct a slice.
        s[1:4]
```

In contrast, the use of `.loc` property instructs `pandas` to use numerical labels to access the array instead. Note the right inclusive behaviour.

```
In [ ]: s.loc[1:4]
```

Take note of a common "gotcha" when using numerical indices. When using `s[n]` where n is intended to be the positional index, pandas interprets this as a numerical label instead. To remove ambiguity then, use `.loc` to clearly indicate label based selection and `.iloc` for positional based selection.

```
In [ ]: s
```

```
In [ ]: # Note that s[0] is an error and s[1] is not 3 but 2
        s[0]
```

```
In [ ]: s[1]
```

Instead use `iloc` to clearly indicate that positional based selection is intended.

```
In [ ]: # s.iloc[1] means take the entry at the 2nd position
        s.iloc[1]
```

```
In [ ]: # s.loc[1] and s[1] mean the same thing
        s.loc[1] == s[1]
```

4.1.3 Statistical functions on series

Everyday statistical functions are available as method calls to the Series object.

```
In [ ]: # Create a series
        import scipy.stats as stats

        s = pd.Series(stats.norm.rvs(loc=23.4, scale=5, size=100, random_state=1234567))
```

```
In [ ]: # Basic statistical functions are available as method calls to the series
        s.mean()
```

```
In [ ]: # Std Dev
        s.std()
```

```
In [ ]: # Median / Q1 / Q3
        s.median() # same as s.quantile(0.5)
```

```
In [ ]: # Max/Min value
        s.max()
```

```
In [ ]: # Which index has the max/min value?
        s.idxmax()
```

4.1.4 Conditional Selection

Besides index and positional based slicing of a series, we can extract data from series using condition or logical based selection.

```
In [ ]: s = pd.Series([65, 90, 101, -7, 125], index=["aa", "ab", "cd", "ce", "ag"])
        # Which values are < 10?
        s
```

Let's search for all entries which are even numbers

```
In [ ]: s % 2 ==0
```

The above is known as a *mask*. Think of it as a filter by which we sift out entries corresponding to `False` and retain only those which are `True`. The above indicates that the entry corresponding to index `"ab"` is an even number.

```
In [ ]: # This displays the series consisting only of even numbers
        s[s%2==0]
```

An range based selection criteria like min $< x <$ max can only be implement using "bit wise" logical operators. That means a criteria like $ 50 50) & (s < 100)‘

```
In [ ]: # Select values which are 2 < x < 10.
        s[(s > 50) & (s < 100)]
```

Use bit wise "or" , `|`, to do range based selection of the form $x < a$ or $x > b$.

```
In [ ]: s[(s < 25) | (s > 100)]
```

The method `.isin()` allows us to select entries that are contained in a given list

```
In [ ]: # The list is passed to the method as an argument.
        # Note that not all entries in the list needs to
        # be present in the series.

        s.isin([-7, 65, 90, 100])
```

```
In [ ]: s[s.isin([-7, 65, 90, 100])]
```

4.1.5 Missing Values

NaN (Not a Number) is the standard missing marker used in pandas. There are methods for us to identify missing values, the indices corresponding to missing values and how to filter missing values.

```
In [ ]: # Create a new series
        s1 = pd.Series([13,np.nan,19])
        s1
```

```
In [ ]: # Check for missing value
        s1.isnull()
```

```
In [ ]: # Drop missing values
        s1.dropna()
```

```
In [ ]: # Replace missing values
        s1.fillna(17)
```

4.2 Data frames

A `pandas` data frame is a library to provide users with tabular or spreadsheet support. A data frame is like an Excel table or R `data.frame` with rows representing individual records and columns representing attributes or features of an observation.

Content:

- DataFrame Structure
- Working with Columns
- Conditional Selection

A data frame has both row index and column index.

```
In [8]: # Import liraries
        import numpy as np
        import pandas as pd
```

4.2.1 DataFrame Structure

Initializing a `Dataframe` from a Python dictionary.

```
In [ ]: # Create a Python dictionary
        data = {'Region':['Central','East','North','North-East','West'],
                'Area':[132.7,93.1,134.5,103.9,201.3],
               'Population':[939890,693500,531860,834450,903010]}
        # Create a DataFrame from dict
        df = pd.DataFrame(data)
        # Display df
        df
```

Note that the row index are assigned automatically and column index are arranged in alphabetical order.

```
In [ ]: # Rearrange the columns
        df = pd.DataFrame(data,columns=['Region','Population','Area'])
        df

In [ ]: # Display columns names
        df.columns

In [ ]: # Display all values
        df.values

In [ ]: # Get the shape of the dataframe, i.e. number of rows and columns
        df.shape

In [ ]: # Size of DataFrame = row x column
        df.size

In [ ]: # Number of rows
        len(df)

In [ ]: # Programming specific information of the dataframe
        df.info()

In [ ]: # Basic statistical description of numerical columns
        df.describe()
```

4.2.2 Working with columns

```
In [ ]: # Rename a column label
        df = df.rename(columns={'Population':'Pop'})
        df

In [ ]: # Select a single column to series
        A = df['Area'] # same answer as df.Area
        A
```

```
In [ ]: # Select a single column to dataframe
        B = df[['Area']]
        B

In [ ]: # Select multiple columns to dataframe
        C = df[['Area','Pop']]
        C

In [ ]: # Change order of columns
        D = df[['Region','Area','Pop']]
        D

In [ ]: # Drop a column by label
        E = df.drop('Area',axis=1)
        E

In [ ]: # Create a new column 'Density' = 'Population'/'Area'
        df['Density'] = df['Pop']/df['Area']
        df

In [ ]: # Sort values by a column, in ascending/descending order
        df.sort_values(by=['Pop'], ascending=False)

In [ ]: # Find index label for max/min values
        df['Density'].idxmax()
```

There are many commonly used column-wide attributes/methods: - df['col'].size - df['col'].describe() - df['col'].count() - df['col'].sum() - df['col'].max() - df['col'].mean() - df['col'].std()

```
In [ ]: # Example: Find total population
        df['Pop'].sum()
```

4.2.3 Conditional Selection

```
In [ ]: # Boolean masking for pop > 800000
        df['Pop']>800000

In [ ]: # Return df by boolean masking
        df[df['Pop']>800000]

In [ ]: # Boolean masking for Region == Central
        df['Region']=='Central'

In [ ]: # Select rows by boolean masking
        df[df['Region']=='Central']

In [ ]: # Using .loc to find the Area of the Central region.
        df.loc[df['Region']=='Central', 'Area']
```

```
In [ ]: # Multiple conditions (and: &) (or: |)
        # Example: Pop < 800000 and Density < 8000
        (df['Pop'] < 800000) & (df['Density']<8000)
```

```
In [ ]: # Select rows by multiple conditions
        # Example: Pop < 800000 and Density < 8000
        df[(df['Pop'] < 800000) & (df['Density']<8000)]
```

```
In [ ]: # Using .query() method
        # Query the columns of a frame with a boolean expression
        df.query("Pop < 800000 & Density < 8000")
```

4.3 Data manipulation

With a basic data frame in hand, we now proceed to learn how to manipulate the data to extract information from it.

Content:

- Groupby: split-apply-combine
- Melting dataframes (wide-form to long-form)

Import and understand data set

In this unit, we will be using the MovieLens datasets.

Citation: F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 (December 2015), 19 pages. DOI=http://dx.doi.org/10.1145/2827872

We will import the MovieLens user data set from directly from url: http://files.grouplens.org/datasets/movielens/ml-100k/u.user

Description: Demographic information about MovieLens users.

This is a tab separated list of 'user_id' | 'age' | 'gender' | 'occupation' | 'zip_code'

```
In [9]: # Import data from url
        user = pd.read_csv('http://files.grouplens.org/datasets/movielens/ml-100k/u.user',
                           sep='|', header=None)

        user.columns = ['user_id','age','gender','occupation','zip_code']

        # Show the last 5 rows
        user.tail()
```

```
Out[9]:      user_id  age gender     occupation zip_code
        938      939   26      F        student    33319
        939      940   32      M  administrator    02215
        940      941   20      M        student    97229
        941      942   48      F      librarian    78209
        942      943   22      M        student    77841
```

```
In [ ]: # Any missing values?
        user.info()
```

```
In [ ]: # How many unique occupations are there in the data set?
        user['occupation'].unique()

In [ ]: # Count the number of unique occupations.
        user['occupation'].nunique()

In [ ]: # How many users for each occupation?
        us = user['occupation']
        us.value_counts()
```

4.3.1 Groupby : Split - Apply - Combine

The Groupby operation allows us to group rows of data together and call aggregate functions.

We can think of it as a split-apply-combine operation, where the data set is split on specific keys, then apply a certain aggregation functions (e.g. sum) within each smaller groups, then combine the results into an output array.

```
In [ ]: # Groupby will return a GroupBy object,
        # upon which we can apply some functions

        gp = user.groupby('occupation')

In [ ]: # What are the mean age for each occupation?
        gp.mean()

In [ ]: # Return a series only
        gp['age'].mean()

In [ ]: # Sort the mean age by descending order.
        # The style of coding below is called "method - chaining "

        gp['age']\
            .mean()\
            .sort_values(ascending=False)

In [ ]: # Get all summarized statistics about the age for each occupation.
        gp['age'].describe()

In [ ]: # Apply customised function
        # Example: Find the range of age for each occupation,
        # i.e. range = max - min

        gp['age'].apply(lambda x: x.max()-x.min())
```

Alternatively, use pivot_table method, which summarizes in a two-dimentional table.

```
In [ ]: # Implementation of Excel's pivot table in Python
        user.pivot_table(values='age', index='occupation',
                         columns='gender', aggfunc='mean', fill_value=0)
```

Optional

Pandas pivot tables are able support multi-indexed tables for a richer presentation of summary data.

```
In [ ]: # Find the number of user, mean age and max age
        # for each combination of occupation and gender
        multi_gp = user.groupby(['occupation','gender'])

        multi_gp['age'].agg(['count', 'mean', 'max'])


        # Output is a multi-index data frame
```

4.3.2 Wide-form vs Long-form data frames

We need to distinguish between to different ways of tabulating data. A wide-form data frame is one where each column represents a repeated measurement made for a single observation. On the other hand, the same information could be presented in a long-form table where each row represents a *single* instance of measurement (who was measured together with the result of measurement). Hence, a single sampling subject is represented in *multiple* rows.

```
In [ ]: # Read 'Marksheet.xlsx' file with sheetname 'Quiz'
        quiz = pd.read_excel('Marksheet.xlsx',sheetname='Quiz')
        # This data frame is in wide format
        quiz.head()

In [ ]: # Melt the wide-form df to long-form df
        quizlong = pd.melt(quiz,id_vars='ID',
                           value_vars=['Quiz1','Quiz2','Quiz3','Quiz4','Quiz5'])
        quizlong.tail()

In [ ]: # Shape of the long-form df
        quizlong.shape

In [ ]: # Reverse process: Change long-form to wide-form by using pivot
        quizwide = quizlong.pivot(index='ID',columns='variable',values='value')
        quizwide.reset_index().head()
```

5 Statistical charts with `seaborn`

The seaborn library provides users with quick access to statistical charting. It is a powerful library with a wide array customizable options for chart outputs.

Contents

- Distribution plots
- Bar charts for categorical data
- Box and whisker plots

```
In [11]: import seaborn as sns
         import numpy as np
         import pandas as pd
         %matplotlib inline

         columns=["mpg", "cylinders", "displacement", "horsepower", "weight",
                  "acceleration", "model year", "origin", "car name"]
         raw_mpg = pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/auto-m
                               header=None,
                               delim_whitespace=True,
                               names=columns,
                               usecols=columns,
                               na_values="?",
                               quoting=2,
                               quotechar='"',
                               dtype={"cylinders": np.int64,
                                      "model year": np.int64,
                                      "origin": np.int64})
```

```
In [ ]: raw_mpg.head()
```

```
In [ ]: raw_mpg.info()
```

We investigate the `mpg` dataset using charts generated by `seaborn`. The task associated with this dataset is to predict the `mpg` (Miles per Gallon) variable in terms of 5 continuous variables and 3 discrete ones.

Source: (https://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.names)

5.1 Histograms

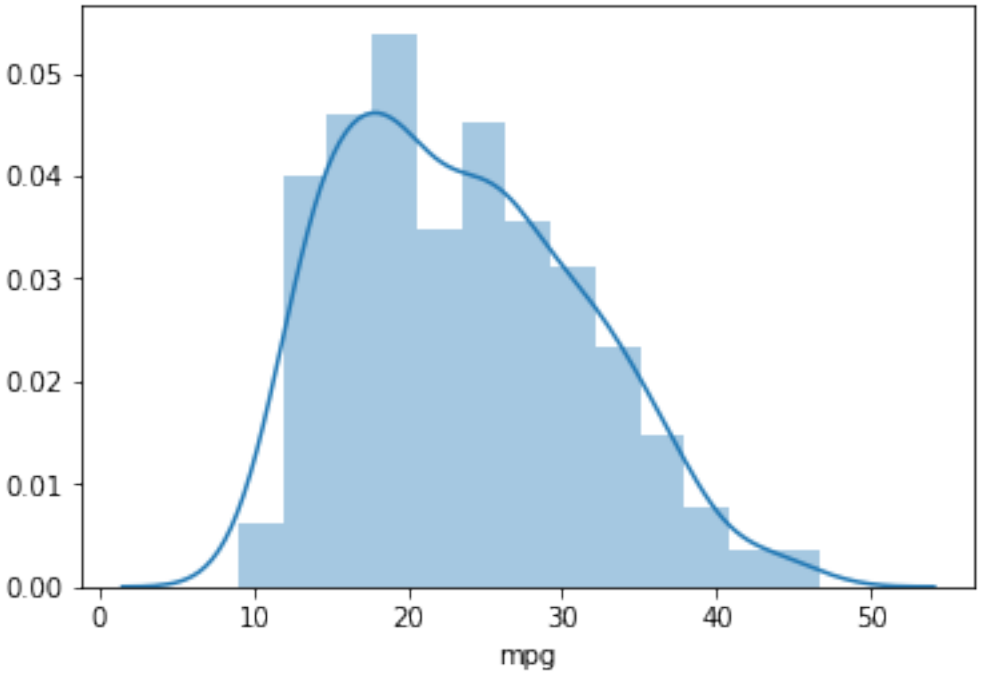
A fundamental tool to investigate continuous variables is the use of histograms. This chart is available by calling the `distplot` function in `seaborn`.

Here, we want to visualize the distribution of miles per gallon variable.

```
In [15]: sns.distplot(raw_mpg.mpg)
```

```
C:\Users\s42355\AppData\Local\conda\conda\envs\sklearn\lib\site-packages\matplotlib\axes\_axes.p
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

```
Out[15]: <matplotlib.axes._subplots.AxesSubplot at 0x1e215df7978>
```

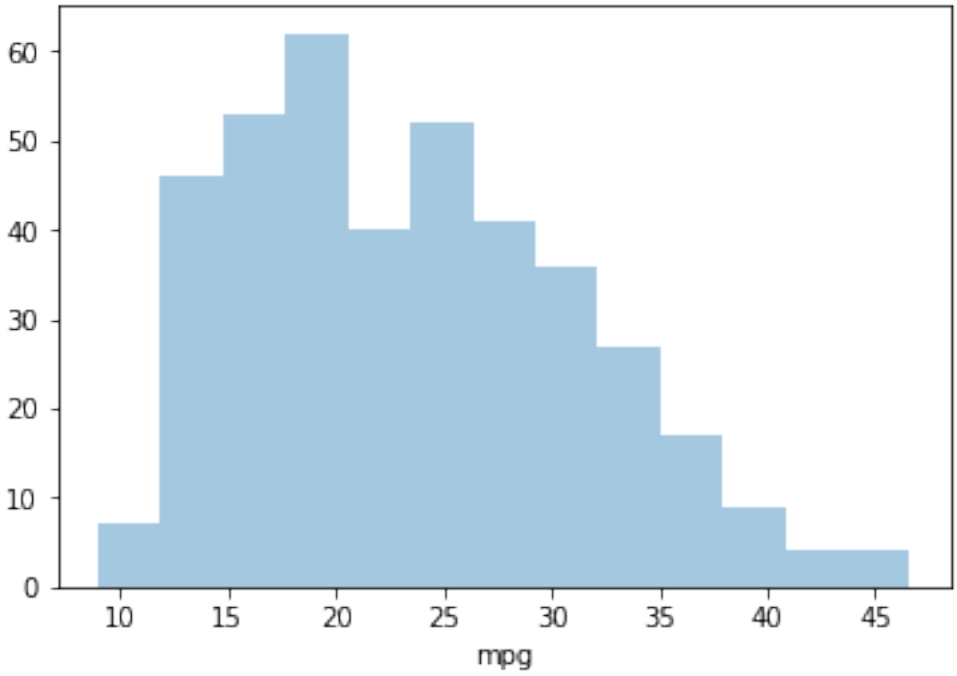


The basic seaborn chart displays a histogram and a kernel density estimate (KDE) of the distribution. The KDE can be turned off by setting the `kde` keyword argument to `False` (the default is `True`).

```
In [16]: sns.distplot(raw_mpg.mpg, kde=False)
```

```
C:\Users\s42355\AppData\Local\conda\conda\envs\sklearn\lib\site-packages\matplotlib\axes\_axes.p
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

```
Out[16]: <matplotlib.axes._subplots.AxesSubplot at 0x1e216e852b0>
```

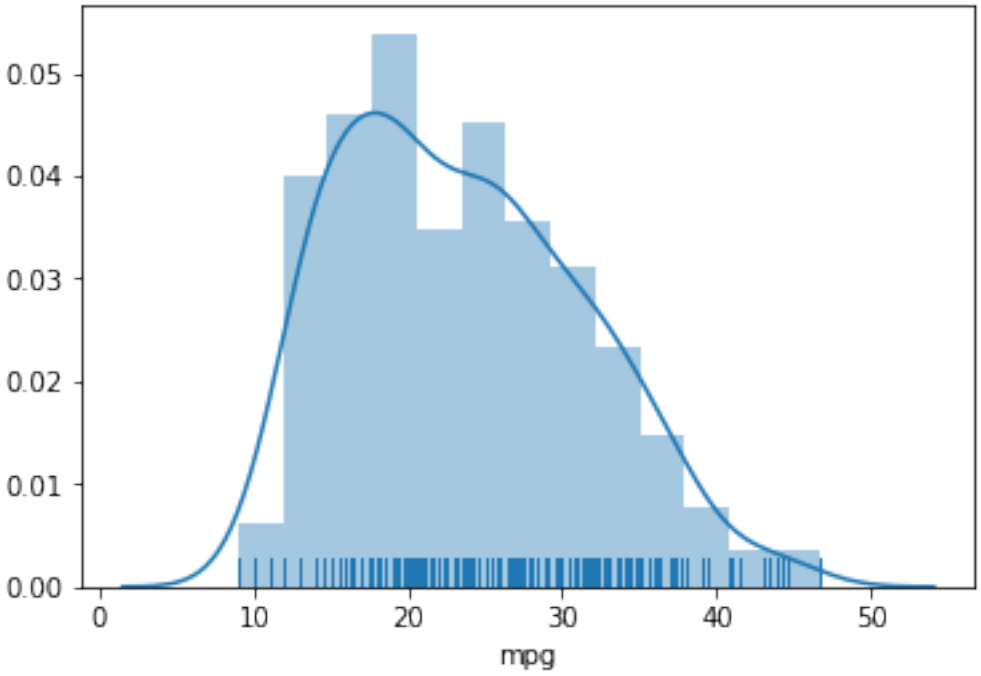


We add in little markers to the plot for each data point. This helps give us a better understanding of the density of the variable

```
In [17]: sns.distplot(raw_mpg.mpg, rug=True)
```

```
C:\Users\s42355\AppData\Local\conda\conda\envs\sklearn\lib\site-packages\matplotlib\axes\_axes.p
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

```
Out[17]: <matplotlib.axes._subplots.AxesSubplot at 0x1e216ec1d68>
```

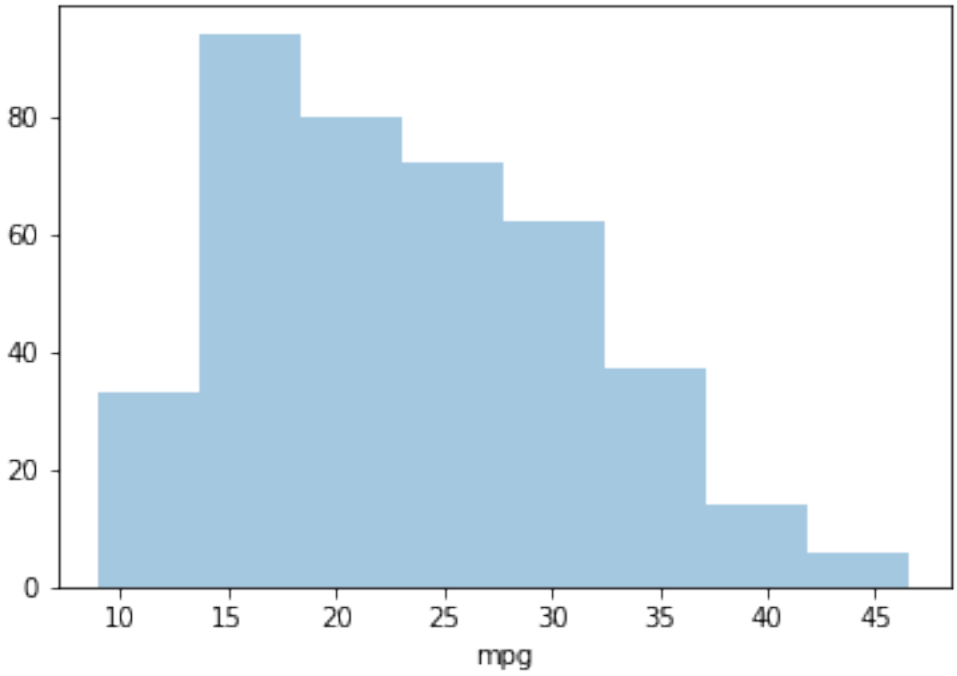


The bin size of a histogram can be customised by passing the desired number of bins to the `bins` argument of the function.

```
In [18]: sns.distplot(raw_mpg.mpg, kde=False, bins=8 )
```

```
C:\Users\s42355\AppData\Local\conda\conda\envs\sklearn\lib\site-packages\matplotlib\axes\_axes.p
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

```
Out[18]: <matplotlib.axes._subplots.AxesSubplot at 0x1e217410780>
```

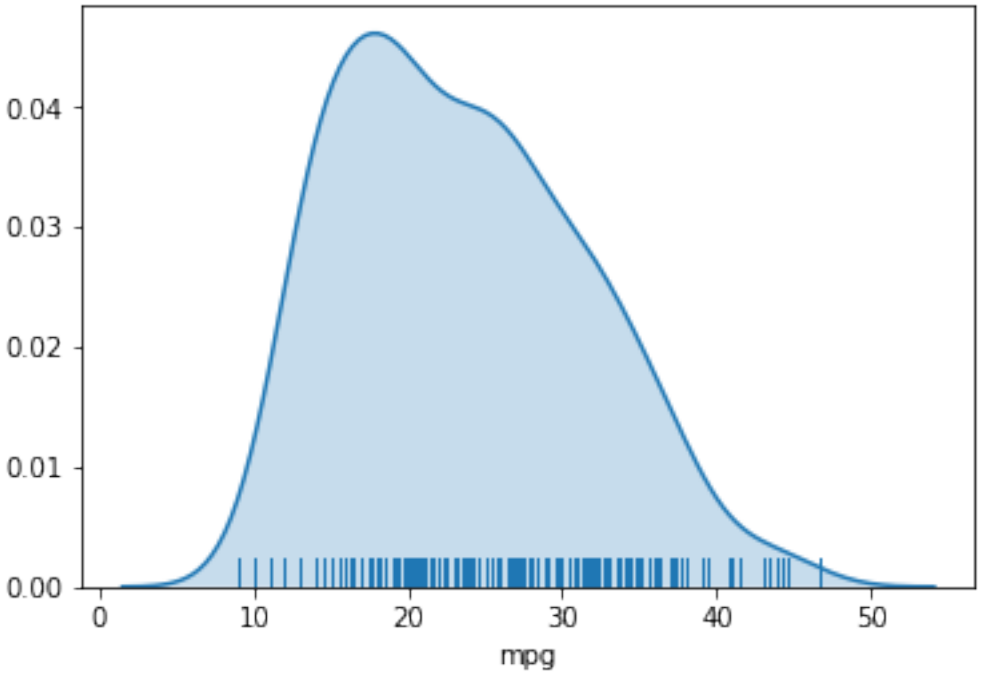


Note that the smaller the number of bins, the distribution becomes coarser and more details of the distribution is left out.

We can also drop the histogram and just plot a KDE estimate of the distribution. Below we plot a rugplot as well and shade in the KDE using a dictionary to pass extra arguments to the underlying KDE plotter.

```
In [19]: sns.distplot(raw_mpg.mpg, hist=False, rug=True, kde_kws={"shade": True})
```

```
Out[19]: <matplotlib.axes._subplots.AxesSubplot at 0x1e2174641d0>
```

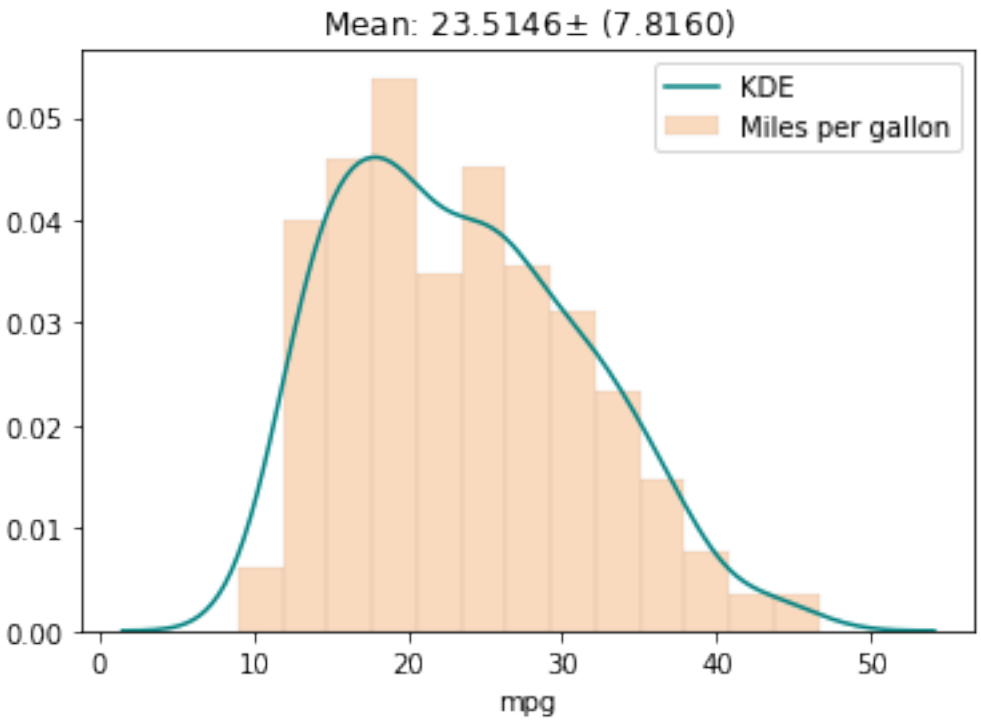


Finally, let's add in a legend label and change the color of the plot, give it a title and further customize the plot by adding a title.

```
In [67]: g = sns.distplot(raw_mpg.mpg, kde=True, color="sandybrown",
                         label="Miles per gallon",

                         # border color of each bar in the histogram
                         hist_kws={"ec": "black",

                                   # the thickness of the border
                                   "lw": 0.1},
                         # labels the KDE estimate line so that it shows
                         # up on the legend
                         kde_kws={"label": "KDE",

                                  # color the of KDE line
                                  "color": "teal"})
         g.legend()
         g.set_title(f'Mean: {raw_mpg.mpg.mean():.4f}'+
                     f'$\pm$ ({raw_mpg.mpg.std():.4f})')
```

```
C:\Users\s42355\AppData\Local\conda\conda\envs\sklearn\lib\site-packages\matplotlib\axes\_axes.p
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

```
Out[67]: Text(0.5,1,'Mean: 23.5146$\\pm$ (7.8160)')
```

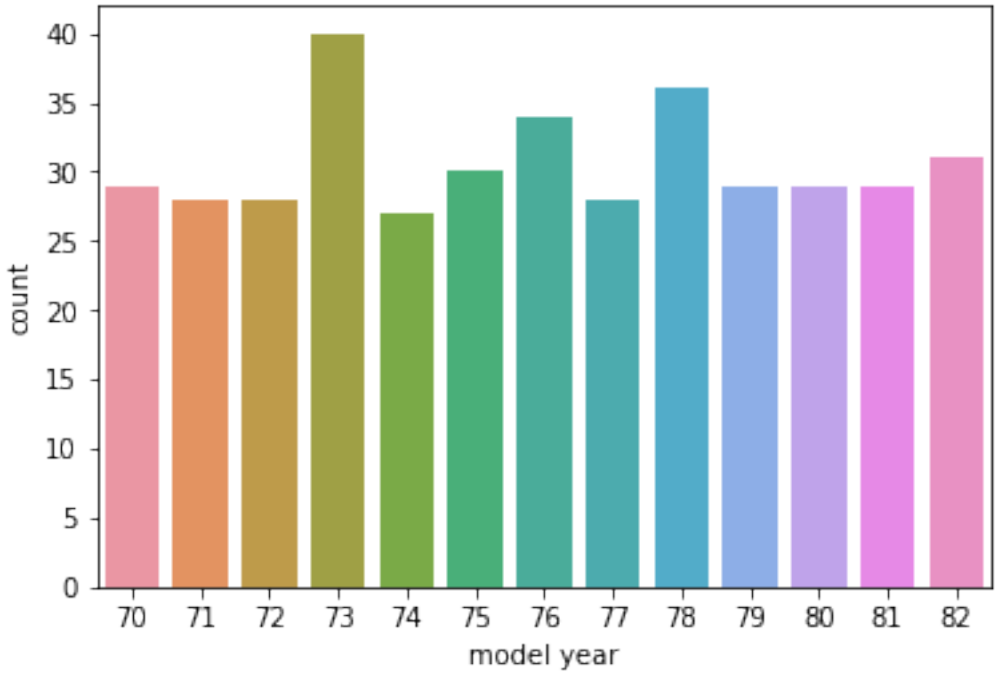


5.2 Analysis of nominal variables with a bar chart

The `mpg` dataset has three categorical or nomimal variables. One basic way to understand nominal data its through a frequency count. To create frequency bar charts use the `countplot` command.

```
In [21]: sns.countplot(raw_mpg["model year"])
```

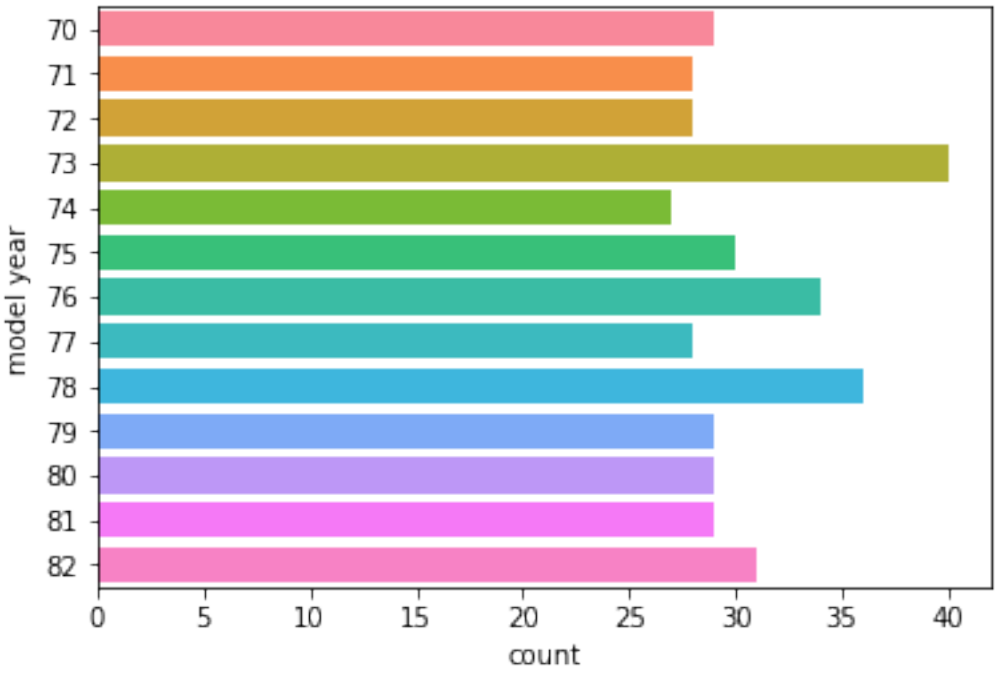
```
Out[21]: <matplotlib.axes._subplots.AxesSubplot at 0x1e2179d2f60>
```



It is also possible to display the bars horizontally.

```
In [23]: sns.countplot(y=raw_mpg["model year"], saturation=1)
```

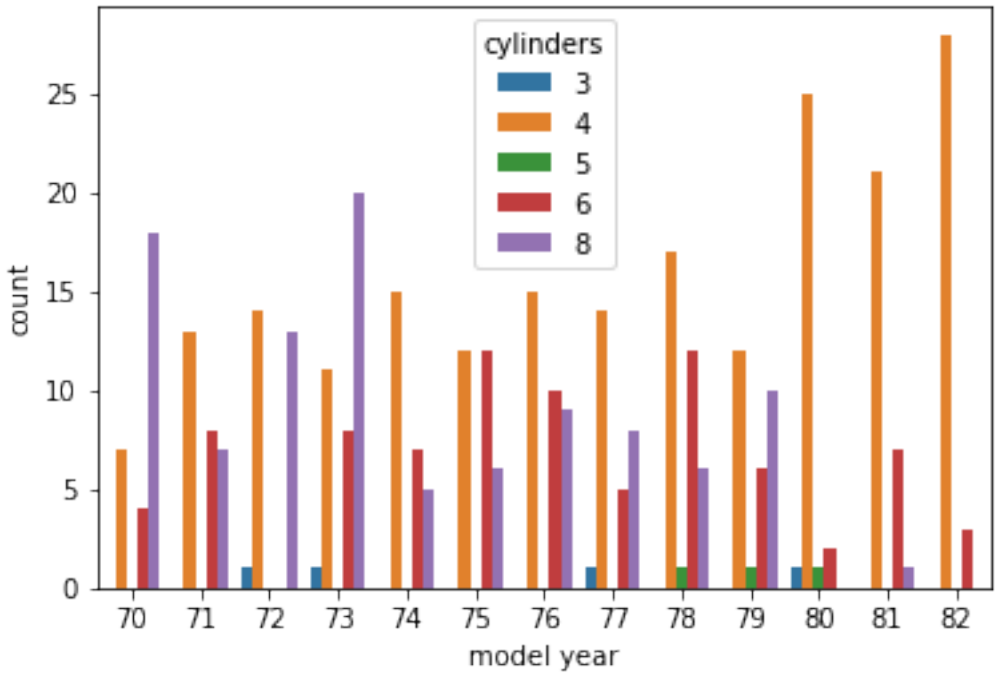
```
Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x1e217b26cc0>
```



We may want to display the counts of cars with different cylinders along with the model year by passing data to the `hue` argument. Now we enter the arguments in a different manner, passing the column name to `x` and the data frame to the `data` argument. The cylinders are passed to the `hue` keyword.

```
In [24]: sns.countplot(x="model year", hue="cylinders", data=raw_mpg, )
```

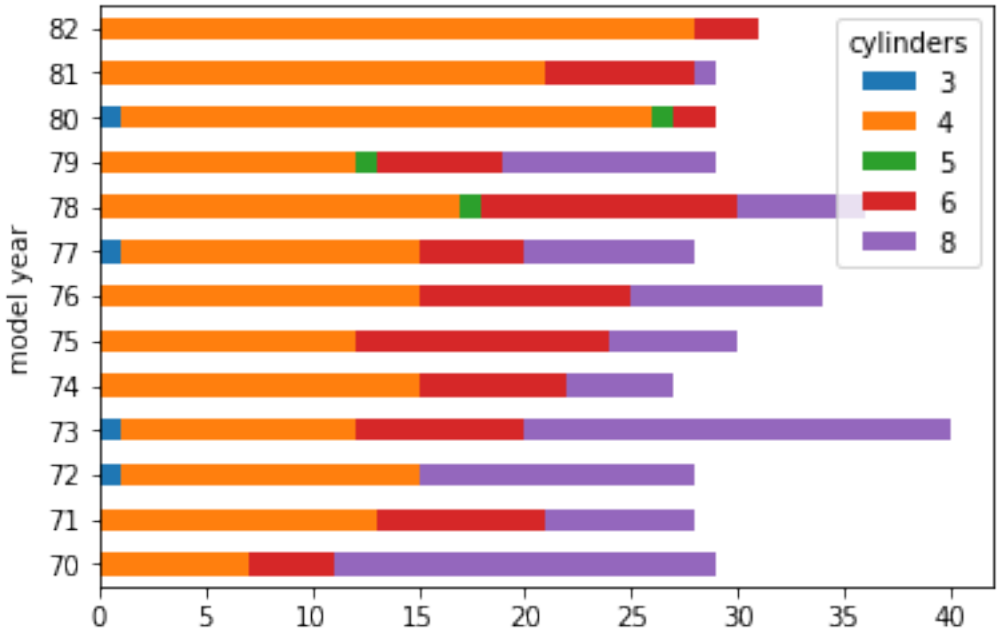
```
Out[24]: <matplotlib.axes._subplots.AxesSubplot at 0x1e217bb88d0>
```



The default settings for such a plot is not really beautiful. So let's change it into a **stacked** barplot, orient it horizontally, add a title and move the position of the legend to outside the plot axis.

```
In [34]: pd.crosstab(index=raw_mpg['model year'], columns=raw_mpg['cylinders'])\
             .plot(kind='barh', stacked=True)
```

```
Out[34]: <matplotlib.axes._subplots.AxesSubplot at 0x1e21a0f6e48>
```



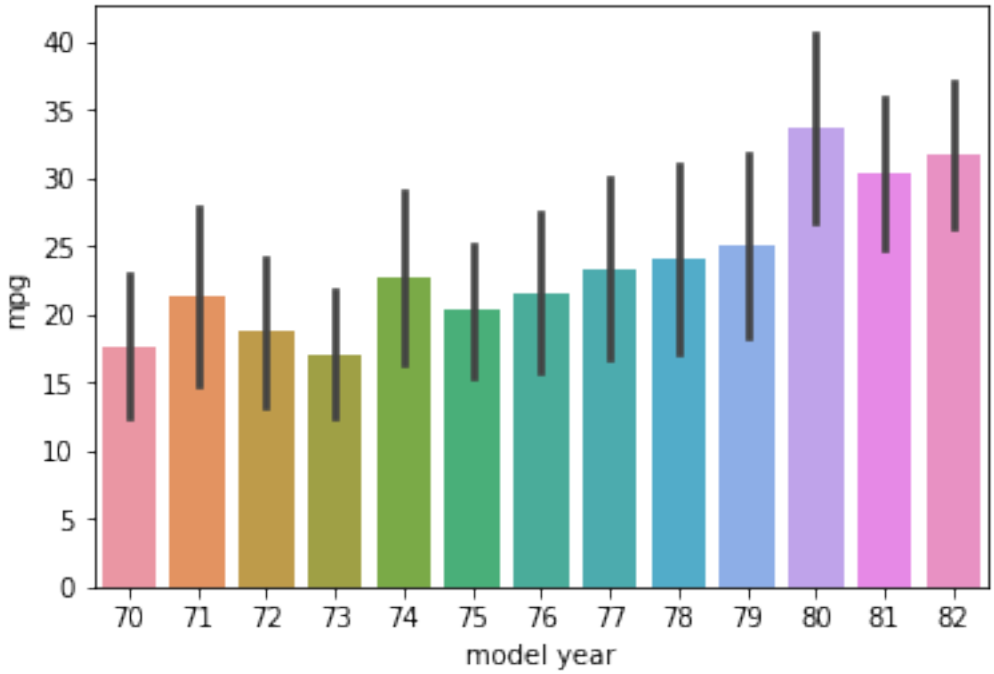
This tells us that the newer cars have 4 cylinders while older cars tend to have 8 cylinder engines.

5.2.1 Visualising aggregate data

How do we visualize the average miles per gallon for cars for a given model year? Note that this involves two variables, miles per gallon and the model year. For this we need `barplot`.

```
In [36]: sns.barplot(x="model year", y="mpg", data=raw_mpg, estimator=np.mean, ci="sd")
```

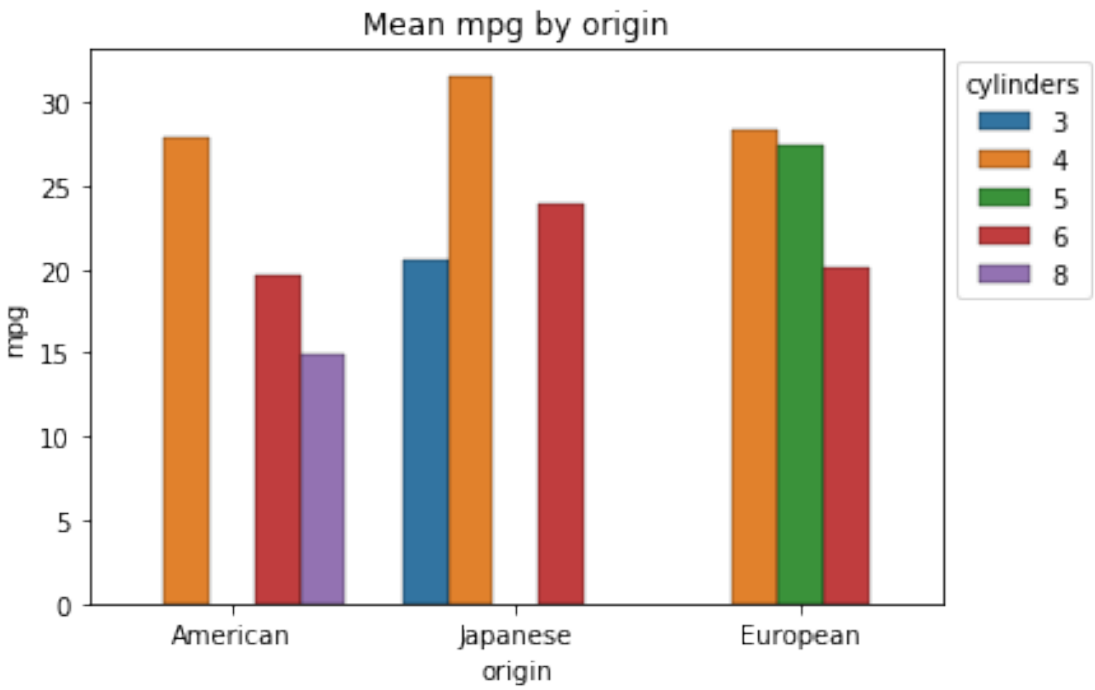
```
Out[36]: <matplotlib.axes._subplots.AxesSubplot at 0x1e21a3452b0>
```



The black bars indicate the standard deviation which gives a general idea of how spread out the mpg values are for a given model year. The option `estimator=count` is equivalent to `countplot`.

Let's now finish with a chart showing the average mpg for place of origin. Before that, we note that orgin is numerically coded. So let's fill in the actual locations corresponding to the code (1=American, 2=European, 3=Japanese)

```
In [37]: raw_mpg["origin"] = raw_mpg["origin"]\
             .map({1: "American", 2:"European", 3:"Japanese"})

In [38]: g = sns.barplot(hue="cylinders", y="mpg",x="origin", data=raw_mpg,
                         ci=None, estimator=np.mean,
                         lw=0.3, ec="black")

         g.legend(title="cylinders", loc="upper left", bbox_to_anchor=(1,1))
         g.set_title("Mean mpg by origin")

Out[38]: Text(0.5,1,'Mean mpg by origin')
```

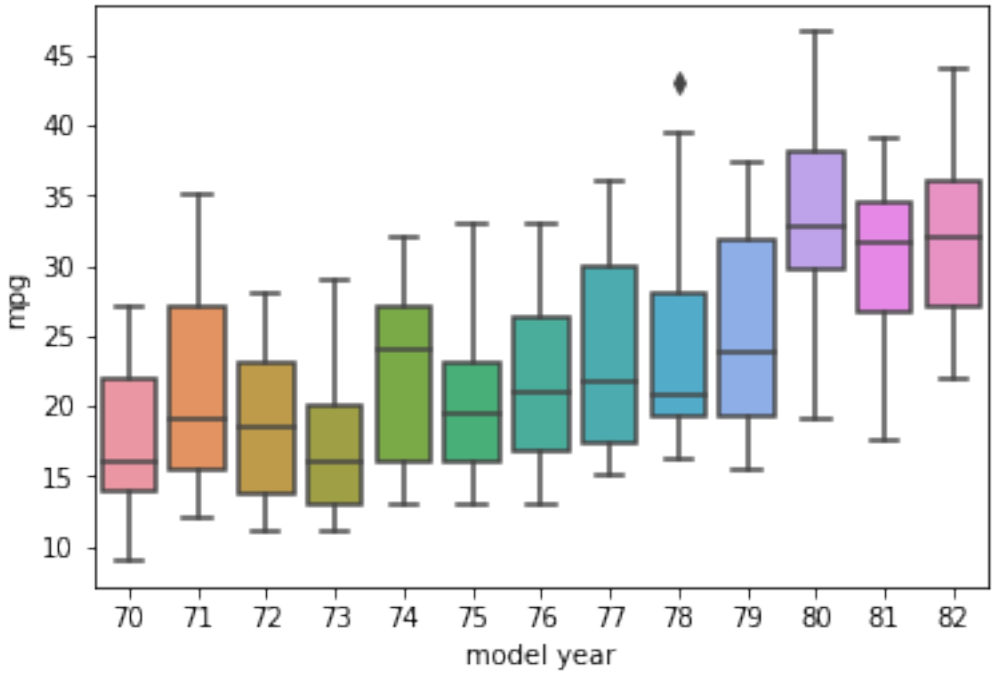


which shows that Japanese cars are generally more fuel efficient.

5.3 Generating boxplots

Box and whisker charts a good way to compare distribution of data between different groups. The function to do so is `boxplot`. Let us compare the distribution of the mpg for cars from different years

```
In [39]: sns.boxplot(x="model year", y="mpg", data=raw_mpg)

Out[39]: <matplotlib.axes._subplots.AxesSubplot at 0x1e21a4b32b0>
```

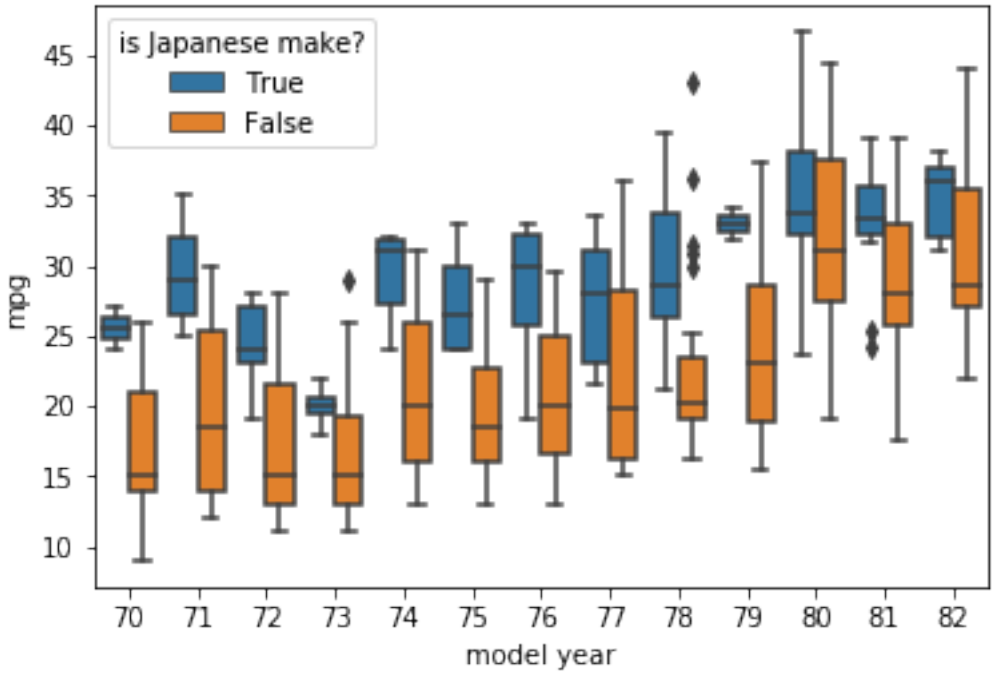


Notice the presence of a single outlier for cars made in 1978. Notice a general upward trend in the general efficiency of cars in the early 80s.

Let us now compare the rising mileage between Japanese and non-Japanese cars.

```
In [42]: raw_mpg["is Japanese make?"] = raw_mpg["origin"].isin(["Japanese"])

         sns.boxplot(x="model year", y="mpg", hue="is Japanese make?", data=raw_mpg, hue_order=[
                     dodge=True)
```

```
Out[42]: <matplotlib.axes._subplots.AxesSubplot at 0x1e21ba5ab70>
```

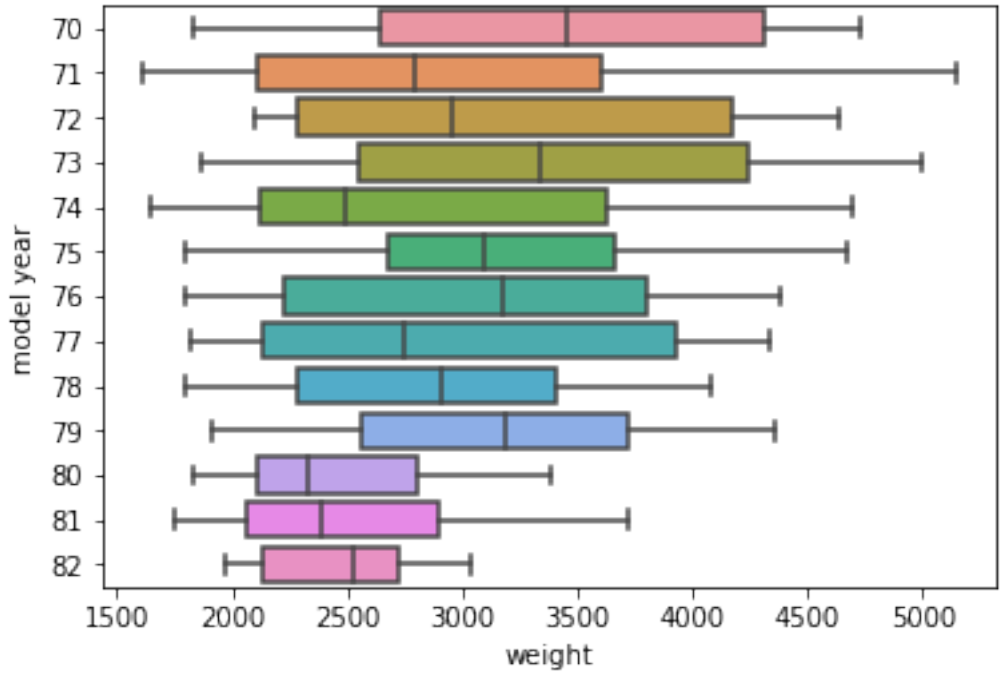


Note that Japanese cars seem to be generally more efficient in the 70s throughout to late 70s. But American and European cars do catch up in the early 80s.

We can view boxplots horizontally as well. We do this to compare horsepower of the engines throughout the years. Perhaps the rise in mileage is due to decreased weight of the vehicle?

```
In [43]: sns.boxplot(x="weight", y="model year", data=raw_mpg, orient="h")
```

```
Out[43]: <matplotlib.axes._subplots.AxesSubplot at 0x1e21bc95320>
```



We needed to specify the `orient` argument as `h` (for horizontal) because both the `weight` and `model year` variables are numerical. Since `model year` is plotted on the y-axis and `weight` on the x-axis, by specifying `h` for orient, we make `weight` to be the response variable. By default, seaborn assumes that anything numerical variable plotted on the `y` axis is a response variable.

We may generate boxplots using `factorplot` which is a seaborn function used to visualize relationships between categorical "factors" and a continous response. What we will try to do is to plot the interaction between `cylinders`, `model year` and `origin`.

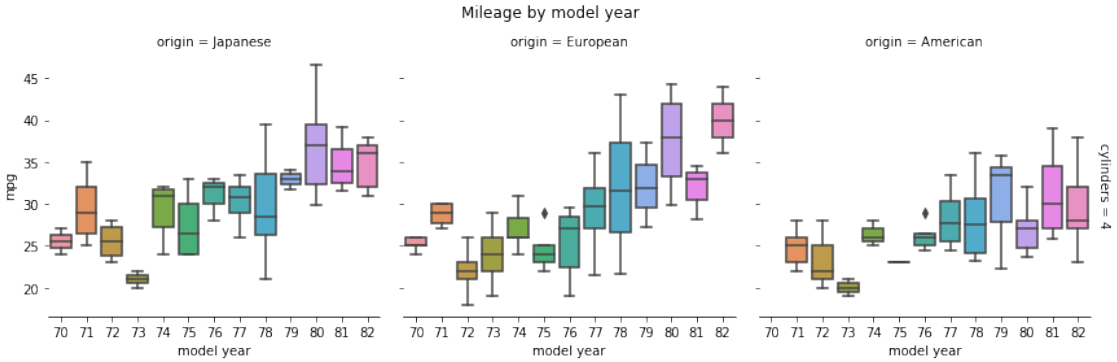
```
In [44]: pd.crosstab(index=raw_mpg.cylinders, columns=raw_mpg["origin"])
```

```
Out[44]: origin     American  European  Japanese
         cylinders
         3                 0         0         4
         4                72        63        69
         5                 0         3         0
         6                74         4         6
         8               103         0         0
```

But we will only do this for 4 cylinder cars.

```
In [46]: g = sns.factorplot(x="model year", y="mpg",
                            # Filter for entries with 4 cylinders
                            data=raw_mpg[raw_mpg.cylinders==4],
                            row="cylinders", col="origin", kind="box", margin_titles=True)
```

```
g.fig.subplots_adjust(top=0.85)
g.fig.suptitle("Mileage by model year")
g.despine(left=True);
```



6 Case Study: Historical Daily Weather Records

We usually go through the following four phases of thinking processes when working on data analytics projects:

1. Background (Business) Understanding
2. Data Understanding and Preparation
3. Descriptive and Visual Analytics
4. Export and Present Results

6.1 Background Understanding

In this case study, we shall use daily weather record at Changi weather station, from Jan-2016 to Dec-2017, to investigate the following weather pattern.

1. Rainfall pattern - Wettest day of the year - Wettest month of the year - Number of rainy days per month

2. Temperature pattern - Warmest/coolest day of the year - Warmest/coolest month of the year - Range of temperature variation

3. Relationship between rainfall and temperature.

The raw data are available on Metrological Service Singapore.

6.2 Data Understanding and Preparation

```
In [47]: # Import libraries
         import numpy as np
         import pandas as pd
         import os
         import scipy.stats as stats
```

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

Each csv file stores the weather record for a month. All the csv files are stored in a folder 'NEA_Daily'. We'll write a simple code to read all the files in the folder and join them together to form a dataframe.

```
In [48]: def get_data():
             path = os.path.join(os.getcwd(), "NEA_Daily")# Get folder path

             files = os.listdir(path) # List all files in this folder

             df = pd.DataFrame()

             # Write a for loop to read each file in the folder
             for file in files:

                 # use Python parser engine instead of the default C engine.
                 # There is a special character '\x97' which we
                 # regard as a null value
                 data = pd.read_csv(os.path.join(path, file), na_values="\x97",
                                    engine="python", encoding="latin1")

                 # append each file in a dataframe
                 df = df.append(data)

             # reset the index of the final dataframe
             df = df.reset_index(drop=True)

             return df

In [ ]: df = get_data()
        df.sample(5)
```

6.3 Descriptive and Visual Analytics

1. Rainfall Pattern

First, we get the basic numerical summaries for Daily Rainfall Total.

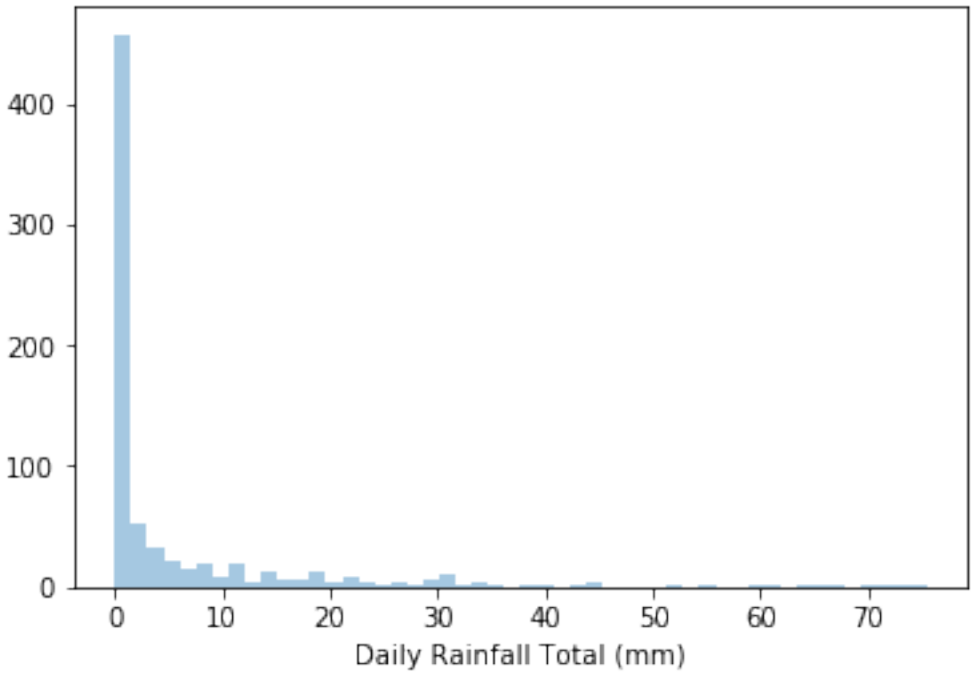
```
In [50]: df['Daily Rainfall Total (mm)'].describe()

Out[50]: count    731.000000
         mean       5.487141
         std       11.619427
         min        0.000000
         25%        0.000000
         50%        0.200000
         75%        4.800000
```

```
max       75.400000
Name: Daily Rainfall Total (mm), dtype: float64
```

We use Seaborn Distplot to visualise the distribution of rainfall.

```
In [51]: sns.distplot(df['Daily Rainfall Total (mm)'], kde=False);
```

```
C:\Users\s42355\AppData\Local\conda\conda\envs\sklearn\lib\site-packages\matplotlib\axes\_axes.p
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```



The distribution is highly skewed, indicating that high rainfall is truly a "once in a lifetime" event.

To find the monthly total rainfall we need to apply `groupby` to both `Year` and `Month` because the same numerical label for month is used for two different years. If we merely grouped by `Month` alone, we would get total rainfall in a month across two years which is not a meaningful value to obtain.

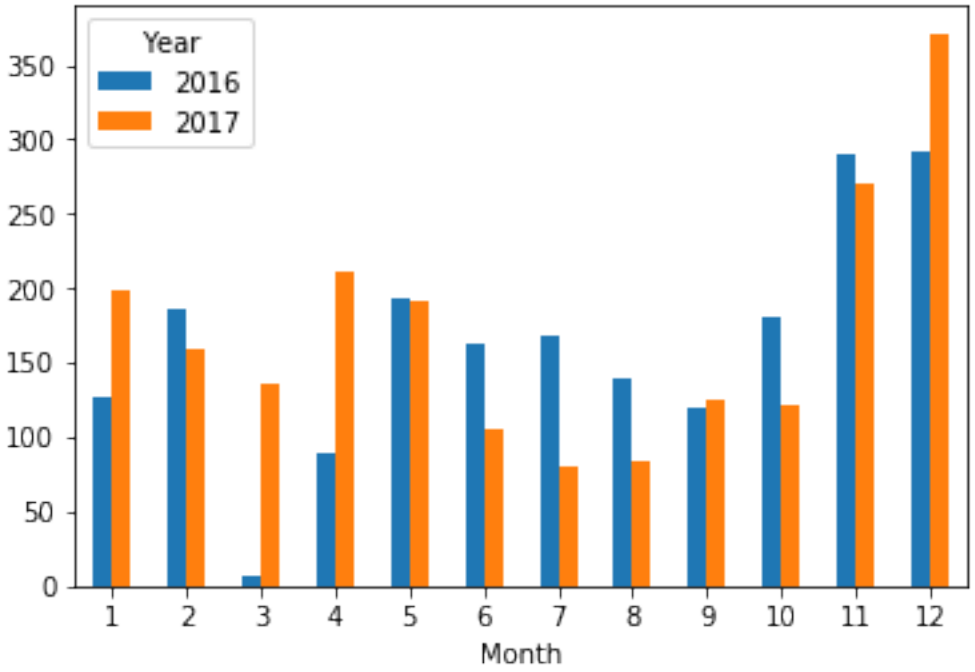
```
In [53]: tab = pd.pivot_table(df, index='Month', columns='Year',
                              values='Daily Rainfall Total (mm)',
                              aggfunc='sum')
         tab
```

```
Out[53]: Year     2016   2017
         Month
         1       127.0  197.8
```

```
2      186.0  159.2
3        6.2  136.6
4       89.8  210.4
5      193.8  191.4
6      163.2  106.0
7      168.6   80.0
8      139.2   84.2
9      118.9  125.2
10     181.0  121.6
11     290.2  271.0
12     292.6  371.2
```

```
In [55]: tab.plot(kind='bar', rot='horizontal')
```

```
Out[55]: <matplotlib.axes._subplots.AxesSubplot at 0x1e21c54e908>
```

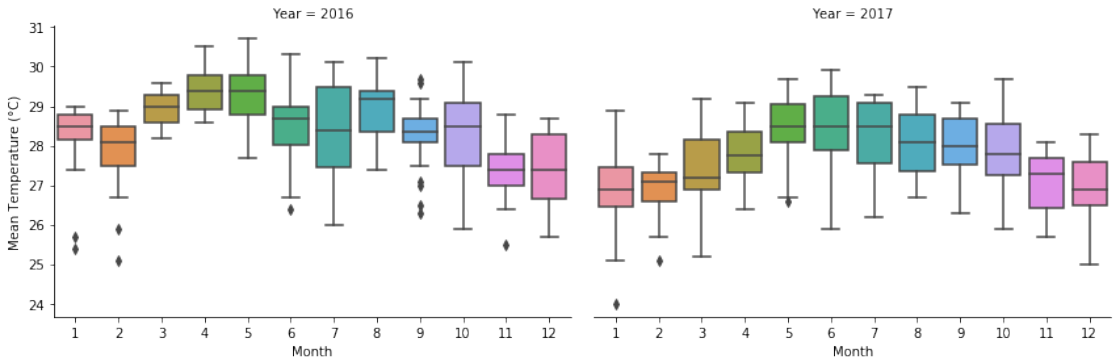


2. Temperature Pattern

Let's look at the distribution of mean daily temperature throughout the observational period.

```
In [57]: # Distribution of mean temperature by Month
         g = sns.factorplot(x="Month", y="Mean Temperature (°C)",
                            col="Year", data=df, kind="box",
                            orient="v", aspect=1.5)

         # More detailed customization
         right_subplot = g.axes.ravel()[1]
```

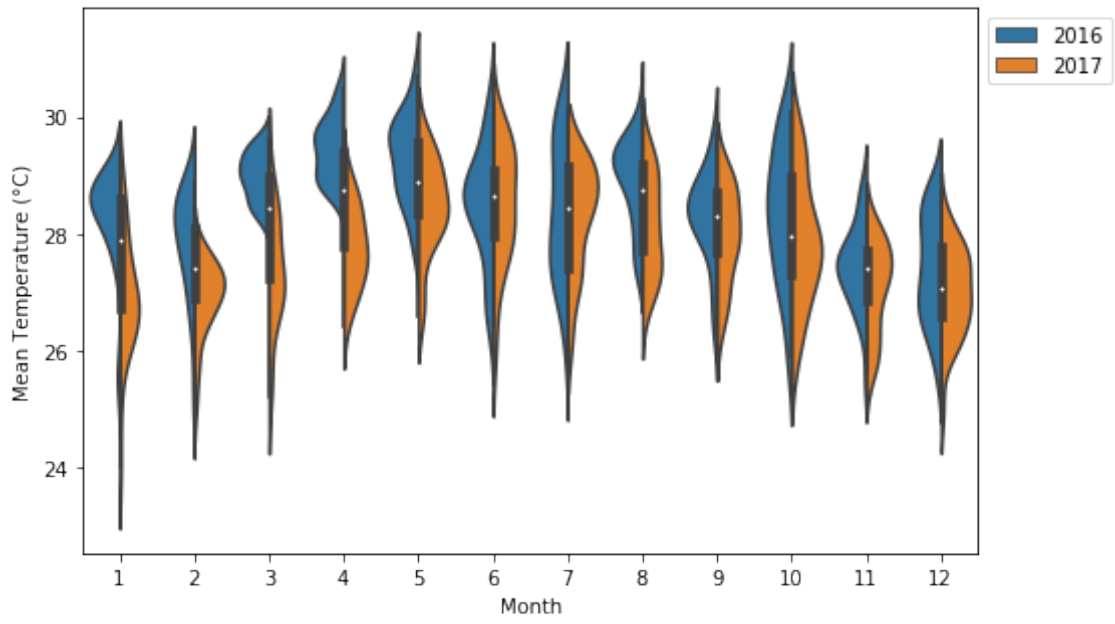
```
right_subplot.spines["left"].set_visible(False)
right_subplot.tick_params(left=False )
```



A direct side by side comparison of monthly temperature distribution over the two years may be meaningful. We will use a `violinplot` for this purpose.

In [58]:
```
# Distribution of mean temperature by Month and Year using subplots
plt.figure(figsize=(8,5))
sns.violinplot(x="Month", hue="Year", y="Mean Temperature (°C)", data=df, split=True )
plt.legend(bbox_to_anchor=(1,1), loc="upper left")
```

Out[58]:
```
<matplotlib.legend.Legend at 0x1e21dc1cac8>
```



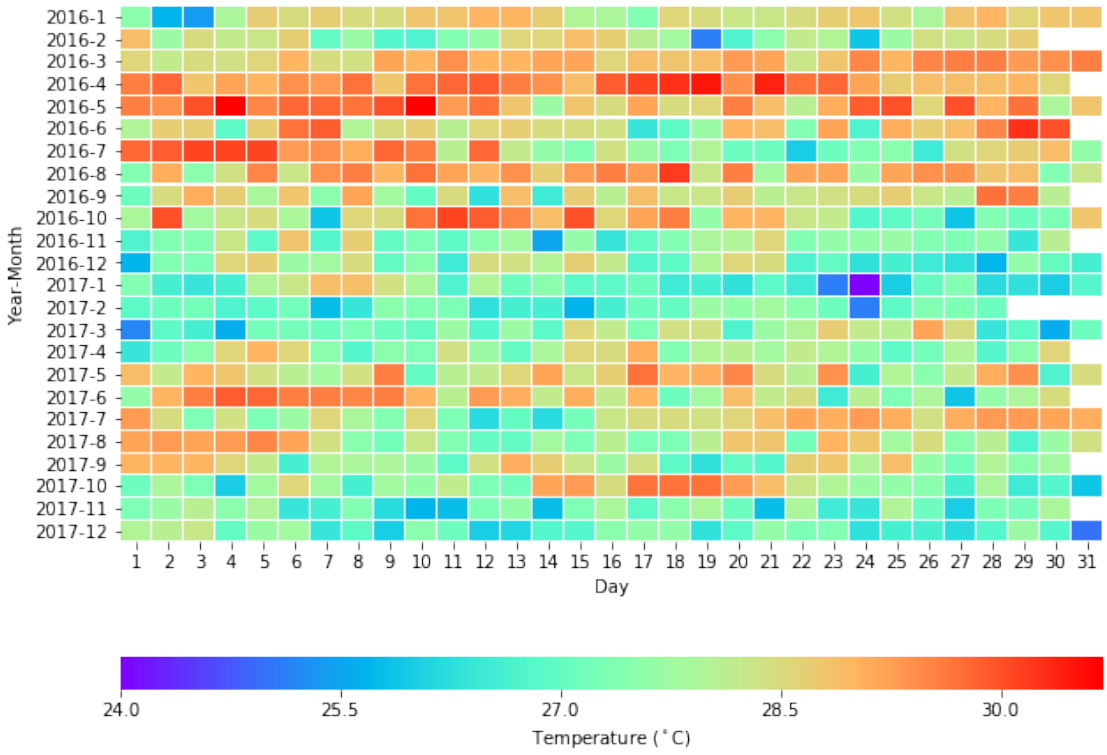
More evidence showing that 2017 was a colder year compared to 2016.

3. Heatmaps

Heatmaps are a good way to give extra emphasis to tabular data. This allows comparison across two dimensions easily.

```
In [59]: ymd = df.pivot_table(values="Mean Temperature (°C)", index="Day",
                              columns=["Year", "Month"])
```

```
In [64]: # Daily mean temperature using heatmap
         plt.figure(figsize=(10,8))
         g = sns.heatmap(ymd.T, cmap="rainbow", square=False, cbar_kws={"use_gridspec": False,
                                "location": "bottom",
                                "label": "Temperature ($^\circ$C)",
                                "aspect": 30,
                                },
                         linewidths=0.1)
```



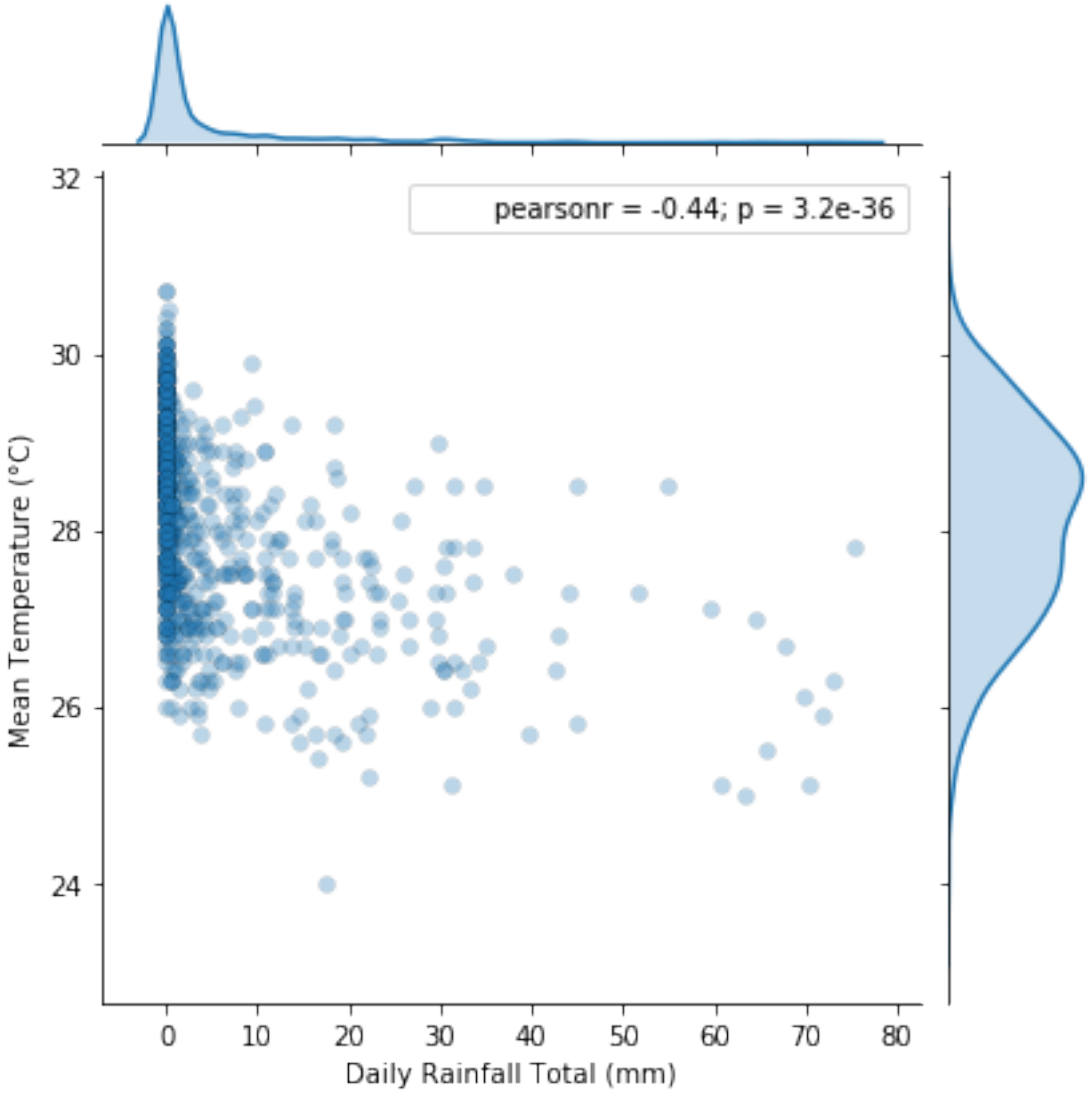
It notable that 2016 seems to be the warmer year on record.

4. Relationship between Rainfall and Temperature

Rainy days tend to be cooler days. Let's investigate the data and see whether this is true.

```
In [65]: # Visualising two numerical variables using jointplot with marginal distribution
         sns.jointplot(x='Daily Rainfall Total (mm)', y='Mean Temperature (°C)',
                       data=df,
                       kind='scatter',
```

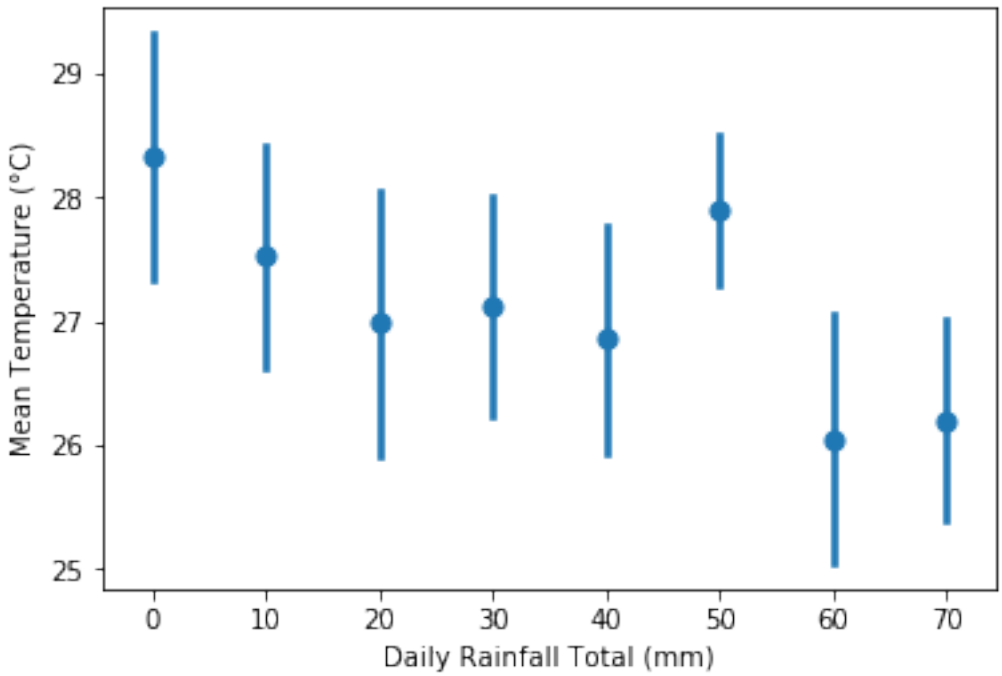
```
                    marginal_kws={"kde":True, "hist":False, "kde_kws": {"shade": True}},
                    alpha=0.3, edgecolors="k", linewidths=0.2);
```



Sometimes binning data may improve our understanding of the relationship between the two variables.

```
In [66]: sns.regplot(x="Daily Rainfall Total (mm)", y='Mean Temperature (°C)', data=df,
                     x_bins=np.arange(0, 80, 10), x_ci="sd", fit_reg=False)
```

```
Out[66]: <matplotlib.axes._subplots.AxesSubplot at 0x1e21c59efd0>
```



We notice that there is an anomaly at the 50mm bin. If we model the mean temperature as a linear function of total rainfall, we want to use a robust estimation against outliers.

So we can conclude that a wetter day does correspond to a colder day.

6.4 Export and Present Results

We can export the summarised statistics to Excel file and the charts as png file.

```
In [ ]: # Save the new dataframe to Excel file
        writer = pd.ExcelWriter('NEA_Cleaned_Data.xlsx')
        df.to_excel(writer,'Sheet1')
        writer.save()

        # Saving a chart to png file
        plt.subplots(figsize=(10,6))
        g = sns.boxplot(x='Month',y='Mean Temperature (°C)',data=df, hue='Year')
        g.set_title('Distribution of Mean Temperature (°C)');
        plt.savefig('Distribution_of_Mean_Temperature',dpi=200,bbox_inches='tight')
```