Python and ML Bootcamp - Day 2

Evangelia Drakopoulou

PhD students: Dimitris Stavropoulos
Vasilis Tsourapis
George Zarpapis

Konstantinos Paschos
Leda Liogka

Useful Packages

- **Numpy**: NumPy is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

- **Pandas**: Pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series.

How to call them in you script?

```
1 import numpy as np
2 import pandas as pd
```

Numpy

Numpy

- One way we can initialise NumPy arrays is from Python lists, using nested lists for two- or higher-dimensional data.

```
c = np.array([1, 2, 3, 4, 5, 6])
```

or

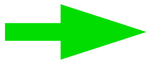
```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

```
print(a[0])
```

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

To create a NumPy array, you can use the function `np.array()`.

```
import numpy as np
b = np.array([1, 2, 3])
```



```
print(b)
[1 2 3]
```

Check its shape: `b.shape`

```
(3,)
```

The `shape` of the array is a tuple of integers giving the size of the array along each dimension.

Task: What's the shape of **a**?

Numpy

- One way we can initialise NumPy arrays is from Python lists, using nested lists for two- or higher-dimensional data.

```
c = np.array([1, 2, 3, 4, 5, 6])
```

or

```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

```
print(a[0])
```

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

To create a NumPy array, you can use the function `np.array()`.

```
import numpy as np
b = np.array([1, 2, 3])
```

```
print(b)
[1 2 3]
```

Check its shape: `b.shape`

```
(3,)
```

The `shape` of the array is a tuple of integers giving the size of the array along each dimension.

Task: What's the shape of **a**?

Answer: `(3, 4)`

Numpy

```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

We can access the elements in the array using square brackets. If you want to access the first element in your array, you'll be accessing element "0".

```
print(a[0])
```
→ `[1 2 3 4]`

Create an array filled with 0's: `np.zeros(2)` → `array([0., 0.])`

an array filled with 1's: `np.ones(2)` → `array([1., 1.])`

an empty array! `np.empty(2)` → `array([3.14, 42.  ])`

The function `empty` creates an array whose initial content is random and depends on the state of the memory. The reason to use `empty` over `zeros` (or something similar) is speed - just make sure to fill every element afterwards!

Create an array with a range of elements: `np.arange(4)` → `array([0, 1, 2, 3])`

Numpy - Adding, removing, and sorting elements

Sorting an element is simple with `np.sort()`. You can specify the axis, kind, and order when you call the function.

```
arr = np.array([2, 1, 5, 3, 7, 4, 6, 8])
```

```
np.sort(arr)
```

```
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Concatenate arrays: `a = np.array([1, 2, 3, 4])` `b = np.array([5, 6, 7, 8])`

```
np.concatenate((a, b))
```

```
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Task: How can you concatenate these arrays?

(i) `x = np.array([[1, 2], [3, 4]])` `y1 = np.array([[5, 6]])`

(ii) `x = np.array([[1, 2], [3, 4]])` `y2 = np.array([7, 8])`

Numpy - Adding, removing, and sorting elements

Concatenate arrays: `a = np.array([1, 2, 3, 4])` `b = np.array([5, 6, 7, 8])`

```
np.concatenate((a, b))
```
→
```
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Task: How can you concatenate these arrays?

(i)
```
x = np.array([[1, 2], [3, 4]])
y1 = np.array([[5, 6]])
```

(ii)
```
x = np.array([[1, 2], [3, 4]])
y2 = np.array([7, 8])
```

Answer:

(i)
```
np.concatenate((x, y1))
```
→
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

(ii)
```
np.concatenate((x, y2))
```
→
```
ValueError: all the input arrays must have same number of dimensions, but the array at index 0 has 2 dimension(s) and the array at index 1 has 1 dimension(s)
```

Numpy - Find the shape and size of an array

`ndarray.ndim` will tell you the number of axes, or dimensions, of the array.

`ndarray.size` will tell you the total number of elements of the array. This is the *product* of the elements of the array's shape.

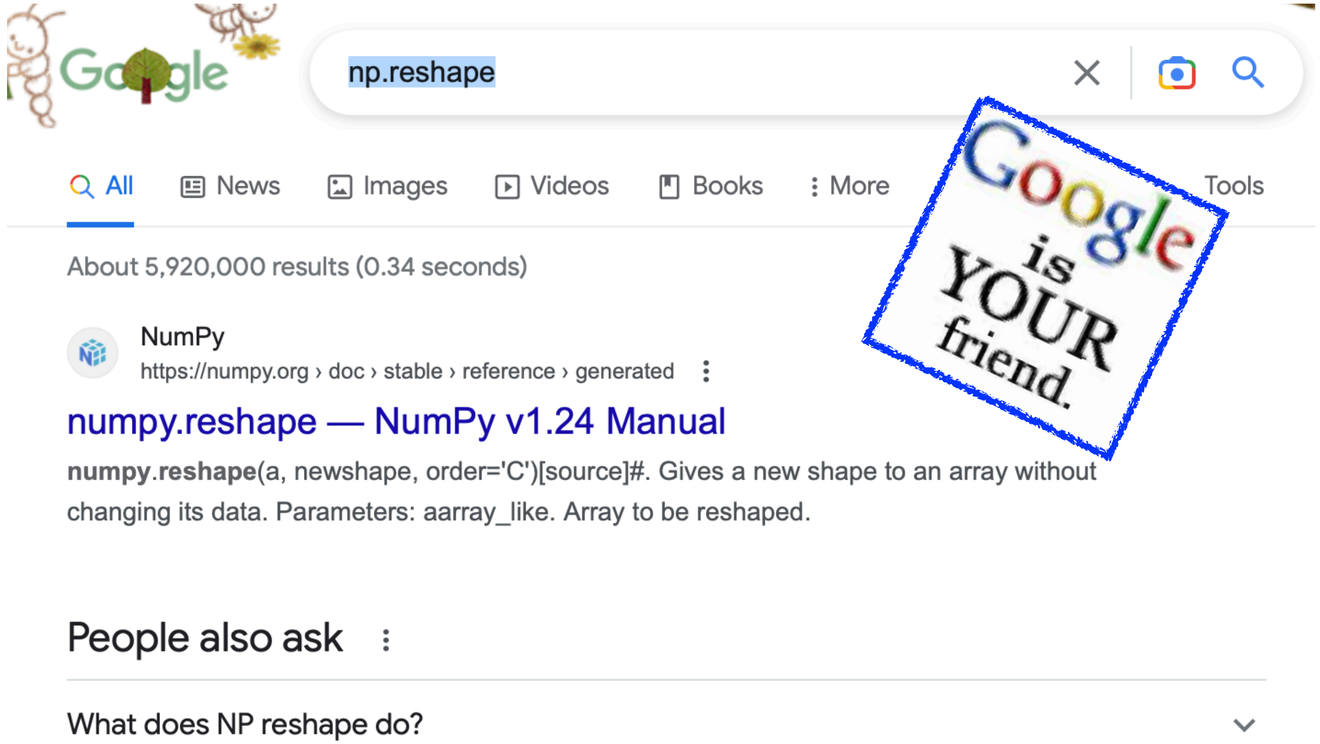
`ndarray.shape` will display a tuple of integers that indicate the number of elements stored along each dimension of the array. If, for example, you have a 2-D array with 2 rows and 3 columns, the shape of your array is `(2, 3)`.

```
array_example = np.array([[0, 1, 2, 3],[4, 5, 6, 7]])
```

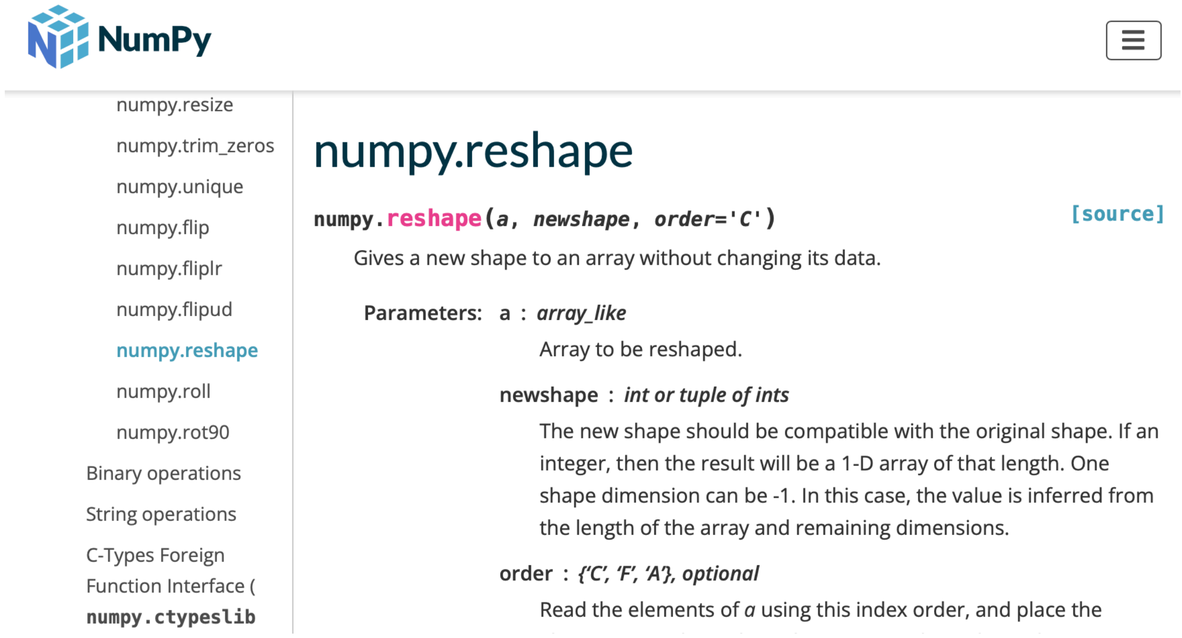
```
array_example.ndim   → 2

array_example.size   → 8

                       rows  columns
array_example.shape  → (2, 4)
```

```
array_example.shape[0] → 2        array_example.shape[1] → 4
```

Numpy - reshape



https://numpy.org/doc/stable/reference/generated/numpy.reshape.html




```
array_example = np.array([[0, 1, 2, 3],[4, 5, 6, 7]])
```

rows columns

```
array_example.shape  →  (2, 4)
```

```
>>> array_example.reshape(4,2)
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

Numpy - reshape

Examples

```
array_example = np.array([[0, 1, 2, 3],[4, 5, 6, 7]])
```

```
>>> array_example.reshape(2,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot reshape array of size 8 into shape (2,3)
>>> array_example.reshape(1,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot reshape array of size 8 into shape (1,2)
>>> array_example.reshape(8,1)
array([[0],
       [1],
       [2],
       [3],
       [4],
       [5],
       [6],
       [7]])
>>> array_example.reshape(1,8)
array([[0, 1, 2, 3, 4, 5, 6, 7]])
>>> array_example.shape
(2, 4)
```

```
>>> a1 = array_example.reshape(1,8)
>>> a1.shape
(1, 8)
```

Numpy - Reshaping and flattening multidimensional arrays

There are two ways to flatten an array: `.flatten()` and `.ravel()`. The primary difference between the two is that the new array created using `ravel()` is actually a reference to the parent array (i.e., a "view"). This means that any changes to the new array will affect the parent array as well. Since `ravel` does not create a copy, it's memory efficient.

```
x = np.array([[1 , 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

`x.flatten()`  *array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])*

When you use `flatten`, changes to your new array won't change the parent array.

Example:

```
a1 = x.flatten()
a1[0] = 99
print(x) # Original array
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
print(a1) # New array
[99  2  3  4  5  6  7  8  9 10 11 12]
```

```
a2 = x.ravel()
a2[0] = 98
print(x) # Original array
[[98  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
print(a2)   # New array
[98  2  3  4  5  6  7  8  9 10 11 12]
```

Numpy - Splitting arrays

numpy.split(array, indices_or_sections, axis=0)

Split an array into multiple sub-arrays as views into *array*.

```
x = np.arange(9.0)
```

numpy.arange([*start*,]*stop*, [*step*,]*dtype=None*, *, *like=None*)
Return evenly spaced values within a given interval.

```
print(x)
```
→
```
[0. 1. 2. 3. 4. 5. 6. 7. 8.]
```

```
np.split(x, 3)
```
→
```
[array([0., 1., 2.]), array([3., 4., 5.]),
array([6., 7., 8.])]
```

```
np.split(x, [3, 5, 6])
```
→
```
[array([0., 1., 2.]), array([3., 4.]),
array([5.]), array([6., 7., 8.])]
```

Task: Create an array with numbers from 3 to 12. Then, split it into 4 array [3,5), [5,9], [10], [11,12]

Numpy - Splitting arrays

numpy.split(array, indices_or_sections, axis=0)

Task: Create an array with numbers from 3 to 12. Then, split it into 4 arrays:

[3,5), [5,9], [10], [11,12]

Answer:

```
x = np.arange(3.0,13.0)
```

```
print(x)
```

```
[ 3.  4.  5.  6.  7.  8.  9. 10. 11. 12.]
```

```
np.split(x, [2,7,8])
```

```
[array([3., 4.]), array([5., 6., 7., 8., 9.]), array([10.]),
array([11., 12.])]
```

Numpy - Slicing and Subsetting

```
x = np.array([[1 , 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

Operator	Description
array[i]	1d array at index i
array[i,j]	2d array at index[i][j]
array[i]<4	Boolean Indexing, see Tricks
array[0:3]	Select items of index 0, 1 and 2
array[0:2,1]	Select items of rows 0 and 1 at column 1
array[:1]	Select items of row 0 (equals array[0:1, :])
array[1:2, :]	Select items of row 1

Examples:

```
>>> x[0]
array([1, 2, 3, 4])
>>> x[0][1]
2
>>> x[0,1]
2
>>> x[2,3]
12
```

```
>>> x
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> x[0]<2
array([ True, False, False, False])
>>> x[0,3]<2
False
>>> x[2]<5
array([False, False, False, False])
>>> x[0]<5
array([ True,  True,  True,  True])
>>> x[0:1,2]
array([3])
>>> x[0:1]
array([[1, 2, 3, 4]])
>>> x[0:2]
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
>>> x[0:2,3]
array([4, 8])
```

Numpy - Basic Statistics

```
x = np.array([[1 , 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

Operator	Description
np.mean(array)	Mean
np.median(array)	Median
np.std(array)	Standard Deviation
array.sum()	Array-wise sum
array.min()	Array-wise minimum value
array.max(axis=0)	Maximum value of specified axis
array.cumsum(axis=0)	Cumulative sum of specified axis

Examples:

```
>>> np.mean(x)
6.5
>>> np.median(x)
6.5
>>> x.sum()
78
>>> x.min()
1
>>> x.max()
12
>>> x.cumsum(0)
array([[ 1,  2,  3,  4],
       [ 6,  8, 10, 12],
       [15, 18, 21, 24]])
>>> x.cumsum()
array([ 1,  3,  6, 10, 15, 21, 28, 36, 45, 55, 66, 78])
>>> x.cumsum(axis=1)
array([[ 1,  3,  6, 10],
       [ 5, 11, 18, 26],
       [ 9, 19, 30, 42]])
```

Hands On

Task: **(i)** Write a NumPy program to calculate the difference between the maximum and the minimum values of a given array along the second axis.

Hint:

Use array: x = np.arange(12) and reshape it to have **2 rows** and **6 columns**. Check its shape. Then calculate and print the difference.

(ii) Create a script (libnumpy_handsOn.py) with a function diff_arr that takes the array, the number of rows and the numbers of columns as arguments and returns the calculated difference. Then call it from another script and print the result.

(iii) Debug this script: BUGnumpy_handsOn.py [https://github.com/edrakopo/pythonMLBootcamp/blob/master/Day2/BUGnumpy_handsOn.py]

Hands On

Task: **(i)** Write a NumPy program to calculate the difference between the maximum and the minimum values of a given array along the second axis.

Answer:

```
import numpy as np

x = np.arange(12)
print(x)
x1 = x.reshape((2, 6))
print("\nOriginal array:")
print(x1)
print(x1.shape)
print("x1.max(axis=1): ",x1.max(axis=1))
print("x1.min(axis=1): ", x1.min(axis=1))

diff= x1.max(axis=1) - x1.min(axis=1)

print("\nDifference between the maximum and the minimum values of the said array:")
print(diff)
```

Hands On

Task: **(ii)** Create a script (libnumpy_handsOn.py) with a function diff_arr that takes the array, the number of rows and the numbers of columns as arguments and returns the calculated difference. Then call it from another script and print the result.

Answer:

libnumpy_handsOn.py

```
import numpy as np

def diff_arr(x, i, j):
    x1 = x.reshape((i,j))
    diff= x1.max(axis=1) - x1.min(axis=1)
    return diff
```

```
import numpy as np
import libnumpy_handsOn as lb

x = np.arange(12)
print(x)

diff = lb.diff_arr(x,2,6)
print("\nDifference between the maximum and the minimum values of the said array:")
print(diff)
```

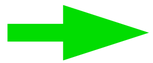
Hands On

Task: **(iii)** Debug this script: BUGnumpy_handsOn.py

Answer:

1

```
python3 BUGnumpy_handsOn.py
  File "BUGnumpy_handsOn.py", line 3
    def diff_arr(x, i, j)
                         ^
SyntaxError: expected ':'
```



```
1 import numpy as np
2
3 def diff_arr(x, i, j):
4     x1 = x.reshape((i,j))
```

2

```
 File "BUGnumpy_handsOn.py", line 10
    x = np.arange(12)
                    ^
IndentationError: unindent does not match any outer indentation level
```

```
9 #use the function:
10 x = np.arange(12)
```

same for lines: 11, 13, 14, 15

Hands On

Task: **(iii)** Debug this script: BUGnumpy_handsOn.py

Answer:

3

```
 python3 BUGnumpy_handsOn.py
[ 0  1  2  3  4  5  6  7  8  9 10 11]
Traceback (most recent call last):
  File "BUGnumpy_handsOn.py", line 13, in <module>
    diff = lb.diff_arr(x,2,6)
NameError: name 'lb' is not defined
```

```
13 diff = diff_arr(x,2,6)
```

4

```
Traceback (most recent call last):
  File "BUGnumpy_handsOn.py", line 13, in <module>
    diff = diff_arr(x,2,6)
  File "BUGnumpy_handsOn.py", line 5, in diff_arr
    print(xl)
NameError: name 'xl' is not defined. Did you mean: 'x'?
```

```
4       x1 = x.reshape((i,j))
5       print(x1)
```

Pandas

Pandas

```
import pandas as pd
```

Creating data: There are two core objects in pandas: the **DataFrame** and the **Series**.

DataFrame: A DataFrame is a table. It contains an array of individual entries, each of which has a certain value. Each entry corresponds to a row (or record) and a column.

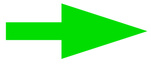
```
pd.DataFrame({'Yes': [50, 21], 'No': [131, 2]})
```

	Yes	No
0	50	131
1	21	2

index

Series: A Series, by contrast, is a sequence of data values. If a DataFrame is a table, a Series is a list. A Series is, in essence, a single column of a DataFrame.

```
pd.Series([1, 2, 3, 4, 5])
```



0	**1**
1	2
2	3
3	4
4	5

```
dtype: int64
```

Pandas - Creating Dataframe (II)

DataFrame entries are not limited to integers. For instance, here's a DataFrame whose values are strings:

```
pd.DataFrame({'Bob': ['I liked it.', 'It was awful.'], 'Sue': ['Pretty good.', 'Bland.']})
```

```
             Bob            Sue
0    I liked it.  Pretty good.
1  It was awful.         Bland.
```

The dictionary-list constructor assigns values to the column labels, but just uses an ascending count from 0 (0, 1, 2, 3, ...) for the row labels. Sometimes this is OK, but oftentimes we will want to assign these labels ourselves.

The list of row labels used in a DataFrame is known as an **Index**. We can assign values to it by using an `index` parameter in our constructor:

```
>>> pd.DataFrame({'Bob': ['I liked it.', 'It was awful.'],
...                'Sue': ['Pretty good.', 'Bland.']},
...               index=['Product A', 'Product B'])
```

```
                 Bob            Sue
Product A    I liked it.  Pretty good.
Product B  It was awful.         Bland.
```

Pandas - Reading data files

Data can be stored in any of a number of different forms and formats. By far the most basic of these is the humble CSV file.

When you open a CSV file you get something that looks like this:

```
Product A,Product B,Product C,
30,21,9,
35,34,1,
41,11,11
```

So a CSV file is a table of values separated by commas. Hence the name: "Comma-Separated Values", or CSV.

Use **pd.read_csv()** to read a .csv file.

```
df = pd.read_csv("../path_to/filename.csv")
```

Stores the data in a dataframe: **df**

`df.shape` → to check the number of rows, columns

`df.head()` → to print the 5 first rows

`df.tail()` → to print the 5 last rows

Pandas - Write dataframe to csv

pandas.DataFrame.to_csv

Write object to a comma-separated values (csv) file.

Example:

```
import pandas as pd

df = pd.DataFrame({'Yes': [50, 21], 'No': [131, 2]})
df
   Yes   No
0   50  131
1   21    2
```

df.to_csv("output.csv") Writes the elements of dataframe: **df** to csv **output.csv**

If you want to save the file in a different path e.g. Desktop, type:

```
df.to_csv("~/Desktop/output.csv")
```

Pandas - Create/Read/Write

Task: Create the following data frame.

		Cows	Goats	Horses
Year	1	12	22	4
Year	2	20	19	7
Year	3	30	5	10
Year	4	35	25	15

Save it in *animals.csv* at your Desktop.

Read *animals.csv* .

Check the dataframe shape and print its 5 first rows.

Pandas - Create/Read/Write

Answer:

```
>>> import pandas as pd
>>> animals = pd.DataFrame({'Cows': [12, 20, 30, 35], 'Goats': [22, 19, 5, 25], 'Horses':[4, 7, 10, 15]}, index=['Year 1', 'Year 2', 'Year 3', 'Year 4'])
>>> animals
        Cows  Goats  Horses
Year 1    12     22       4
Year 2    20     19       7
Year 3    30      5      10
Year 4    35     25      15
>>> animals.to_csv("~/Desktop/animals.csv")
>>> animals = pd.read_csv("~/Desktop/animals.csv")
>>> animals.shape
(4, 4)
>>> animals.head()
```

	Unnamed: 0	Cows	Goats	Horses
0	Year 1	12	22	4
1	Year 2	20	19	7
2	Year 3	30	5	10
3	Year 4	35	25	15

Pandas - Create/Read/Write

Answer 2:

```
>>> import pandas as pd
>>> animals = pd.DataFrame({'Cows': [12, 20, 30, 35], 'Goats': [22, 19, 5, 25], 'Horses':[4, 7, 10, 15]}, index=['Year 1', 'Year 2', 'Year 3', 'Year 4'])
>>> animals
        Cows  Goats  Horses
Year 1    12     22       4
Year 2    20     19       7
Year 3    30      5      10
Year 4    35     25      15
>>> animals.to_csv("~/Desktop/animals.csv",index=False)
>>> animals = pd.read_csv(“~/Desktop/animals.csv”, index_col=0)
>>> animals.shape
(4, 4)
>>> animals.head()
```

Unnamed: 0.1	Cows	Goats	Horses
Year 1	12	22	4
Year 2	20	19	7
Year 3	30	5	10
Year_4	35	25	15

Pandas - Indexing, Selecting & Assigning

	Cows	Goats	Horses
0	12	22	4
1	20	19	7
2	30	5	10
3	35	25	15

```
animals.iloc[1:3, 0]
1    20
2    30
Name: Cows, dtype: int64
```

```
>>> animals.iloc[0]
Cows      12
Goats     22
Horses     4
Name: 0, dtype: int64
```

To select the first row of data in a DataFrame

```
>>> animals.iloc[:,0]
0    12
1    20
2    30
3    35
Name: Cows, dtype: int64
```

To get a column with iloc

everything

```
>>> animals.loc[0, 'Goats']
22
```

Task: How do you select column “Horses”?

Pandas - Indexing, Selecting & Assigning

	Cows	Goats	Horses
0	12	22	4
1	20	19	7
2	30	5	10
3	35	25	15

Task: How do you select column “Horses”?

iloc is conceptually simpler than loc because it ignores the dataset's indices. When we use **iloc** we treat the dataset like a big matrix (a list of lists), one that we have to index into by position. **loc**, by contrast, uses the information in the indices to do its work. Since your dataset usually has meaningful indices, it's usually easier to do things using **loc**.

Answer: Two ways:

(a)

```
>>> animals.loc[:, 'Horses']
0     4
1     7
2    10
3    15
Name: Horses, dtype: int64
```

(b)

```
>>> animals.iloc[:,2]
0     4
1     7
2    10
3    15
Name: Horses, dtype: int64
```

Pandas - Indexing, Selecting & Assigning

```
   Cows  Goats  Horses
0    12     22       4
1    20     19       7
2    30      5      10
3    35     25      15
```

Select data satisfying specific conditions:

```
>>> animals.loc[animals.Goats > 10]
   Cows  Goats  Horses
0    12     22       4
1    20     19       7
3    35     25      15
```

Assigning data:

```
animals['Goats'] = 'white'
```

```
>>> print(animals)
   Cows  Goats  Horses
0    12  white       4
1    20  white       7
2    30  white      10
3    35  white      15
```

```
>>> animals.loc[2,"Goats"] = 14
>>> animals
   Cows  Goats  Horses
0    12     22       4
1    20     19       7
2    30     14      10
3    35     25      15
```

Pandas - “Summary” Functions

	Cows	Goats	Horses
0	12	22	4
1	20	19	7
2	30	5	10
3	35	25	15

Pandas provides many simple "summary functions” which restructure the data in some useful way.

```
>>> animals.describe()
             Cows      Goats     Horses
count    4.000000   4.000000   4.000000
mean    24.250000  17.750000   9.000000
std     10.275375   8.845903   4.690416
min     12.000000   5.000000   4.000000
25%     18.000000  15.500000   6.250000
50%     25.000000  20.500000   8.500000
75%     31.250000  22.750000  11.250000
max     35.000000  25.000000  15.000000
```

To see the mean of a column:

```
>>> animals.Goats.mean()
17.75
```

To see a list of unique values we can use the unique() function

```
>>> animals.Goats.unique()
array([22, 19,  5, 25])
```

Pandas - Renaming

```
   Cows  Goats  Horses
0    12     22       4
1    20     19       7
2    30      5      10
3    35     25      15
```

The function rename() lets you change index names and/or column names.

```
>>> animals.rename(columns={'Goats':'Sheep'})
   Cows  Sheep  Horses
0    12     22       4
1    20     19       7
2    30      5      10
3    35     25      15
```

rename() also lets you rename index or column values by specifying a index or column keyword parameter, respectively.

```
>>> animals.rename(index={0:'1st', 1:'2nd'})
      Cows  Goats  Horses
1st     12     22       4
2nd     20     19       7
2       30      5      10
3       35     25      15
```

Pandas

Task: (i) Create the following data frames.

```
>>> animals
   Cows  Goats  Horses
0    12     22       4
1    20     19       7
2    30      5      10
3    35     25      15
```

```
>>> zoo_animals
   Lion  Tiger
0     4      9
1    10     11
2     5      5
3     3      2
```

(ii) Save the first one in *animals.csv* and the second one in zoo_*animals.csv* at your Desktop.

(iii) Read *animals.csv* and zoo_*animals.csv*.

(iv) Check the dataframe shape, print its 5 first rows.

(v) Find the mean value for each column.

Pandas

Answer (i-ii):

```
import pandas as pd
>>> animals = pd.DataFrame({'Cows': [12, 20, 30, 35], 'Goats': [22, 19, 5, 25], 'Horses':[4, 7, 10, 15]})
>>> zoo_animals = pd.DataFrame({'Lion': [4, 10, 5, 3], 'Tiger': [9, 11, 5, 2]})
```

```
>>> animals
   Cows  Goats  Horses
0    12     22       4
1    20     19       7
2    30      5      10
3    35     25      15
```

```
>>> zoo_animals
   Lion  Tiger
0     4      9
1    10     11
2     5      5
3     3      2
```

(ii) Save the first one in *animals.csv* and the second one in zoo_*animals.csv* at your Desktop.

```
>>> animals.to_csv("~/Desktop/animals.csv")
```

```
>>> zoo_animals.to_csv("~/Desktop/zoo_animals.csv")
```

Pandas

Answer (iii-v): (iii) Read *animals.csv* and zoo_*animals.csv*.

(iv) Check the dataframe shape, print its 5 first rows.

(v) Find the mean value for each column.

```
>>> import pandas as pd
>>> animals = pd.read_csv("~/Desktop/animals.csv", index_col=0)
>>> zoo_animals = pd.read_csv("~/Desktop/zoo_animals.csv", index_col=0)
>>> animals.shape
(4, 3)
>>> zoo_animals.shape
(4, 2)
```

```
>>> animals.head()
   Cows  Goats  Horses
0    12     22       4
1    20     19       7
2    30      5      10
3    35     25      15
>>> zoo_animals.head()
   Lion  Tiger
0     4      9
1    10     11
2     5      5
3     3      2
```

(v)

```
>>> animals.describe()
            Cows      Goats     Horses
count   4.000000   4.000000   4.000000
mean   24.250000  17.750000   9.000000
std    10.275375   8.845903   4.690416
min    12.000000   5.000000   4.000000
25%    18.000000  15.500000   6.250000
50%    25.000000  20.500000   8.500000
75%    31.250000  22.750000  11.250000
max    35.000000  25.000000  15.000000
>>> zoo_animals.describe()
            Lion      Tiger
count   4.000000   4.000000
mean    5.500000   6.750000
std     3.109126   4.031129
min     3.000000   2.000000
25%     3.750000   4.250000
50%     4.500000   7.000000
75%     6.250000   9.500000
max    10.000000  11.000000
```

Pandas - Combining

Combine the two dataframes:

pandas.concat(*objs, axis=0, join='outer', ignore_index=False, keys=None, levels=None, names=None, verify_integrity=False, sort=False, copy=True*)

```
>>> animals
   Cows  Goats  Horses
0    12     22       4
1    20     19       7
2    30      5      10
3    35     25      15
```

```
>>> zoo_animals
   Lion  Tiger
0     4      9
1    10     11
2     5      5
3     3      2
```

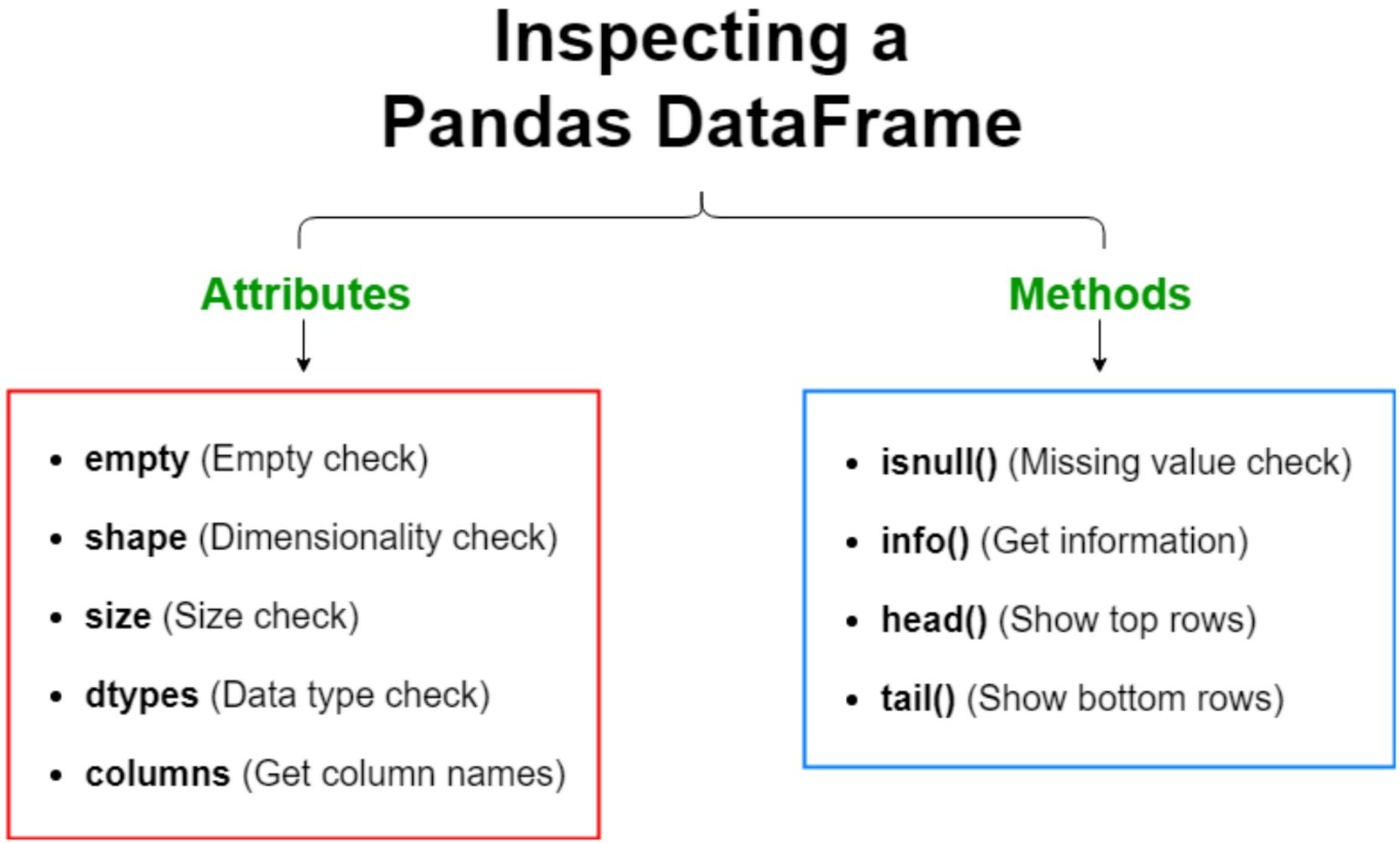
```
>>> pd.concat([animals, zoo_animals])
   Cows  Goats  Horses  Lion  Tiger
0  12.0   22.0     4.0   NaN    NaN
1  20.0   19.0     7.0   NaN    NaN
2  30.0    5.0    10.0   NaN    NaN
3  35.0   25.0    15.0   NaN    NaN
0   NaN    NaN     NaN   4.0    9.0
1   NaN    NaN     NaN  10.0   11.0
2   NaN    NaN     NaN   5.0    5.0
3   NaN    NaN     NaN   3.0    2.0
```

```
>>> pd.concat([animals, zoo_animals], axis=1)
   Cows  Goats  Horses  Lion  Tiger
0    12     22       4     4      9
1    20     19       7    10     11
2    30      5      10     5      5
3    35     25      15     3      2
```

axis*{0/'index', 1/'columns'}, default 0*
The axis to concatenate along.

Inspecting a Pandas Dataframe

Most of the time the dataframe is already created and you need to understand your data.



Inspecting a Pandas Dataframe

Task: (i) Read the *animalsLarge.csv* and explore it.

Note: Get *animalsLarge.csv*
@git: https://github.com/edrakopo/pythonMLBootcamp/tree/master/Day2

Hint:

Check if the daraframe is empty, its size and shape, if it has missing values, column names and values etc.

Inspecting a Pandas Dataframe

Answer :

```
>>> import pandas as pd
>>> animals = pd.read_csv("animalsLarge.csv", index_col=0)
```

Attributes

```
>>> animals.empty
False
>>> animals.shape
(11, 3)
>>> animals.size
33
>>> animals.dtypes
Cows      float64
Goats      object
Horses    float64
dtype: object
>>> animals.columns
Index(['Cows', 'Goats', 'Horses'], dtype='object')
```

Methods

```
>>> animals.isnull()
      Cows  Goats  Horses
0    False  False   False
1    False  False   False
2    False  False   False
3    False  False   False
4    False  False   False
5    False  False   False
6     True  False   False
7    False  False    True
8     True  False   False
9    False  False   False
10   False  False   False
>>> animals.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 11 entries, 0 to 10
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Cows    9 non-null      float64
 1   Goats   11 non-null     object
 2   Horses  10 non-null     float64
dtypes: float64(2), object(1)
memory usage: 352.0+ bytes
```

string

Inspecting a Pandas Dataframe

Answer :

Methods

```
>>> print(animals.head())
   Cows Goats  Horses
0  12.0    22     4.0
1  20.0    19     7.0
2  30.0     5    10.0
3  35.0    25    15.0
4  27.0    14     4.0
>>> print(animals.tail())
    Cows Goats  Horses
6    NaN   not    12.0
7   12.0    30     NaN
8    NaN    26    13.0
9   20.0    30    12.0
10  24.0    34     5.0
```

Fill the NaN values with 0:

```
animals.fillna(0)
```

Inspecting a Pandas Dataframe - groupby

pandas.DataFrame.groupby

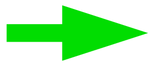
```
DataFrame.groupby(by=None, axis=0, level=None, as_index=True, sort=True,
group_keys=True, observed=False, dropna=True)
```

Group DataFrame using a mapper or by a Series of columns.
A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

`groupby()` function returns `DataFrameGroupBy` object after collecting the identical data into groups from pandas DataFrame.This object contains several methods (`sum()`, `mean()` e.t.c) that can be used to aggregate the grouped rows.

Examples

```
1 import pandas as pd
2 technologies   = ({
3     'Courses':["Spark","PySpark","Hadoop","Python","Pandas","Hadoop","Spark","Python","NA"],
4     'Fee' :[22000,25000,23000,24000,26000,25000,25000,22000,1500],
5     'Duration':['30days','50days','55days','40days','60days','35days','30days','50days','40days'],
6     'Discount':[1000,2300,1000,1200,2500,None,1400,1600,0]
7           })
8 df = pd.DataFrame(technologies)
9 print(df)
10
11 # Use groupby() to compute the sum
12 df2 =df.groupby(['Courses']).sum()
13 print(df2)
```



Courses	Fee	Discount
Hadoop	48000	1000.0
NA	1500	0.0
Pandas	26000	2500.0
PySpark	25000	2300.0
Python	46000	2800.0
Spark	47000	2400.0

Inspecting a Pandas Dataframe - groupby

Examples

Group by multiple columns

```
df2 =df.groupby(['Courses', 'Duration']).sum()
print(df2)
```

```
# Group by multiple columns:
                    Fee  Discount
Courses Duration
Hadoop  35days     25000       0.0
        55days     23000    1000.0
NA      40days      1500       0.0
Pandas  60days     26000    2500.0
PySpark 50days     25000    2300.0
Python  40days     24000    1200.0
        50days     22000    1600.0
Spark   30days     47000    2400.0
```

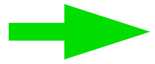
Add Row Index to the group by result

```
df2 = df.groupby(['Courses','Duration']).sum().reset_index()
print(df2)
```

```
# Add Row Index to the group by result
   Courses Duration    Fee  Discount
0   Hadoop   35days  25000       0.0
1   Hadoop   55days  23000    1000.0
2       NA   40days   1500       0.0
3   Pandas   60days  26000    2500.0
4  PySpark   50days  25000    2300.0
5   Python   40days  24000    1200.0
6   Python   50days  22000    1600.0
7    Spark   30days  47000    2400.0
```

Drop rows that have None/Nan on group keys

```
df2=df.groupby(by=['Courses'], dropna=True).sum()
print(df2)
```



```
             Fee  Discount
Courses
Hadoop     48000    1000.0
NA          1500       0.0
Pandas     26000    2500.0
PySpark    25000    2300.0
Python     46000    2800.0
Spark      47000    2400.0
```

dropna: ***bool, default True***

If **True**, and if group keys contain NA values, NA values together with row/column will be dropped. If **False**, NA values will also be treated as the key in groups.

Inspecting a Pandas Dataframe - pivot table

A pivot table is a table of grouped values that aggregates the individual items of a more extensive table within one or more discrete categories. This summary might include sums, averages, or other statistics, which the pivot table groups together using a chosen aggregation function applied to the grouped values. [wikipedia]

pandas.pivot_table

pandas.pivot_table(*data*, *values=None*, *index=None*, *columns=None*, *aggfunc='mean'*, *fill_value=None*, *margins=False*, *dropna=True*, *margins_name='All'*, *observed=False*, *sort=True*)
Create a spreadsheet-style pivot table as a DataFrame.

Examples

```
import pandas as pd
technologies   = ({
    'Courses':["Spark","PySpark","Hadoop","Python","Pandas","Hadoop","Spark","Python","NA"],
    'Fee' :[22000,25000,23000,24000,26000,25000,25000,22000,1500],
    'Duration':['30days','50days','55days','40days','60days','35days','30days','50days','40days'],
    'Discount':[1000,2300,1000,1200,2500,None,1400,1600,0]
          })
```

```
pivot_table = pd.pivot_table(df, index=['Courses']) #default aggfunc='mean'
print(pivot_table)

pivot_table_sum = pd.pivot_table(df, index=['Courses'],aggfunc=np.sum)
print(pivot_table_sum)
```

Inspecting a Pandas Dataframe - pivot table

Examples

```
1 import pandas as pd
 2 import numpy as np
 3
 4 technologies   = ({
 5     'Courses':["Spark","PySpark","Hadoop","Python","Pandas","Hadoop","Spark","Python","NA"],
 6     'Fee' :[22000,25000,23000,24000,26000,25000,25000,22000,1500],
 7     'Duration':['30days','50days','55days','40days','60days','35days','30days','50days','40days'],
 8     'Discount':[1000,2300,1000,1200,2500,None,1400,1600,0]
 9           })
10 df = pd.DataFrame(technologies)
11 print(df)
12
13 # Use groupby() to compute the sum
14 df2 =df.groupby(['Courses']).sum()
15 print(df2)
16
...........
32 pivot_table = pd.pivot_table(df, index=[‘Courses']) #default aggfunc='mean'
33 print(pivot_table)
34
35 pivot_table_sum = pd.pivot_table(df, index=['Courses'],aggfunc=np.sum)
36 print(pivot_table_sum)
```

Courses	Discount	Fee
Hadoop	1000.0	24000
NA	0.0	1500
Pandas	2500.0	26000
PySpark	2300.0	25000
Python	1400.0	23000
Spark	1200.0	23500

Courses	Discount	Fee
Hadoop	1000.0	48000
NA	0.0	1500
Pandas	2500.0	26000
PySpark	2300.0	25000
Python	2800.0	46000
Spark	2400.0	47000

Pandas

Data Cleaning

df.columns = ['a','b','c']	Rename columns
pd.isnull()	Checks for null Values, Returns Boolean Arrray
pd.notnull()	Opposite of pd.isnull()
df.dropna()	Drop all rows that contain null values
df.dropna(axis=1)	Drop all columns that contain null values
df.dropna(axis=1,thresh=n)	Drop all rows have have less than n non null values
df.fillna(x)	Replace all null values with x
s.fillna(s.mean())	Replace all null values with the mean
s.astype(float)	Convert the datatype of the series to float
s.replace(1,'one')	Replace all values equal to 1 with 'one'

Filter, Sort, and Groupby

df[df[col] > 0.6]	Rows where the column col is greater than 0.6
df[(df[col] > 0.6) & (df[col] < 0.8)]	Rows where 0.8 > col > 0.6
df.sort_values(col1)	Sort values by col1 in ascending order
df.sort_values(col2,ascending=False)	Sort values by col2 in descending order.5
df.sort_values([col1,col2],ascending=[True,False])	Sort values by col1 in ascending order then col2 in descending order
df.groupby(col)	Returns a groupby object for values from one column
df.groupby([col1,col2])	Returns groupby object for values from multiple columns

Hands On

Task: **(i)** Explore the dataset with Covid-19 cases from:

https://github.com/CSSEGISandData/COVID-19/tree/master/csse_covid_19_data/csse_covid_19_daily_reports

Hint: covid_data = pd.read_csv('https://raw.githubusercontent.com/CSSEGISandData/COVID-19/master/csse_covid_19_data/csse_covid_19_daily_reports/03-16-2020.csv')

(ii) Write a Python program to display first 5 rows from COVID-19 dataset. Also print the dataset information and check the missing values.

(iii) Write a Python program to get the latest number of confirmed, deaths, recovered and active cases of Novel Coronavirus (COVID-19) Country wise.

(iv) Write a Python program to get the latest number of confirmed deaths and recovered people of Novel Coronavirus (COVID-19) cases Country/Region - Province/State wise.

Hands On

(ii) Write a Python program to display first 5 rows from COVID-19 dataset. Also print the dataset information and check the missing values.

Answer:

```
import pandas as pd
covid_data= pd.read_csv('https://raw.githubusercontent.com/CSSEGISandData/COVID-19/master/csse_covid_19_data/csse_covid_19_daily_reports/03-17-2020.csv')
print(covid_data.head())
print("\nDataset information:")
print(covid_data.info())
print("\nMissing data information:")
print(covid_data.isna().sum())
```

Hands On

(iii) Write a Python program to get the latest number of confirmed, deaths, recovered and active cases of Novel Coronavirus (COVID-19) Country wise.

Answer:

```
import pandas as pd
covid_data= pd.read_csv('https://raw.githubusercontent.com/CSSEGISandData/COVID-19/master/csse_covid_19_data/csse_covid_19_daily_reports/03-17-2020.csv')
covid_data['Active'] = covid_data['Confirmed'] - covid_data['Deaths'] - covid_data['Recovered']
result = covid_data.groupby('Country/Region')['Confirmed', 'Deaths', 'Recovered', 'Active'].sum().reset_index()
print(result)
```

Hands On

(iv) Write a Python program to get the latest number of confirmed deaths and recovered people of Novel Coronavirus (COVID-19) cases Country/Region - Province/State wise.

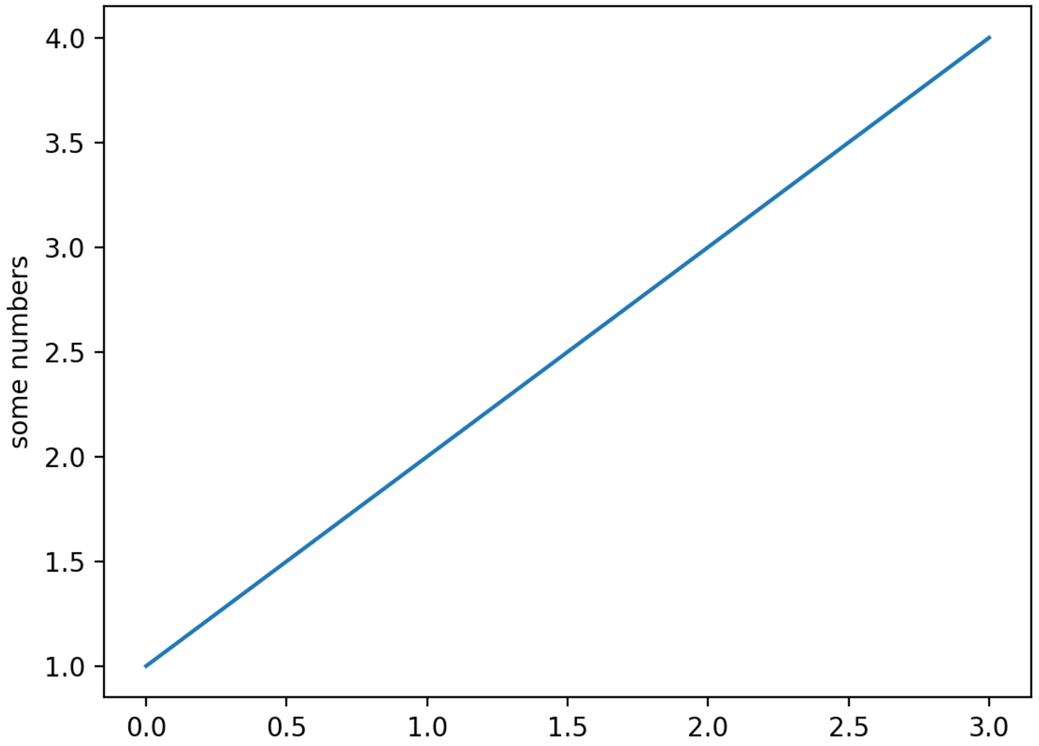
Answer:

```
import pandas as pd
covid_data= pd.read_csv('https://raw.githubusercontent.com/CSSEGISandData/COVID-19/master/csse_covid_19_data/csse_covid_19_daily_reports/03-16-2020.csv')
data = covid_data.groupby(['Country/Region', 'Province/State'])['Confirmed', 'Deaths', 'Recovered'].max()
pd.set_option('display.max_rows', None)
print(data)
```

Visualising your data

Data Visualisation - matplotlib.pyplot

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4])
plt.ylabel('some numbers')
plt.show() ||   plt.savefig(“plot1.png")
```



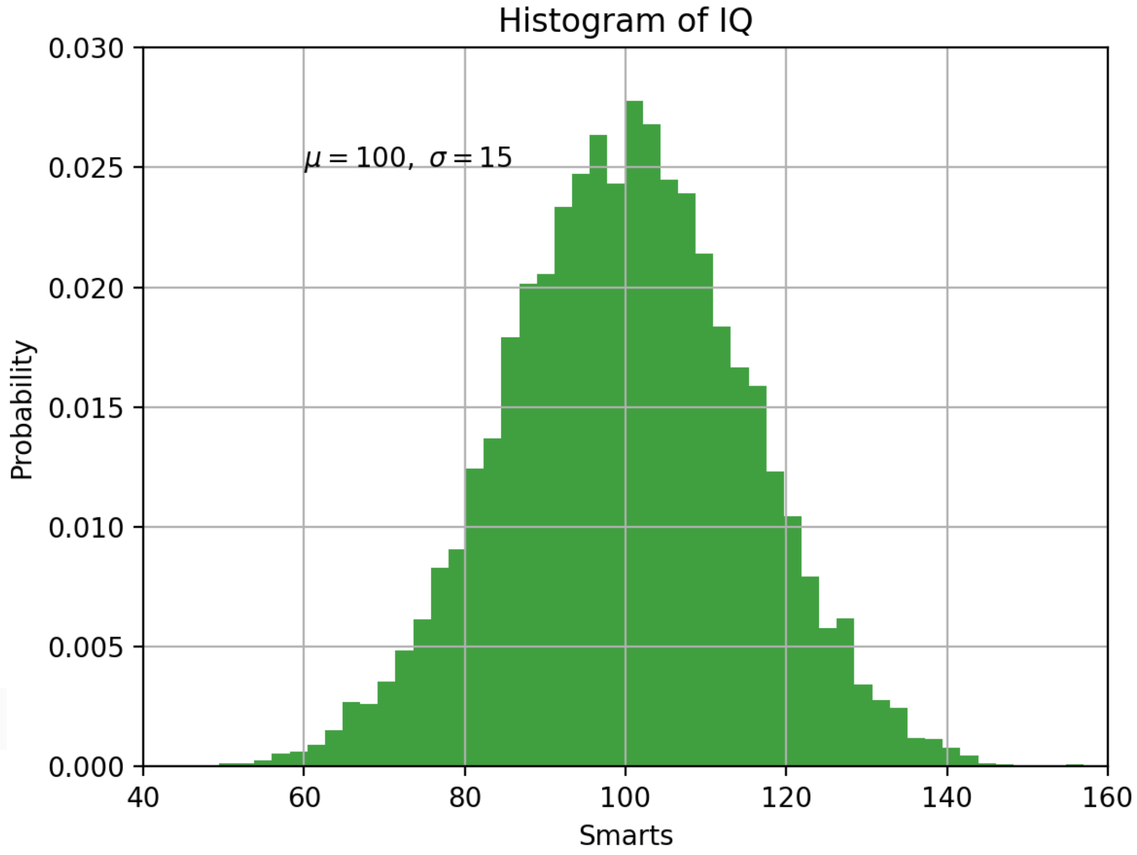
Why the x-axis ranges from 0-3 and the y-axis from 1-4?

If you provide a single list or array to **`plot`**, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you. Since python ranges start with 0, the default x vector has the same length as y but starts with 0. Hence the x data are `[0, 1, 2, 3]`.

Data Visualisation - matplotlib.pyplot

```
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50,
density=1, facecolor='g', alpha=0.75)


plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\
\sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()  || plt.savefig(“plot2.png”)
```

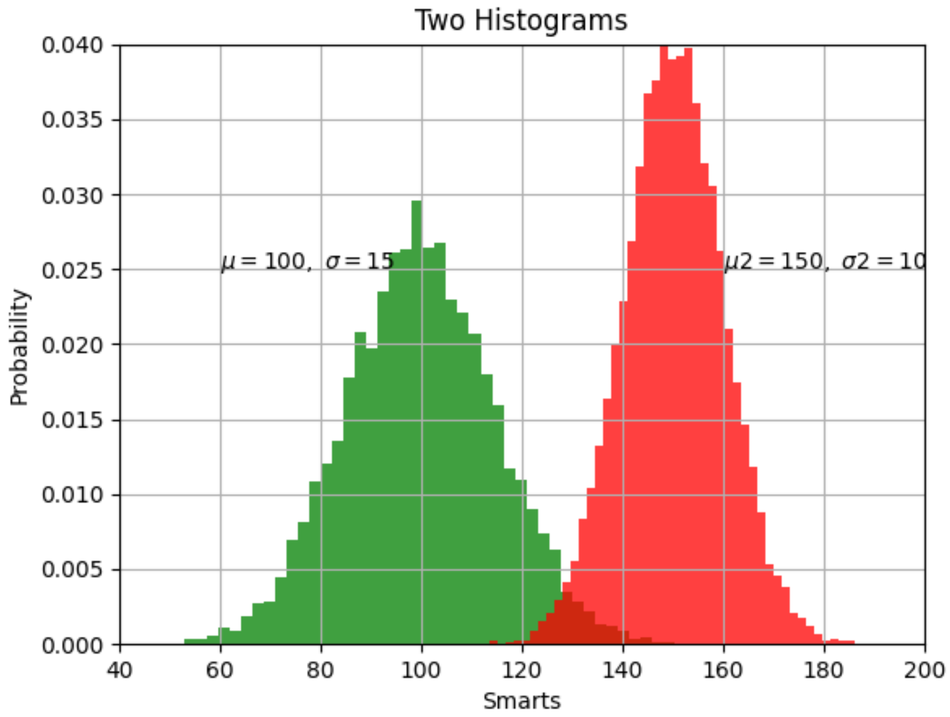
randn: generates points following a normal distribution



Data Visualisation - matplotlib.pyplot

```
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)
# the histogram of the data
n, bins, patches = plt.hist(x, 50, density=1, facecolor='g', alpha=0.75)

mu2, sigma2 = 150, 10
x = mu2 + sigma2 * np.random.randn(10000)
# the histogram of the data
n, bins, patches = plt.hist(x, 50, density=1, facecolor='r', alpha=0.75)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Two Histograms')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.text(160, .025, r'$\mu2=150,\ \sigma2=10$')
plt.axis([40, 200, 0, 0.04])
plt.grid(True)
plt.show()
```

2 histograms in the same plot



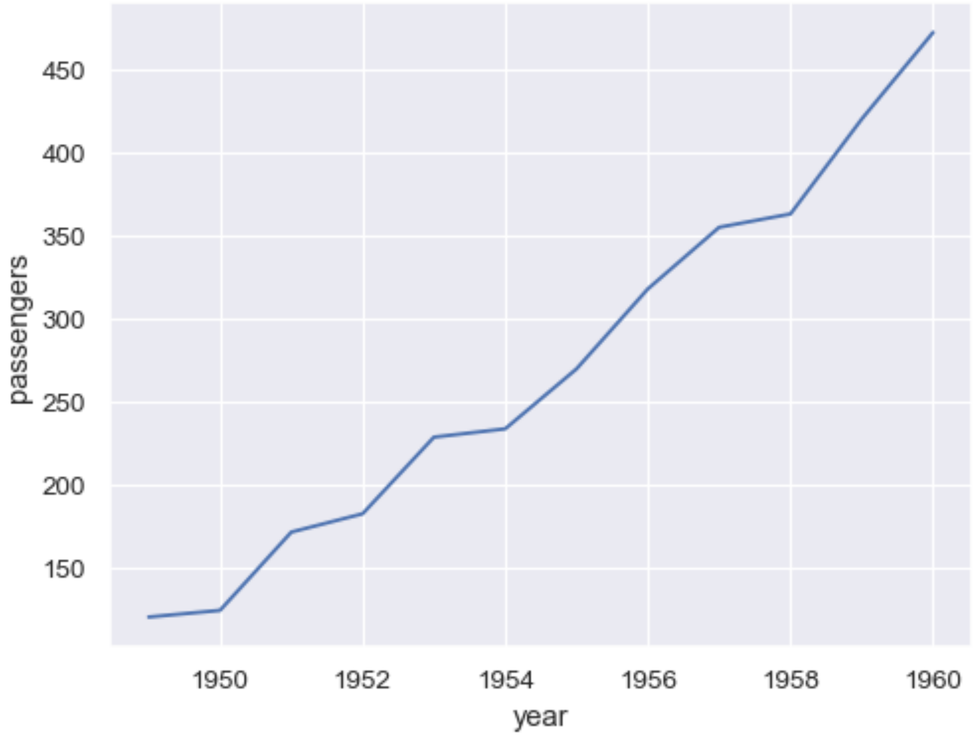
Data Visualisation - seaborn

seaborn : a powerful but easy-to-use data visualisation tool

seaborn.lineplot Draw a line plot with possibility of several semantic groupings.

```
import matplotlib.pyplot as plt
import seaborn as sns
flights = sns.load_dataset("flights")
flights.head()
```

	year	month	passenger
0	1949	Jan	112
1	1949	Feb	118
2	1949	Mar	132
3	1949	Apr	129
4	1949	May	121




```
may_flights = flights.query("month == 'May'")
sns.lineplot(data=may_flights, x="year",
y="passengers")
plt.savefig("plot3.png")
```

Data Visualisation - seaborn

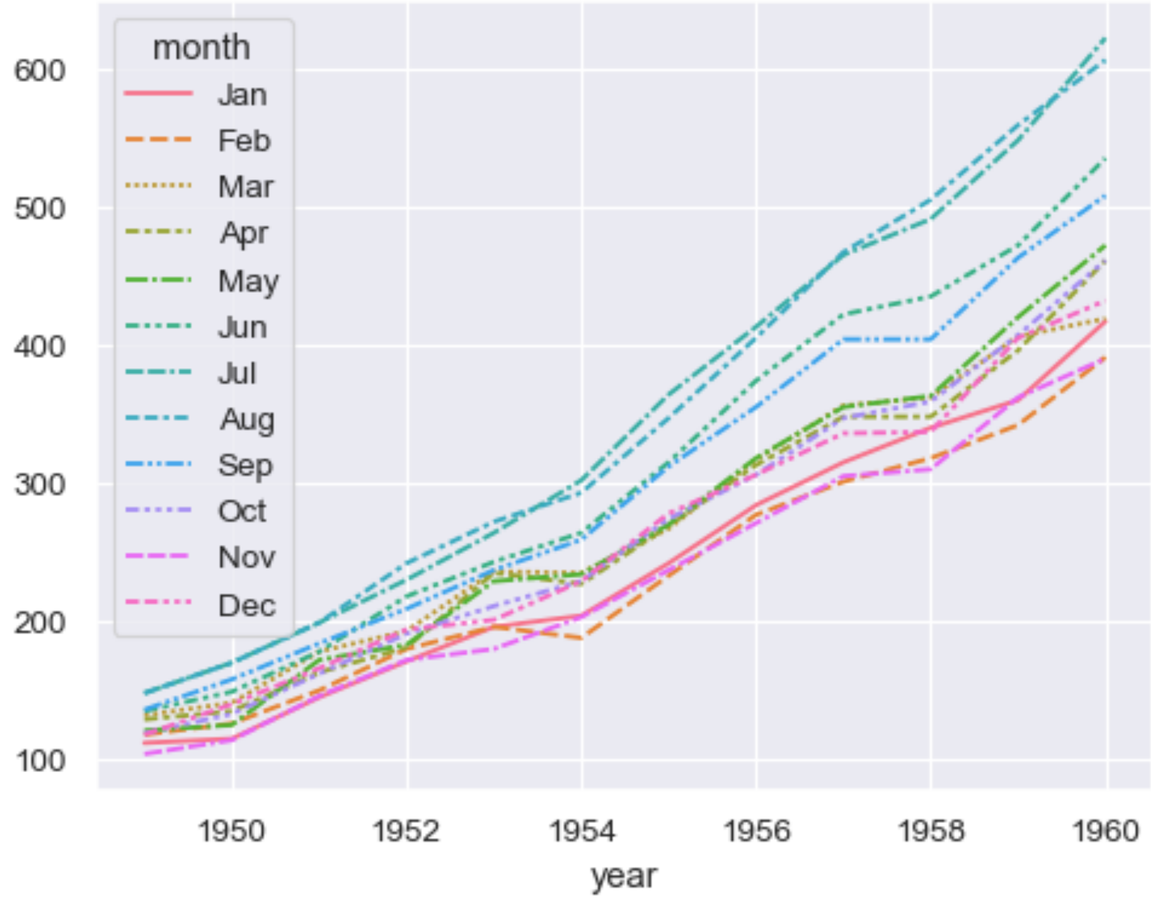
seaborn.lineplot

Pivot the dataframe to a wide-form representation:

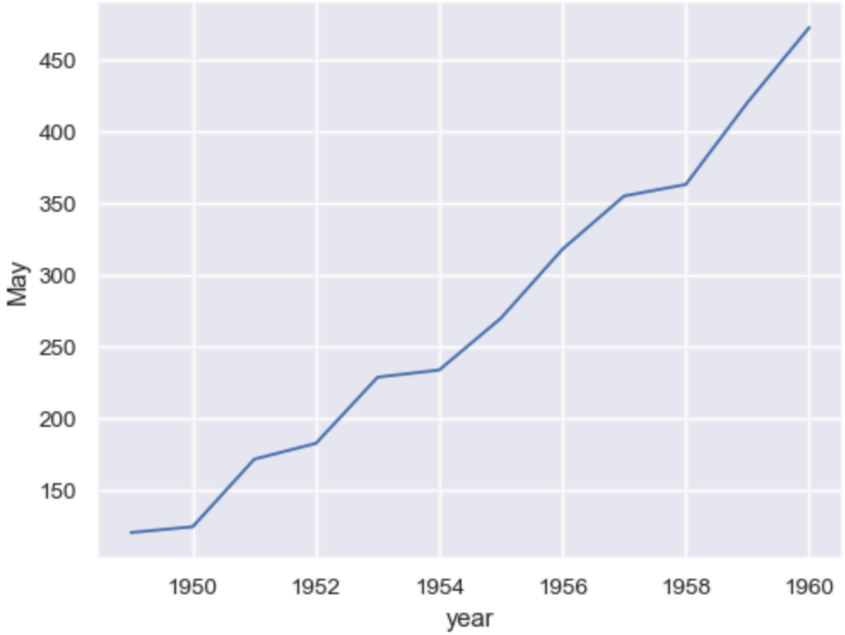
The **pivot()** function is used to reshape a DataFrame organised by given index / column values.

```
flights_wide = flights.pivot("year", "month", "passengers")
flights_wide.head()
```

```
sns.lineplot(data=flights_wide)
```




```
sns.lineplot(data=flights_wide["May"])
```



Data Visualisation - seaborn: Check the iris dataset

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

https://en.wikipedia.org/wiki/Iris_flower_data_set

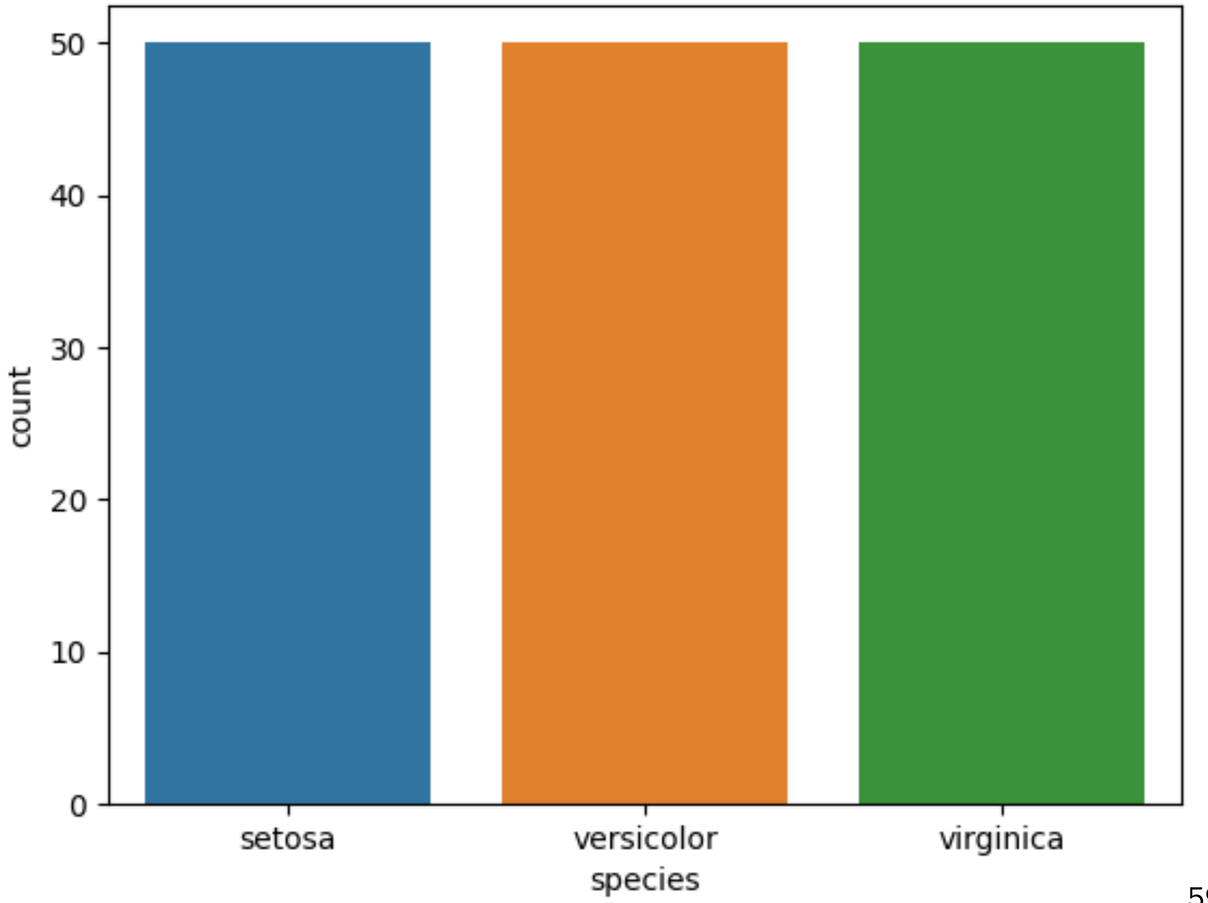
```
>>> iris = sns.load_dataset("iris")
>>> iris.head()
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
>>> iris.describe()
       sepal_length  sepal_width  petal_length  petal_width
count    150.000000   150.000000    150.000000   150.000000
mean       5.843333     3.057333      3.758000     1.199333
std        0.828066     0.435866      1.765298     0.762238
min        4.300000     2.000000      1.000000     0.100000
25%        5.100000     2.800000      1.600000     0.300000
50%        5.800000     3.000000      4.350000     1.300000
75%        6.400000     3.300000      5.100000     1.800000
max        7.900000     4.400000      6.900000     2.500000
>>> iris.tail()
     sepal_length  sepal_width  petal_length  petal_width    species
145           6.7          3.0           5.2          2.3  virginica
146           6.3          2.5           5.0          1.9  virginica
147           6.5          3.0           5.2          2.0  virginica
148           6.2          3.4           5.4          2.3  virginica
149           5.9          3.0           5.1          1.8  virginica
```

Data Visualisation - seaborn

seaborn.countplot

```
sns.countplot(x="species", data=iris)
```

To visualise the counts of each species in our dataset, we can use a bar graph.

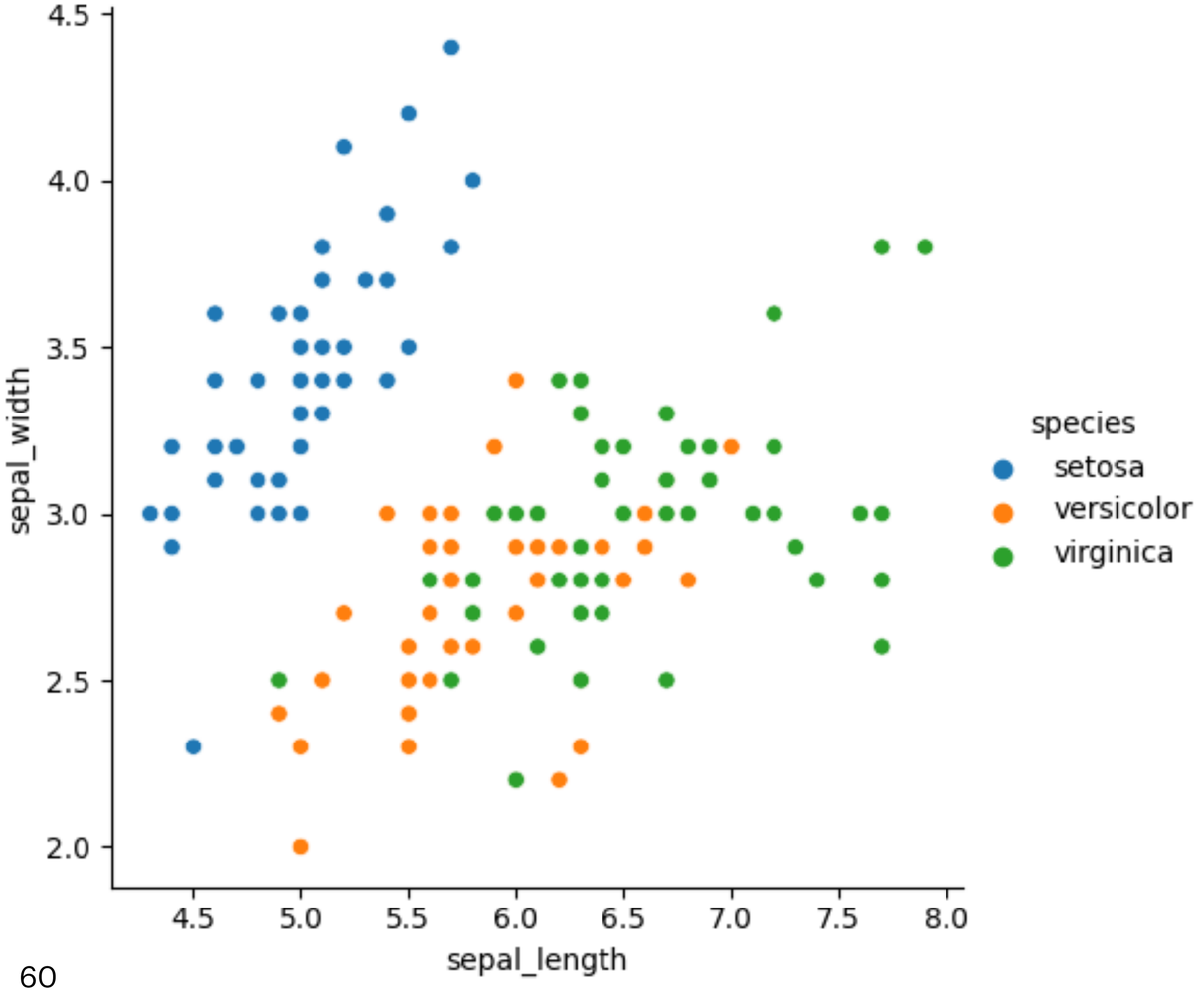


Data Visualisation - seaborn

`seaborn.relplot`

```
sns.relplot(x='sepal_length', y='sepal_width', hue='species', data=iris)
```

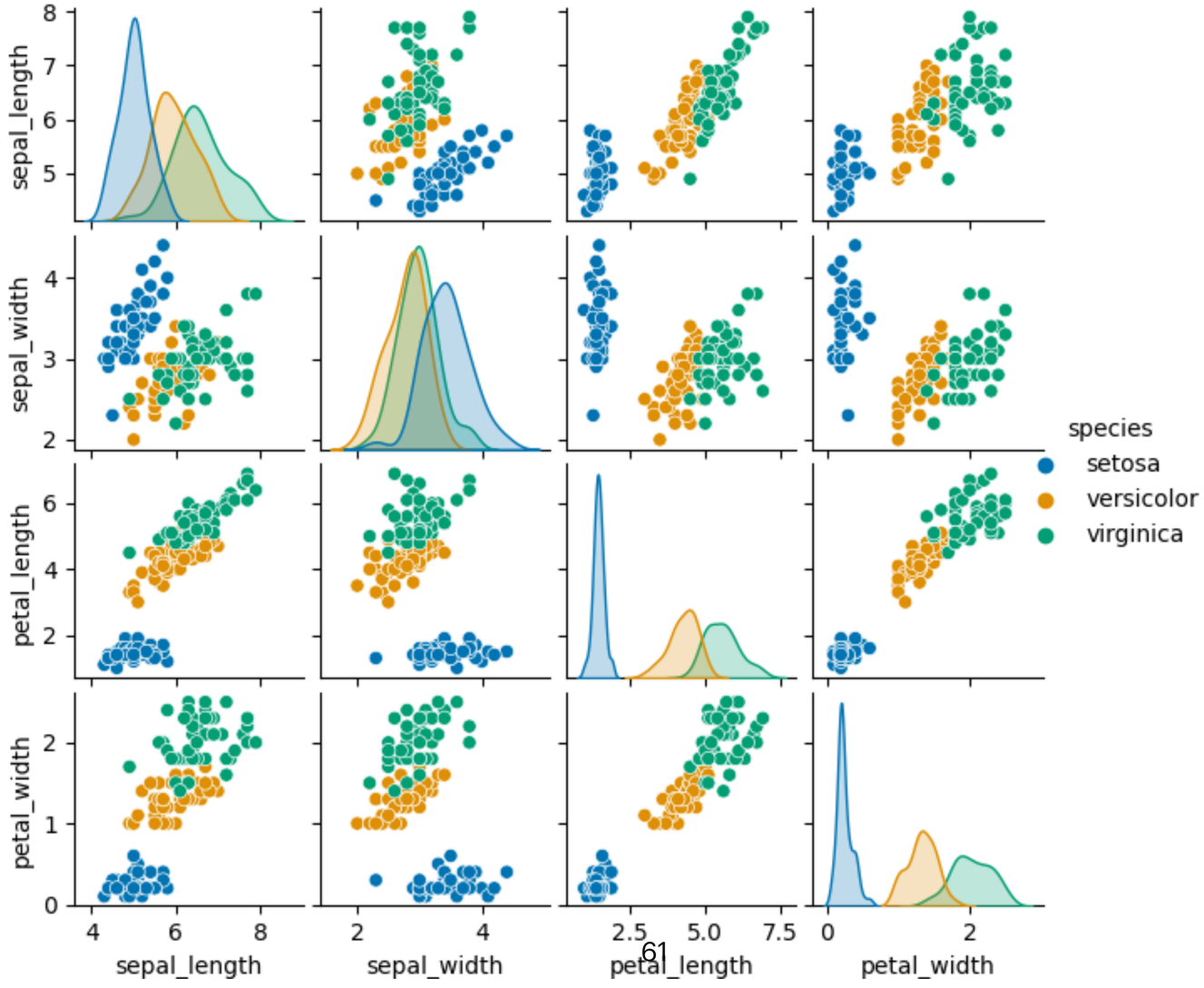
We can change the color of the points based on which species it represents. This can be done by including the **hue** input. We set this equal to the Species variable, so each color corresponds to a different group.



Data Visualisation - seaborn

seaborn.pairplot

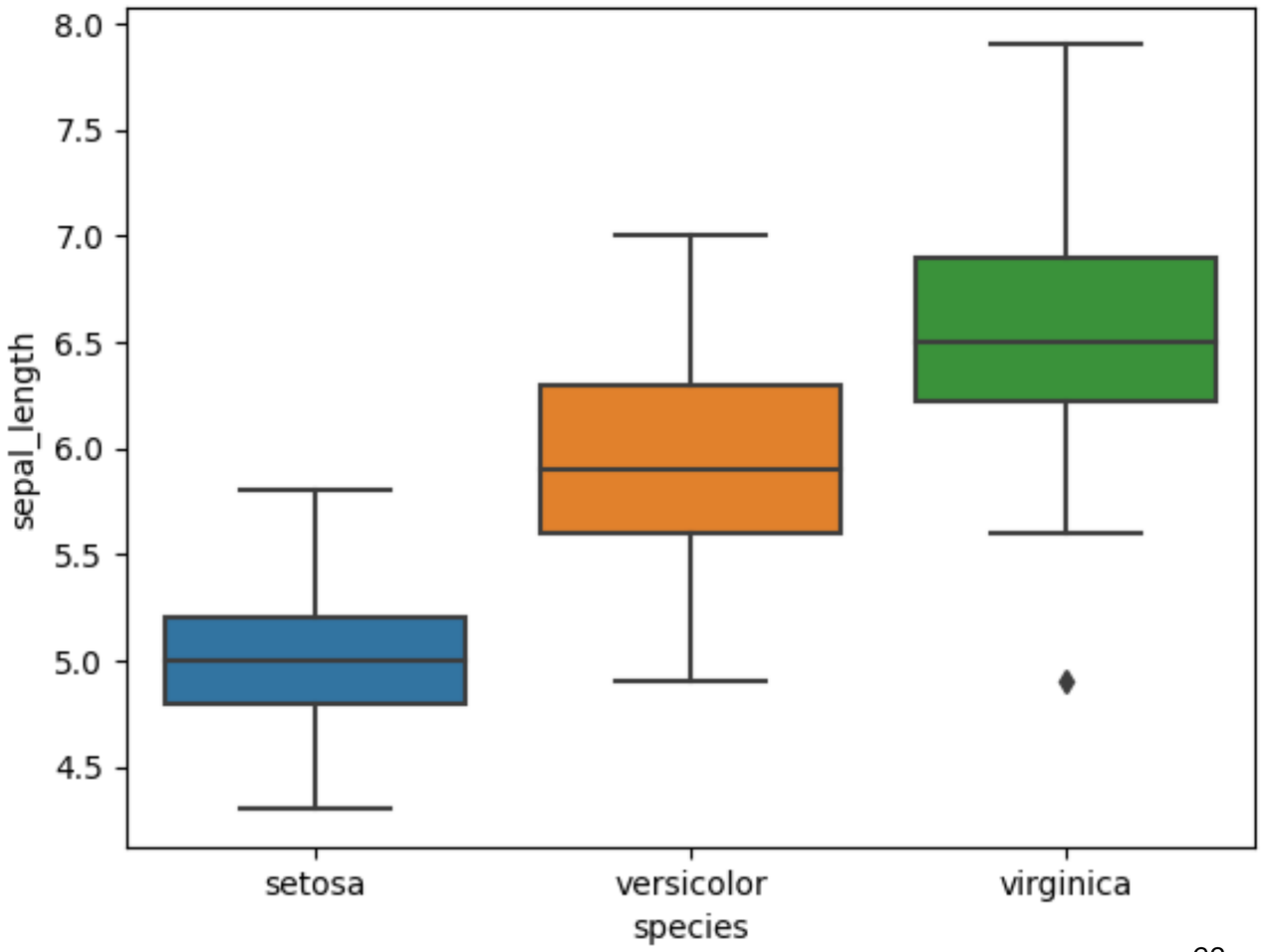
```
sns.pairplot(iris, hue="species", height = 2, palette = 'colorblind
```

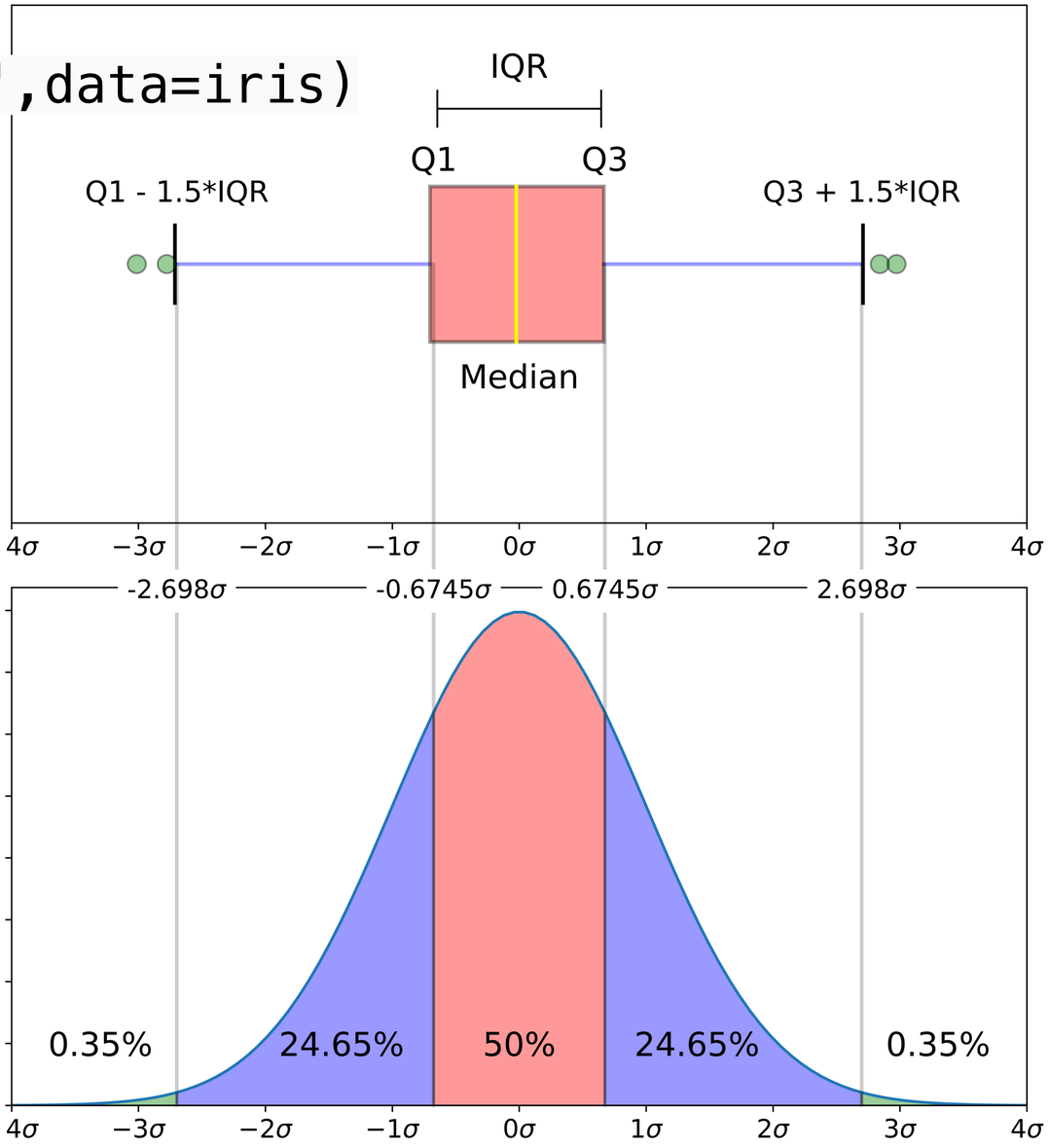


Data Visualisation - seaborn

seaborn.boxplot

```
sns.boxplot(x='species', y='sepal_length',data=iris)
```





Data Visualisation

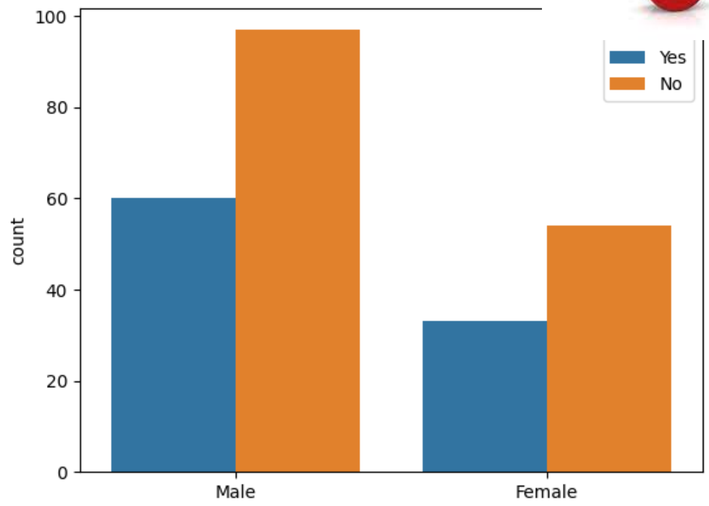
Task: (i) Read and explore the tips dataset from seaborn.

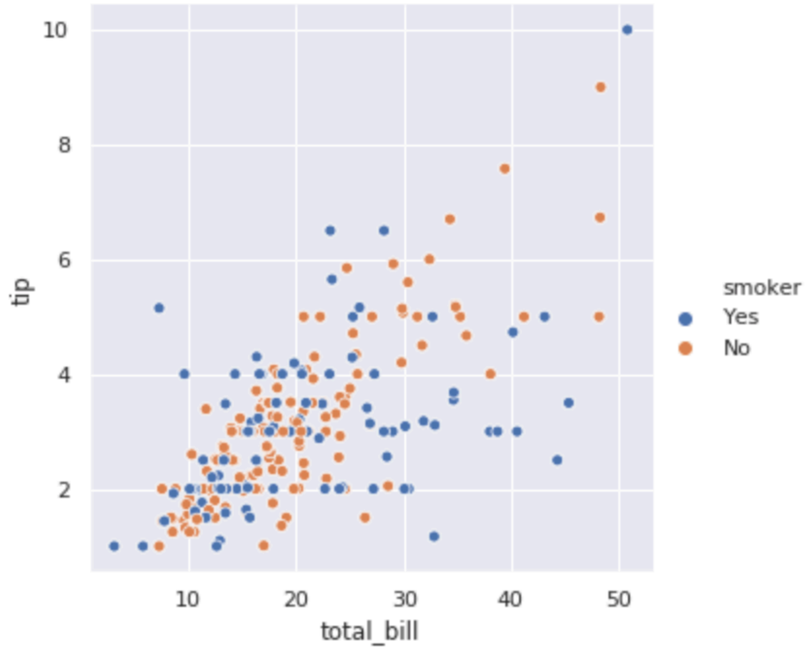
`Hint: import matplotlib.pyplot, seaborn, pandas, numpy`

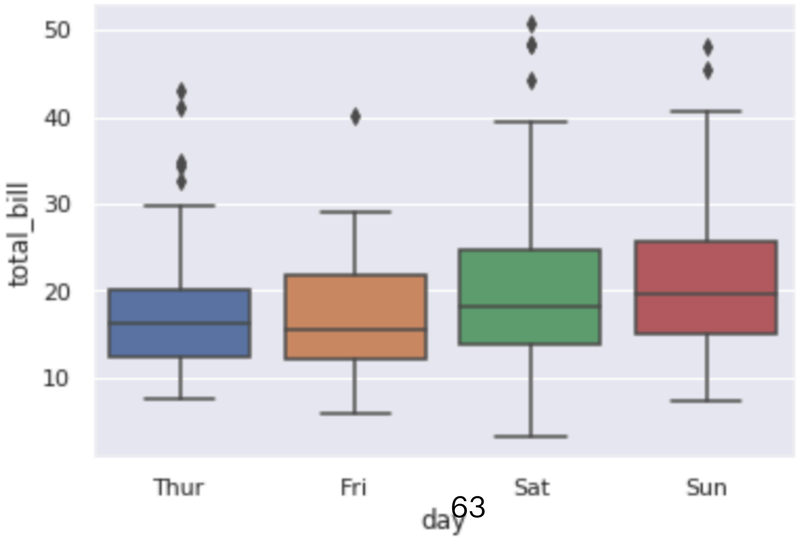
and load the dataset using:

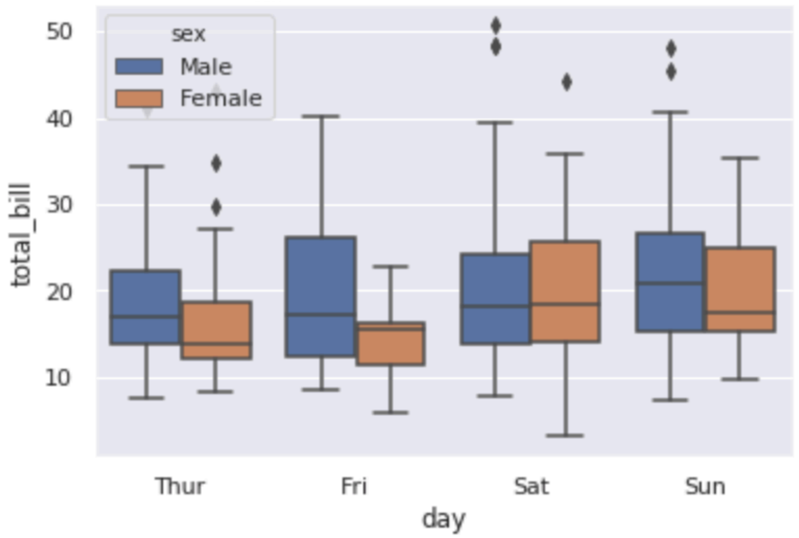
```
tips = sns.load_dataset("tips")
```

(ii) Create the following plots.





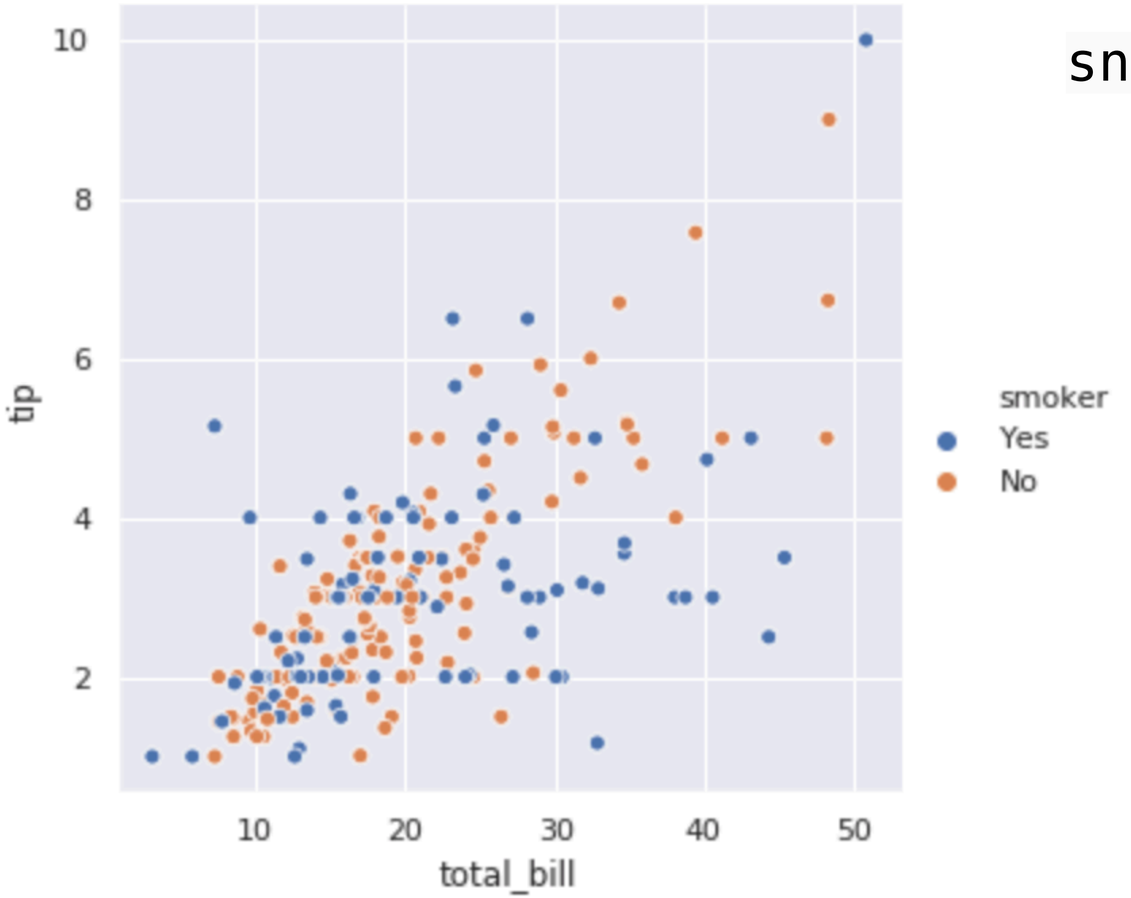




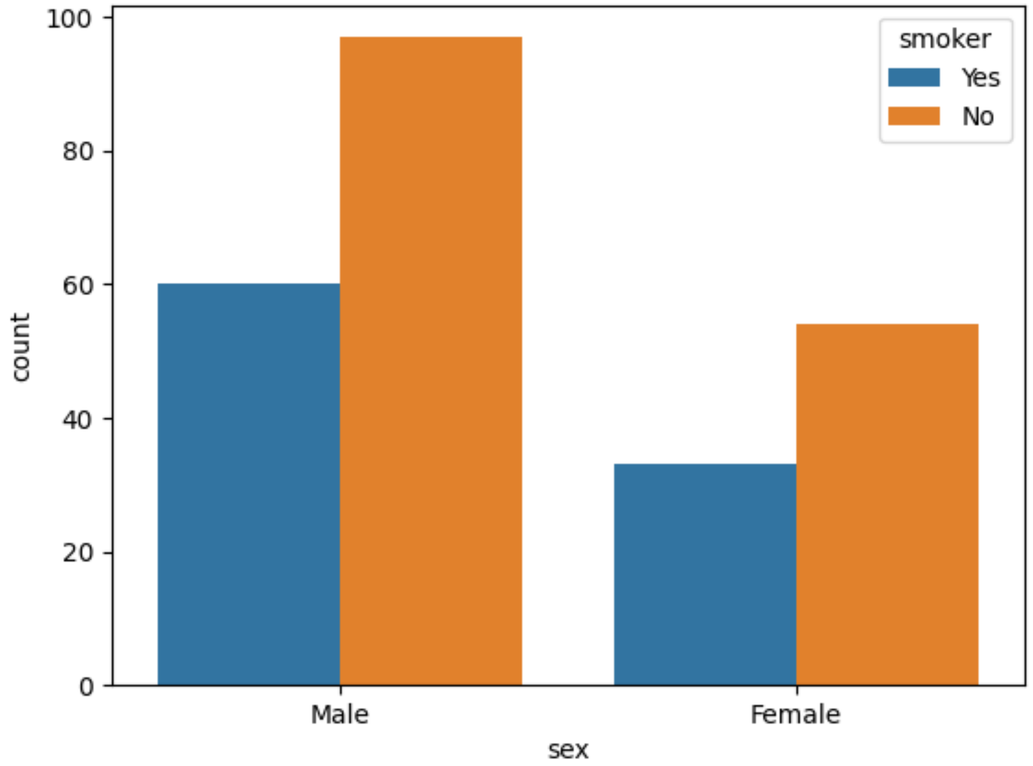
Data Visualisation

Answer :

```
sns.relplot(x = "total_bill", y = "tip", hue= "smoker", data = df);
```



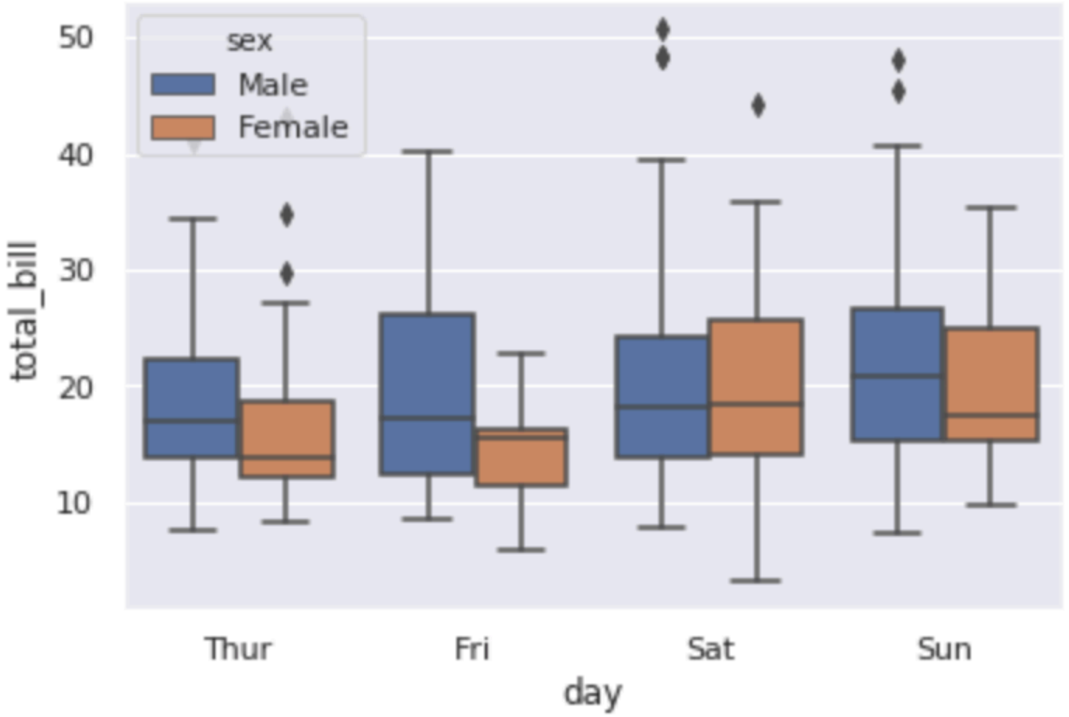

```
sns.countplot(x="sex",hue="smoker",data=tips)
```



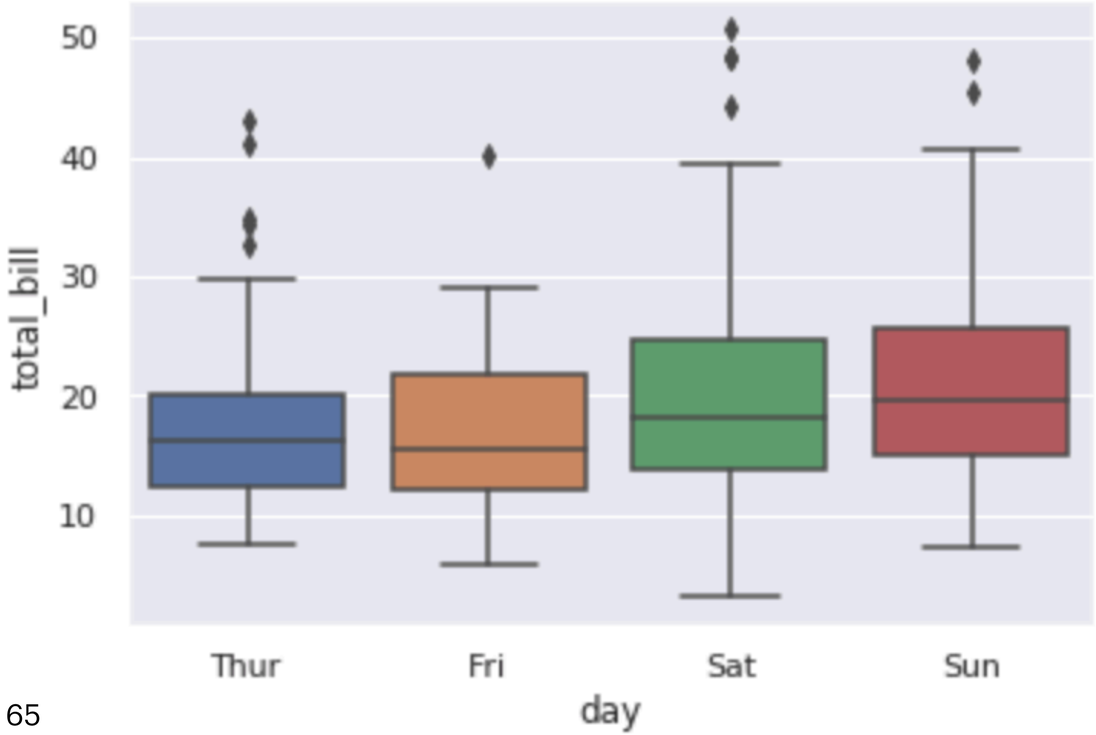
Data Visualisation

Answer :

```
sns.boxplot(x = "day", y = "total_bill", hue = "sex", data = df);
```




```
sns.boxplot(x = "day", y = "total_bill", data = df);
```



Correlation matrix

A correlation is a value between -1 and 1 that amounts to how closely values of two separate features move simultaneously. A *positive* correlation means that as one feature increases the other one also increases, while a *negative* correlation means one feature increases as the other decreases.

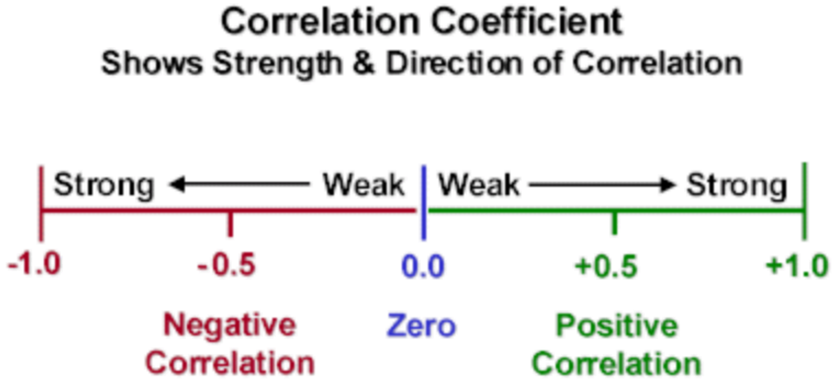


Image from edugyan.in

df.corr() calculates the correlations between the numeric features and returns a DataFrame.

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import seaborn as sns
5
6 iris=sns.load_dataset("iris")
7 corrs = iris.corr(numeric_only=True)
8 print(corrs)
9
10 plt.figure(figsize=(10,8))
11 sns.heatmap(corrs, cmap='RdBu_r', annot=True)
12 plt.show()
```

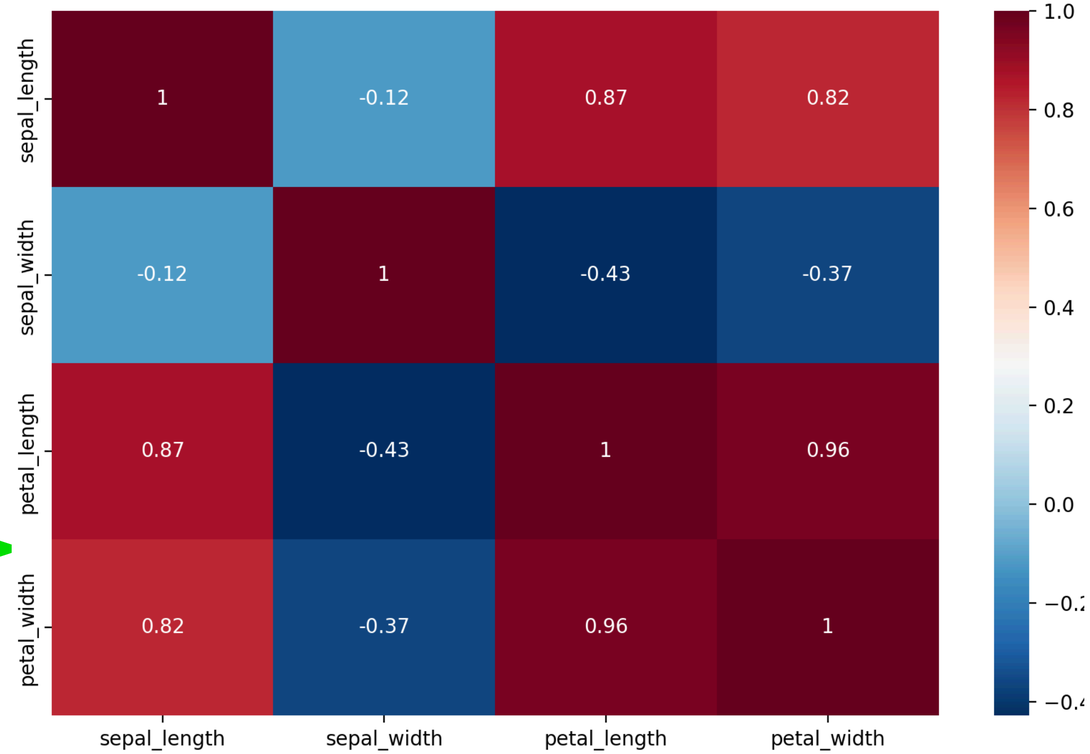
	sepal_length	sepal_width	petal_length	petal_width
sepal_length	1.000000	-0.117570	0.871754	0.817941
sepal_width	-0.117570	1.000000	-0.428440	-0.366126
petal_length	0.871754	-0.428440	1.000000	0.962865
petal_width	0.817941	-0.366126	0.962865	1.000000

Heatmap of the correlations

The cmap='RdBu_r' argument tells the heatmap what colour palette to use. A high positive correlation appears as *dark red* and a high negative correlation as *dark blue*. Closer to white signifies a weak relationship.

annot=True includes the values of the correlations in the boxes for easier reading and interpretation.

```
1  import numpy as np
2  import pandas as pd
3  import matplotlib.pyplot as plt
4  import seaborn as sns
5
6  iris=sns.load_dataset("iris")
7  corrs = iris.corr(numeric_only=True)
8  print(corrs)
9
10 plt.figure(figsize=(10,8))
11 sns.heatmap(corrs, cmap='RdBu_r', annot=True)
12 plt.show()
```



Scaling/Normalising Dataframes

What's the difference between **scaling** and **normalisation**?

- in **scaling**, you're changing the range of your data

you're transforming your data so that it fits within a specific scale, like 0-100 or 0-1.

- in **normalisation**, you're changing the shape of the distribution of your data.

- **Let's see some examples!**

Task:

- Create a .py script and
- save it to a folder.

```
# modules we'll use
import pandas as pd
import numpy as np

# for Box-Cox Transformation
from scipy import stats

# for min_max scaling
from mlxtend.preprocessing import minmax_scaling

# plotting modules
import seaborn as sns
import matplotlib.pyplot as plt

# set seed for reproducibility
np.random.seed(0)
```

Scaling/Normalising Dataframes

- **Troubleshooting!**

```
>>> from mlxtend.preprocessing import minmax_scaling
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'mlxtend'
>>> exit()
root@45df587da064:/# pip3 install mlxtend
```

Scaling/Normalising Dataframes

What's the difference between **scaling** and **normalisation**?

- in **scaling**, you're changing the range of your data

you're transforming your data so that it fits within a specific scale, like 0-100 or 0-1.

- in **normalisation**, you're changing the shape of the distribution of your data.

Scaling example

```
# generate 1000 data points randomly drawn from an exponential distribution
original_data = np.random.exponential(size=1000)

# mix-max scale the data between 0 and 1
scaled_data = minmax_scaling(original_data, columns=[0])

# plot both together to compare
fig, ax = plt.subplots(1, 2, figsize=(15, 3))
sns.histplot(original_data, ax=ax[0], kde=True, legend=False)
ax[0].set_title("Original Data")
sns.histplot(scaled_data, ax=ax[1], kde=True, legend=False)
ax[1].set_title("Scaled data")
plt.show()  ||  plt.savefig("plotScale.png")
```

Scaling/Normalising Dataframes

What's the difference between **scaling** and **normalisation**?

- in **scaling**, you're changing the range of your data

you're transforming your data so that it fits within a specific scale, like 0-100 or 0-1.

- in **normalisation**, you're changing the shape of the distribution of your data.

Normalisation example

```
# normalize the exponential data with boxcox
normalized_data = stats.boxcox(original_data)

# plot both together to compare
fig, ax=plt.subplots(1, 2, figsize=(15, 3))
sns.histplot(original_data, ax=ax[0], kde=True, legend=False)
ax[0].set_title("Original Data")
sns.histplot(normalized_data[0], ax=ax[1], kde=True, legend=False)
ax[1].set_title("Normalized data")
plt.show() II   plt.savefig("plotNorm.png")
```

Exploratory Data Analysis

Exploratory Data Analysis (EDA) is a technique to analyze data using some visual Techniques. With this technique, we can get detailed information about the statistical summary of the data. We will also be able to deal with the duplicates values, outliers, and also see some trends or patterns present in the dataset.

Task: (i) Run and understand the script: iris_explorDataAnal.py.

More Examples

https://www.w3schools.com/python/default.asp

https://www.w3resource.com/python-exercises/

https://www.kaggle.com/code/kamilpolak/tutorial-how-to-use-pivot-table-in-pandas

https://plotly.com/python/plotly-express/