

# PYTHON FOR DATA SCIENCE – 23PCS2CC6

STAFF : DN

I M.Sc CS

---

## UNIT II

### TOPIC 1: SEQUENCES

Sequences are an object type in Python that allows the user to store data one after each other. Operations can be performed on a sequence, to examine and manipulate the stored items. There are several different types of sequence objects in Python.

A sequence is a positionally ordered collection of items. And you can refer to any item in the sequence by using its index number e.g., `s[0]` and `s[1]`.

In Python, the sequence index starts at 0, not 1. So the first element is `s[0]` and the second element is `s[1]`. If the sequence `s` has `n` items, the last item is `s[n-1]`.

Python has the following built-in sequence types: lists, bytearrays, strings, tuples, range, and bytes. Python classifies sequence types as mutable and immutable.

The mutable sequence types are lists and bytearrays while the immutable sequence types are strings, tuples, range, and bytes.

A sequence can be homogeneous or heterogeneous. In a homogeneous sequence, all elements have the same type. For example, strings are homogeneous sequences where each element is of the same type.

Lists, however, are heterogeneous sequences where you can store elements of different types including integer, strings, objects, etc.

In general, homogeneous sequence types are more efficient than heterogeneous in terms of storage and operations.

### SYNTAX

`len(seq)`

The following example uses the `len` function to get the number of items in the cities list:

```
cities = ['San Francisco', 'New York', 'Washington DC']  
  
print(len(cities))
```

---

## TOPIC 2: INTRODUCTION TO LIST

Lists are used to store multiple items in a single variable. Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.

Lists are created using square brackets:

Example Create a List:

```
thislist = ["apple", "banana", "cherry"]  
  
print(thislist)
```

List Items List items are ordered, changeable, and allow duplicate values.

List items are indexed, the first item has index [0], the second item has index [1] etc. Ordered When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list  
Changeable The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Allow Duplicates Since lists are indexed, lists can have items with the same value:

Example Lists allow duplicate values:

```
thislist = ["apple", "banana", "cherry", "apple", "cherry"]  
  
print(thislist)
```

List Length To determine how many items a list has, use the len() function:

Example

Print the number of items in the list:

```
thislist = ["apple", "banana", "cherry"]
```

```
print(len(thislist))
```

List Items - Data Types List items can be of any data type:

Example

String, int and boolean data types:

```
list1 = ["apple", "banana", "cherry"] list2 = [1, 5, 7, 9, 3]
```

```
list3 = [True, False, False]
```

A list can contain different data types:

Example

A list with strings, integers and boolean values:

```
list1 = ["abc", 34, True, 40, "male"] type()
```

Example

What is the data type of a list?

```
mylist = ["apple", "banana", "cherry"]
```

```
print(type(mylist))
```

---

### TOPIC 3: LIST SLICING

In Python, list slicing is a common practice and it is the most used technique for programmers to solve efficient problems. Consider a Python list, in order to access a range of elements in a list, you need to slice a list. One way to do this is to use the simple slicing operator i.e. colon(:). With this operator, one can specify where to start the slicing, where to end, and specify the step. List slicing returns a new list from the existing list.

#### **Python List Slicing Syntax**

The format for list slicing is of Python List Slicing is as follows:

```
Lst[ Initial : End : IndexJump ]
```

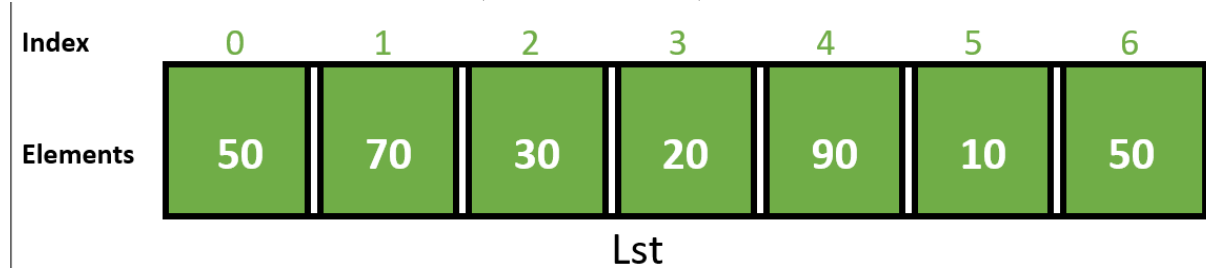
If *Lst* is a list, then the above expression returns the portion of the list from index *Initial* to index *End*, at a step size *IndexJump*.

#### **Indexing in Python List**

Indexing is a technique for accessing the elements of a Python List. There are various ways by which we can access an element of a list.

## Positive Indexes

In the case of Positive Indexing, the first element of the list has the index number 0, and the last element of the list has the index number N-1, where N is the total number of elements in the list (size of the list).



## Positive Indexing of a Python List

### Example:

In this example, we will display a whole list using positive index slicing.

```
# Initialize list

Lst = [50, 70, 30, 20, 90, 10, 50]

# Display list

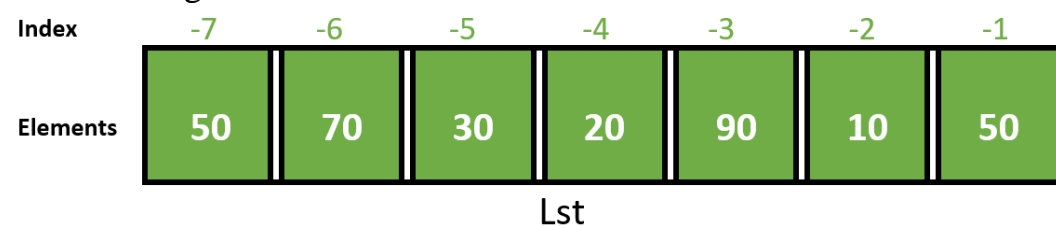
print(Lst[::])
```

### Output:

```
[50, 70, 30, 20, 90, 10, 50]
```

## Negative Indexes

The below diagram illustrates a list along with its negative indexes. Index -1 represents the last element and -N represents the first element of the list, where N is the length of the list.



## Negative Indexing of a Python List

### Example:

In this example, we will access the elements of a list using negative indexing.

```
# Initialize list

Lst = [50, 70, 30, 20, 90, 10, 50]

# Display list

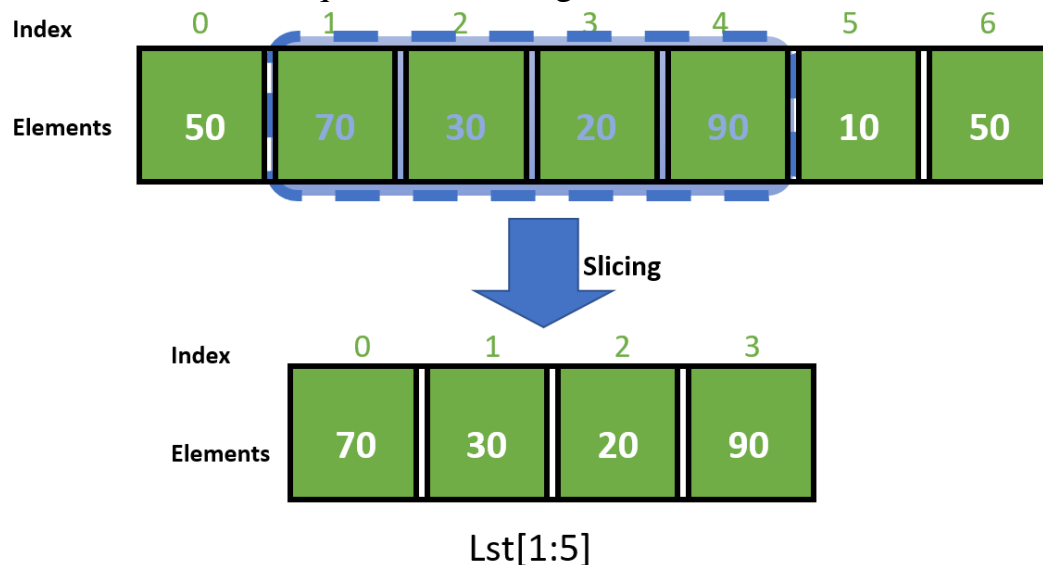
print(Lst[-7::1])
```

### Output:

```
[50, 70, 30, 20, 90, 10, 50]
```

### Slicing

As mentioned earlier list slicing in Python is a common practice and can be used both with positive indexes as well as negative indexes. The below diagram illustrates the technique of list slicing:



## Python List Slicing

### Example:

In this example, we will transform the above illustration into Python code.

- Python3

```
# Initialize list

Lst = [50, 70, 30, 20, 90, 10, 50]

# Display list

print(Lst[1:5])
```

**Output:**

[70, 30, 20, 90]

**Examples of List Slicing in Python**

**Example 1:** Leaving any argument like Initial, End, or IndexJump blank will lead to the use of default values i.e. 0 as Initial, length of the list as End, and 1 as IndexJump.

```
# Initialize list

List = [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Show original list

print("Original List:\n", List)

print("\nSliced Lists: ")
```

```
# Display sliced list
```

```
print(List[3:9:2])
```

```
# Display sliced list
```

```
print(List[::2])
```

```
# Display sliced list
```

```
print(List[::])
```

**Output:**

Original List:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Sliced Lists:

```
[4, 6, 8]
```

```
[1, 3, 5, 7, 9]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

---

**TOPIC 4: FINDING ITEMS IN A LIST USING IN OPERATOR**

To simply check if a list contains a particular item in Python you can use the `in` operator like this:

```
fruit = ['apple', 'banana', 'orange', 'lime']
```

```
if 'lime' in fruit:
```

```
    print('Lime found!')
```

OUTPUT

```
Lime found!
```

---

## TOPIC 5: LIST METHODS AND USEFUL BUILT-IN FUNCTIONS

- [Python List index\(\)](#)
- **[Python List append\(\)](#)**
- [Python List extend\(\)](#)
- [Python List insert\(\)](#)
- [Python List remove\(\)](#)
- [Python List count\(\)](#)
- [Python List pop\(\)](#)
- [Python List reverse\(\)](#)
- [Python List sort\(\)](#)
- [Python List copy\(\)](#) •
- [Python List clear\(\)](#)
- **Python LIST index()**

List index starts from 0.

The `index()` method returns the index of the specified element in the list.

### Example

```
animals = ['cat', 'dog', 'rabbit', 'horse']
```

```
index=animals.index('dog')
```

output

1

### Python List append()

The `append()` method adds an item to the end of the list.

### Example

```
currencies = ['Dollar', 'Euro', 'Pound']
```

```
# append 'Yen' to the list
```



```
currencies.append('Yen')
print(currencies)
```

```
# Output: ['Dollar', 'Euro', 'Pound', 'Yen']
```

## Python List extend()

The `extend()` method adds all the elements of an iterable (list, tuple, string etc.) to the end of

### Example

```
# create a list prime_numbers
```

```
= [2, 3, 5]
```

```
# create another list
```

```
numbers = [1, 4]
```

```
# add all elements of prime_numbers to numbers
```

```
numbers.extend(prime_numbers)
```

```
print('List after extend():', numbers)
```

```
# Output: List after extend(): [1, 4, 2, 3, 5]
```

```
# 'o' is inserted at index 3 (4th position)
```

```
vowel.insert(3, 'o')
```

```
print('List:', vowel)
```

```
# Output: List: ['a', 'e', 'i', 'o', 'u']
```

## Python List remove()

The `remove()` method removes the first matching element (which is passed as an argument) from

### Example

```
# create a list prime_numbers = [2, 3,
5, 7, 9, 11]

# remove 9 from the list
prime_numbers.remove(9)
print('Updated List: ', prime_numbers)

# Output: Updated List: [2, 3, 5, 7, 11]
```

### Python List count()

The `count()` method returns the number of times the specified element appears in the list.

#### Example

```
# create a list numbers = [2, 3, 5,
2, 11, 2, 7]

# check the count of 2
count =
numbers.count(2)
print('Count of 2:', count)

# Output: Count of 2: 3
```

### Python List pop()

The `pop()` method removes the item at the given index from the list and returns the removed

#### Example

```
# create a list of prime numbers prime_numbers = [2, 3, 5, 7]

# remove the element at index 2 removed_element = prime_numbers.pop(2)
print('Removed Element:', removed_element) print('Updated List:',
prime_numbers)

# Output:
# Removed Element: 5
# Updated List: [2, 3, 7]
```

## Python List reverse()

The `reverse()` method reverses the elements of the list.

### Example

```
# create a list of prime numbers
prime_numbers = [2, 3, 5, 7]

# reverse the order of list elements
prime_numbers.reverse()

print('Reversed List:', prime_numbers)

# Output: Reversed List: [7, 5, 3, 2]
```

## Python List sort()

The `sort()` method sorts the items of a list in ascending or descending order.

### Example

```
prime_numbers = [11, 3, 7, 5, 2]

# sorting the list in ascending order
prime_numbers.sort()

print(prime_numbers)

# Output: [2, 3, 5, 7, 11]
```

## Python List copy()

The `copy()` method returns a shallow copy of the list.

### Example

```
# mixed list
prime_numbers p = [2, 3,
5]
```

```
# copying a list
numbers = prime_numbers.copy()
print('Copied List:', numbers)

# Output: Copied List: [2, 3, 5]
```

### Python List clear()

The `clear()` method removes all items from the list.

#### Example

```
prime_numbers = [2, 3, 5, 7, 9, 11]
# remove all elements
prime_numbers.clear()

# Updated prime_numbers
List
print('List after clear():', prime_numbers)
# Output: List after clear(): []
```

---

## TOPIC 6: COPYING LISTS

You cannot copy a list simply by typing `list2 = list1`, because: `list2` will only be a *reference* to `list1`, and changes made in `list1` will automatically also be made in `list2`.

There are ways to make a copy, one way is to use the built-in List method `copy()`.

### Example Get your own Python Server

Make a copy of a list with the `copy()` method:

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

OUTPUT

## TOPIC 7: PROCESSING LIST

In Python, the sequence of various data types is stored in a list. A list is a collection of different kinds of values or items. Since Python lists are mutable, we can change their elements after forming. The comma (,) and the square brackets [enclose the List's items] serve as separators.

Although six Python data types can hold sequences, the List is the most common and reliable form. A list, a type of sequence data, is used to store the collection of data. Tuples and Strings are two similar data formats for sequences.

Lists written in Python are identical to dynamically scaled arrays defined in other languages, such as Array List in Java and Vector in C++. A list is a collection of items separated by commas and denoted by the symbol [].

### Characteristics of Lists

The characteristics of the List are as follows:

- The lists are in order.
- The list element can be accessed via the index.
- The mutable type of List is
- The rundowns are changeable sorts.
- The number of various elements can be stored in a list.

1. # a simple list
2. list1 = [1, 2, "Python", "Program", 15.9]
3. list2 = ["Amy", "Ryan", "Henry", "Emma"]
- 4.
5. # printing the list
6. **print**(list1)
7. **print**(list2)
- 8.

9. # printing the type of list
10. **print**(type(list1))
11. **print**(type(list2))

### Output:

```
[1, 2, 'Python', 'Program', 15.9]
['Amy', 'Ryan', 'Henry', 'Emma']
< class ' list ' >
< class ' list ' >
```

### Updating List Values

Due to their mutability and the slice and assignment operator's ability to update their values, lists are Python's most adaptable data structure. Python's `append()` and `insert()` methods can also add values to a list.

Consider the following example to update the values inside the List.

1. # updating list values
2. `list = [1, 2, 3, 4, 5, 6]`
3. **print**(list)
4. # It will assign value to the value to the second index
5. `list[2] = 10`
6. **print**(list)
7. # Adding multiple-element
8. `list[1:3] = [89, 78]`
9. **print**(list)
10. # It will add value at the end of the list
11. `list[-1] = 25`
12. **print**(list)

### Output:

```
[1, 2, 3, 4, 5, 6]
[1, 2, 10, 4, 5, 6]
[1, 89, 78, 4, 5, 6]
[1, 89, 78, 4, 5, 25]
```

The list elements can also be deleted by using the **del** keyword. Python also provides us the **remove()** method if we do not know which element is to be deleted from the list.

## Python List Operations

The concatenation (+) and repetition (\*) operators work in the same way as they were working with the strings. The different operations of list are

1. Repetition
2. Concatenation
3. Length
4. Iteration
5. Membership

Let's see how the list responds to various operators.

### 1. Repetition

The redundancy administrator empowers the rundown components to be rehashed on different occasions.

#### Code

1. # repetition of list
2. # declaring the list
3. list1 = [12, 14, 16, 18, 20]
4. # repetition operator \*
5. l = list1 \* 2
6. **print(l)**

#### Output:

```
[12, 14, 16, 18, 20, 12, 14, 16, 18, 20]
```

### 2. Concatenation

It concatenates the list mentioned on either side of the operator.

#### Code

1. # concatenation of two lists
2. # declaring the lists
3. list1 = [12, 14, 16, 18, 20]
4. list2 = [9, 10, 32, 54, 86]
5. # concatenation operator +
6. l = list1 + list2
7. **print(l)**

**Output:**

```
[12, 14, 16, 18, 20, 9, 10, 32, 54, 86]
```

### 3. Length

It is used to get the length of the list

**Code**

1. # size of the list
2. # declaring the list
3. list1 = [12, 14, 16, 18, 20, 23, 27, 39, 40]
4. # finding length of the list
5. **len(list1)**

**Output:**

```
9
```

### 4. Iteration

The for loop is used to iterate over the list elements.

**Code**

1. # iteration of the list
2. # declaring the list
3. list1 = [12, 14, 16, 39, 40]
4. # iterating
5. **for i in list1:**
6.     **print(i)**

**Output:**



```
12  
14  
16  
39  
40
```

## 5. Membership

It returns true if a particular item exists in a particular list otherwise false.

### Code

1. # membership of the list
2. # declaring the list
3. list1 = [100, 200, 300, 400, 500]
4. # true will be printed if value exists
5. # and false if not
- 6.
7. **print(600 in list1)**
8. **print(700 in list1)**
9. **print(1040 in list1)**
- 10.
11. **print(300 in list1)**
12. **print(100 in list1)**
13. **print(500 in list1)**

### Output:

```
False  
False  
False  
True  
True  
True
```

### Iterating a List

A list can be iterated by using a for - in loop. A simple list containing four strings, which can be iterated as follows.

### Code

1. # iterating a list

2. `list = ["John", "David", "James", "Jonathan"]`
3. `for i in list:`
4. `# The i variable will iterate over the elements of the List and contains each element in each iteration.`
5. `print(i)`

**Output:**

```
John
David
James
Jonathan
```

---

## TOPIC 8: TWO DIMENSIONAL LIST

In the case of a list, our old-fashioned one-dimensional list looks like this:

```
myList = [0,1,2,3]
```

And a two-dimensional list looks like this:

```
myList = [ [0,1,2,3], [3,2,1,0], [3,5,6,1], [3,8,3,4] ]
```

For our purposes, it is better to think of the two-dimensional list as a matrix. A matrix can be thought of as a grid of numbers, arranged in rows and columns, kind of like a bingo board. We might write the two-dimensional list out as follows to illustrate this point:

```
myList = [ [0, 1, 2, 3],
            [3, 2, 1, 0],
            [3, 5, 6, 1],
            [3, 8, 3, 4] ]
```

### EXAMPLE

```
myList= [ [0, 1, 2]
           [3, 4, 5]
           [6, 7, 8] ]

for i in len(myList):

    for j in len(myList[0]):

        print(myList[i][j] )
```

## OUTPUT

```
[ [0, 1, 2]
  [3, 4, 5]
  [6, 7, 8] ]
```

---

## TOPIC 9: TUPLES

A comma-separated group of items is called a Python triple. The ordering, settled items, and reiterations of a tuple are to some degree like those of a rundown, but in contrast to a rundown, a tuple is unchanging.

The main difference between the two is that we cannot alter the components of a tuple once they have been assigned. On the other hand, we can edit the contents of a list.

### Example

("Suzuki", "Audi", "BMW", " Skoda ") is a tuple.

### Features of Python Tuple

- Tuples are an immutable data type, meaning their elements cannot be changed after they are generated.

- Each element in a tuple has a specific order that will never change because tuples are ordered sequences.

### Forming a Tuple:

All the objects-also known as "elements"-must be separated by a comma, enclosed in parenthesis (). Although parentheses are not required, they are recommended.

(dictionary, string, float, list, etc.), can be contained in a tuple.

```
# Python program to show how to create a tuple
# Creating an empty tuple
empty_tuple = ()
print("Empty tuple: ", empty_tuple)

# Creating tuple having integers
int_tuple = (4, 6, 8, 10, 12, 14)
print("Tuple with integers: ", int_tuple)

# Creating a tuple having objects of different data types
mixed_tuple = (4, "Python", 9.3)
print("Tuple with different data types: ", mixed_tuple)

# Creating a nested tuple
nested_tuple = ("Python", {4: 5, 6: 2, 8:2}, (5, 3, 5, 6))
print("A nested tuple: ", nested_tuple)
```

### Output:

```
Empty tuple: ()
Tuple with integers: (4, 6, 8, 10, 12, 14)
Tuple with different data types: (4, 'Python', 9.3)
A nested tuple: ('Python', {4: 5, 6: 2, 8: 2}, (5, 3, 5, 6))
```

Parentheses are not necessary for the construction of multiples. This is known as triple pressing.

### Code

```
# Python program to create a tuple without using parentheses
```

```

# Creating a tuple
tuple_ = 4, 5.7, "Tuples", ["Python", "Tuples"]
# Displaying the tuple created
print(tuple_)
# Checking the data type of object tuple_
print(type(tuple_))
# Trying to modify tuple_
try:
    tuple_[1] = 4.2
except:
    print(TypeError)

```

### Output:

```

(4, 5.7, 'Tuples', ['Python', 'Tuples'])
<class 'tuple'>
<class 'TypeError'>

```

### Accessing Tuple Elements

A tuple's objects can be accessed in a variety of ways.

### Indexing

**Indexing** We can use the index operator [] to access an object in a tuple, where the index starts at 0.

The indices of a tuple with five items will range from 0 to 4. An Index Error will be raised assuming we attempt to get to a list from the Tuple that is outside the scope of the tuple record. An index above four will be out of range in this scenario.

```

# Python program to show how negative indexing works in Python tuples
# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Collection")
# Printing elements using negative indices
print("Element at -1 index: ", tuple_[-1])
print("Elements between -4 and -1 are: ", tuple_[-4:-1])

```

### Output:

Element at -1 index: Collection

Elements between -4 and -1 are: ('Python', 'Tuple', 'Ordered')

## Slicing

Tuple slicing is a common practice in Python and the most common way for programmers to deal with practical issues. Look at a tuple in Python. Slice a tuple to access a variety of its elements. Using the colon as a straightforward slicing operator (:) is one strategy.

To gain access to various tuple elements, we can use the slicing operator colon (:).

## Code

```
# Python program to show how slicing works in Python tuples
# Creating a tuple
tuple_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Objects")
# Using slicing to access elements of the tuple
print("Elements between indices 1 and 3: ", tuple_[1:3])
# Using negative indexing in slicing
print("Elements between indices 0 and -4: ", tuple_[:-4])
# Printing the entire tuple by using the default start and end values.
print("Entire tuple: ", tuple_[:])
```

## Output:

Elements between indices 1 and 3: ('Tuple', 'Ordered')

Elements between indices 0 and -4: ('Python', 'Tuple')

Entire tuple: ('Python', 'Tuple', 'Ordered', 'Immutable', 'Collection', 'Objects')

## Deleting a Tuple

A tuple's parts can't be modified, as was recently said. We are unable to eliminate or remove tuple components as a result.

However, the keyword `del` can completely delete a tuple.

---

## TOPIC 10: MANIPULATING STRINGS

You might have learned that you need to *declare* or *type* variables before you can store anything in them. This is not necessary when working with strings in

Python. We can create a string simply by putting content wrapped with quotation marks into it with an equal sign (=):

```
message = "Hello World"
```

String Operators: Adding and Multiplying string-operators-adding-and-multiplying

A string is a type of object, one that consists of a series of characters. Python already knows how to deal with a number of general-purpose and powerful representations, including strings. One way to manipulate strings is by using *string operators*. These operators are represented by symbols that you likely associate with mathematics, such as +, -, \*, /, and =. When used with strings, they perform actions that are similar to, but not the same as, their mathematical counterparts.

### Concatenate concatenate

This term means to join strings together. The process is known as *concatenating* strings and it is done using the plus (+) operator. Note that you must be explicit about where you want blank spaces to occur by placing them between single quotation marks also.

In this example, the string “message1” is given the content “hello world”.

```
message1 = 'hello' + ' ' + 'world'  
print(message1)  
-> hello world
```

### Multiply multiply

If you want multiple copies of a string, use the multiplication (\*) operator. In this example, string *message2a* is given the content “hello” times three; string *message2b* is given content “world”; then we print both strings.

```
message2a = 'hello ' * 3  
message2b = 'world'  
print(message2a + message2b)  
-> hello hello hello world
```

### Append append

What if you want to add material to the end of a string successively? There is a special operator for that (+=).

```
message3 = 'howdy'
```

```
message3 += '  
message3 += 'world'  
print(message3)  
-> howdy world
```

## String Methods: Finding, Changing

In addition to operators, Python comes pre-installed with dozens of string methods that allow you to do things to strings. Used alone or in combination, these methods can do just about anything you can imagine to strings. The good news is that you can reference a list of String Methods on the [Python website](#), including information on how to use each properly. To make sure that you've got a basic grasp of string methods, what follows is a brief overview of some of the more commonly used ones:

### Length `length`

You can determine the number of characters in a string using `len`. Note that the blank space counts as a separate character.

```
message4 = 'hello' + ' ' + 'world'  
print(len(message4))  
-> 11
```

### Find `find`

You can search a string for a *substring* and your program will return the starting index position of that substring. This is helpful for further processing. Note that indexes are numbered from left to right and that the count starts with position 0, not 1.

```
message5 = "hello world"  
message5a = message5.find("worl")  
print(message5a)  
-> 6
```

If the substring is not present, the program will return a value of -1.

```
message6 = "Hello World"  
message6b = message6.find("squirrel")  
print(message6b)  
-> -1
```

### Lower Case `lower-case`

Sometimes it is useful to convert a string to lower case. For example, if we standardize case it makes it easier for the computer to recognize that “Sometimes” and “sometimes” are the same word.



```
message7 = "HELLO WORLD"
message7a = message7.lower()
print(message7a)
-> hello world
```

The opposite effect, raising characters to upper case, can be achieved by changing `.lower()` to `.upper()`.

### Replace replace

If you need to replace a substring throughout a string you can do so with the `replace` method.

```
message8 = "HELLO WORLD"
message8a = message8.replace("L", "pizza")
print(message8a)
-> HEpizzapizzaO WORpizzaD
```

### Slice slice

If you want to slice off unwanted parts of a string from the beginning or end you can do so by creating a substring. The same kind of technique also allows you to break a long string into more manageable components.

```
message9 = "Hello World"
message9a = message9[1:8]
print(message9a)
-> ello Wo
```

You can substitute variables for the integers used in this example.

```
startLoc = 2
endLoc = 8
message9b = message9[startLoc: endLoc]
print(message9b)
-> llo Wo
```

This makes it much easier to use this method in conjunction with the `find` method as in the next example, which checks for the letter “d” in the first six characters of “Hello World” and correctly tells us it is not there (-1). This technique is much more useful in longer strings – entire documents for example. Note that the absence of an integer before the colon signifies we want to start at the beginning of the string. We could use the same technique to tell the program to go all the way to the end by putting no integer after the colon. And remember, index positions start counting from 0 rather than 1.

```
message9 = "Hello World"
print(message9[:5].find("d"))
-> -1
```

There are lots more, but the string methods above are a good start. Note that in this last example, we are using square brackets instead of parentheses. This difference in *syntax* signals an important distinction. In Python, parentheses are usually used to *pass an argument* to a function. So when we see something like

```
print(len(message7))
```

it means pass the string *message7* to the function `len` then send the returned value of that function to the `print` statement to be printed. If a function can be called without an argument, you often have to include a pair of empty parentheses after the function name anyway. We saw an example of that, too:

```
message7 = "HELLO WORLD"  
message7a = message7.lower()  
print(message7a)  
-> hello world
```

This statement tells Python to apply the `lower` function to the string *message7* and store the returned value in the string *message7a*.

The square brackets serve a different purpose. If you think of a string as a sequence of characters, and you want to be able to access the contents of the string by their location within the sequence, then you need some way of giving Python a location within a sequence. That is what the square brackets do: indicate a beginning and ending location within a sequence as we saw when using the slice method.

---

## TOPIC 11: BASIC STRING OPERATIONS

What's a String in Python?

**A python string is a list of characters in an order. A character is anything you can type on the keyboard in one keystroke, like a letter, a number, or a backslash. Strings can also have spaces, tabs, and newline characters.**

ex

```
myStr="hello world"
```

**We can also define an empty string that has 0 characters as shown below.**

```
myStr=""
```

In python, every string starts with and ends with quotation marks i.e. single quotes ' ', double quotes " " or triple quotes "" "" "".

---

## String Manipulation

### Creating a string

---

- i. Access Characters
- ii. Finding length
- iii. Finding a character
- iv. Counting
- v. String slicing

#### Create a String in Python

To create a string with given characters, you can assign the characters to a variable after enclosing them in double quotes or single quotes as shown below.

```
word = "Hello World"  
print(word)
```

Output:

```
Hello World
```

#### Access Characters in a String in Python

To access characters of a string, we can use the python indexing operator [ ] i.e.

square brackets to access characters in a string as shown below.

```
word = "Hello World"  
print("The word is:",word)  
letter=word[0]  
print("The letter is:",letter)
```

Output:

```
The word is: Hello World
```

```
The letter is: H
```

## Find Length of a String in Python

To find the length of a string in Python, we can use the `len()` function. The `len()` function takes a string as input argument and returns the length of the string as shown below.

```
word = "Hello World"
print("The string is:",word)
length=len(word)
print("The length of the string is:",length)
```

Output:

```
The string is: Hello World
The length of the string is: 11
```

## Find a Character in a String in Python

To find the index of a character in a string, we can use the `find()` method.

```
word = "Hello World"
print("The string is:",word)
character="W"
print("The character is:",character)
position=word.find(character)
print("The position of the character in the string is:",position)
```

Output:

```
The string is: Hello World
The character is: W
The position of the character in the string is: 6
```

## Count the Number of Spaces in a String in Python

Spaces are also characters. Hence, you can use the `count()` method count the number of spaces in a string in Python.

```
myStr = "Count, the number of spaces"
```

```
print("The string is:",myStr)
character=" "
position=myStr.count(character)
print("The number of spaces in the string is:",position)
```

Output:

```
The string is: Count, the number of spaces
```

```
The number of spaces in the string is: 4
```

## String Slicing in Python

---

To perform string manipulation in Python, you can use the syntax `string_name[start_index : end_index ]` to get a substring of a string.

```
word = "Hello World"
```

```
print word[0] #get one char of the word
```

```
print word[0:1] #get one char of the word (same as above)
```

```
print word[0:3] #get the first three char
```

```
print word[:3] #get the first three char
```

```
print word[-3:] #get the last three char
```

```
print word[3:] #get all but the three first char
```

```
print word[:-3] #get all but the three last character word = "Hello World"
```

```
word[start:end] # items start through end-1 word[start:] # items start through the
rest of the list word[:end] # items from the beginning through end-1 word[:] # a
copy of the whole list
```

---

## TOPIC 12: DICTIONARIES

```
thisdict = { "brand": "Ford",
             "model": "Mustang",
             "year": 1964
           }
```

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered\*, changeable and do not allow duplicates.

As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

Dictionaries are written with curly brackets, and have keys and values:

Example

Create and print a dictionary:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}
```

Print (thisdict)

Dictionary Items

Dictionary items are ordered, changeable, and does not allow duplicates.

Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

### Example

Print the "brand" value of the dictionary:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(thisdict["brand"])
```

Ordered or Unordered?

As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.

Unordered means that the items does not have a defined order, you cannot refer to an item by using an index.

### Changeable

Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

---

### Duplicates Not Allowed

Dictionaries cannot have two items with the same key:

#### **Example**

Duplicate values will overwrite existing values:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964,  
    "year": 2020  
}  
print(thisdict)
```

### Dictionary Length

To determine how many items a dictionary has, use the len() function:

#### **Example**

Print the number of items in the dictionary:

```
print(len(thisdict))
```

### Dictionary Items - Data Types

The values in dictionary items can be of any data type:

## Example

String, int, boolean, and list data types:

```
thisdict = {  
    "brand": "Ford",  
    "electric": False,  
    "year": 1964,  
    "colors": ["red", "white", "blue"]  
}
```

[Try it Yourself »](#)

## TOPIC 13: SETS

### Set

Sets are used to store multiple items in a single variable.

Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.

A set is a collection which is *unordered*, *unchangeable\**, and *unindexed*.

```
thisset = {"apple", "banana", "cherry"}  
print(thisset)
```

### Set Items

Set items are unordered, unchangeable, and do not allow duplicate values.

---

### Unordered

Unordered means that the items in a set do not have a defined order.

Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

---

### Unchangeable



Set items are unchangeable, meaning that we cannot change the items after the set has been created.

Once a set is created, you cannot change its items, but you can remove items and add new items.

---

Duplicates Not Allowed

Sets cannot have two items with the same value.

### Example

Duplicate values will be ignored:

```
thisset = {"apple", "banana", "cherry", "apple"}
```

```
print(thisset)
```

Get the number of items in a set:

```
thisset = {"apple", "banana", "cherry"}
```

```
print(len(thisset))
```

---

## TOPIC 14: SERIALIZING OBJECTS

### Serialization in Python

**Serialization** is converting an information item in memory into a layout that may be saved or transmitted and later reconstructed into the original object. In Python, serialization permits you to *store complex records systems, consisting of lists, dictionaries, and custom objects*, to a document or transfer them over a community.

Python gives several integrated serialization modules, including *pickle*, *JSON*, and *marshal*.

```
import pickle
# Object to serialize
data = [1, 2, 3, 4, 5]
# Serialize object to a file
with open('data.pkl', 'wb') as file:
    pickle.dump(data, file)
```

```
# Deserialize object from the file
with open('data.pkl', 'rb') as file:
loaded_data = pickle.load(file)
print(loaded_data)
```

### **Output:**

```
[1, 2, 3, 4, 5]
```

---

## **TOPIC 15: CLASSES AND OBJECTS : OOPS**

Python Classes/Objects

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

---

Create a Class

To create a class, use the keyword class:

### **Example**

```
class MyClass:
    x = 5
```

Create Object

Now we can use the class named MyClass to create objects:

### **Example**

Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

The `__init__()` Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in `__init__()` function.

All classes have a function called `__init__()`, which is always executed when the class is being initiated.

Use the `__init__()` function to assign values to object properties, or other operations that are necessary to do when the object is being created:

### Example

Create a class named `Person`, use the `__init__()` function to assign values for `name` and `age`:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

### Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Let us create a method in the `Person` class:

### Example

Insert a function that prints a greeting, and execute it on the `p1` object:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        print("Hello my name is " + self.name)
```

p1 = Person("John", 36)  
p1.myfunc()

---

## TOPIC 16: GETTING MYSQL FOR PYTHON

Python can be used in database applications.

One of the most popular databases is MySQL.

---

### MySQL Database

To be able to experiment with the code examples in this tutorial, you should have MySQL installed on your computer.

You can download a MySQL database at <https://www.mysql.com/downloads/>.

---

### Install MySQL Driver

Python needs a MySQL driver to access the MySQL database.

In this tutorial we will use the driver "MySQL Connector".

We recommend that you use PIP to install "MySQL Connector".

PIP is most likely already installed in your Python environment.

Navigate your command line to the location of PIP, and type the following:

Download and install "MySQL Connector":

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>python -m pip install mysql-connector-python
```

Now you have downloaded and installed a MySQL driver.

### Test MySQL Connector

To test if the installation was successful, or if you already have "MySQL Connector" installed, create a Python page with the following content:

```
demo_mysql_test.py:
```

```
import mysql.connector
```

If the above code was executed with no errors, "MySQL Connector" is installed and ready to be used.

### Create Connection

Start by creating a connection to the database.

Use the username and password from your MySQL database:

```
demo_mysql_connection.py:
```

```
import mysql.connector
```


```
mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword"
)
```

```
print(mydb)
```

---

## TOPIC 17: IMPORTING MYSQL FOR PYTHON

02 March 2023

- 5
- **15682** views
- **2**
  - 
  - 
  - 
  - 
  - 
  -

Retrieving MySQL data from within Python

**This article is part of Robert Sheldon's continuing series on Learning MySQL. To see all of the items in the series, [click here](#).**

Applications of all types commonly access MySQL to retrieve, add, update, or delete data. The applications might be written in Python, Java, C#, or another programming language. Most languages support multiple methods for working with a MySQL database and manipulating its data.

The approach you take when accessing MySQL will depend on the programming language you're using and the connector you choose for interfacing with the database. Whatever approach you take, the same basic principles generally apply to each environment. You must establish a connection to the database and then issue the commands necessary to retrieve or modify the data.

Because MySQL can play such an important role in application development, I wanted to provide you with an overview how to access MySQL data from within your application code. This article demonstrates how to use the MySQL Connector from within Python to establish a connection and run a query.

- MySQL 8.0
- Python 3.10
- PyCharm Community Edition IDE
- MySQL Connector/Python 8.0 module

## Defining a connection to MySQL

When connecting to a MySQL database in Python, you need to take several basic steps:

1. Import the connect method from the MySQL Connector module.
2. Use the connect method to create a connection object that includes your connection details.
3. Use the connection object to run your data-related code.
4. Close the connection.

```
# import the connect method
from mysql.connector import connect
```

```
# define a connection object
conn = connect(
    user = 'root',
```

```
password = 'SqlPW_py@310!ab',
host = 'localhost',
database = 'travel')
```

```
print('A connection object has been created.')
```

```
# close the database connection
conn.close()
```

---

## TOPIC 18: MYSQLDB :CONNECTING WITH A DATABASE

### What is MySQLdb?

MySQLdb is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and is built on top of the MySQL C API.

### Packages to Install

mysql-connector-python

mysql-python

If using anaconda

```
conda install -c anaconda mysql-python
```

```
conda install -c anaconda mysql-connector-python
```

else

```
pip install MySQL-python
```

```
pip install MySQL-python-connector
```

### Import-Package

```
import MySQLdb
```

### How to connect to a remote MySQL database using python?

Before we start you should know the basics of SQL. Now let us discuss the methods used in this code:

- **connect():** This method is used for creating a connection to our database it has four arguments:
  1. Server Name
  2. Database User Name
  3. Database Password
  4. Database Name
- **cursor():** This method creates a cursor object that is capable of executing SQL queries on the database.
- **execute():** This method is used for executing SQL queries on the database. It takes a sql query( as string) as an argument.

- **fetchone():** This method retrieves the next row of a query result set and returns a single sequence, or None if no more rows are available.
- **close() :** This method close the database connection.

## EXAMPLE

```
# Module For Connecting To MySQL database
import MySQLdb

# Function for connecting to MySQL database
def mysqlconnect():
    #Trying to connect
    try:
        db_connection= MySQLdb.connect
        ("Hostname","dbusername","password","dbname")
    # If connection is not successful
    except:
        print("Can't connect to database")
        return 0
    # If Connection Is Successful
    print("Connected")

# Making Cursor Object For Query Execution
cursor=db_connection.cursor()

# Executing Query
cursor.execute("SELECT CURDATE();")

# Above Query Gives Us The Current Date
# Fetching Data
m = cursor.fetchone()

# Printing Result Of Above
print("Today's Date Is ",m[0])

# Closing Database Connection
db_connection.close()

# Function Call For Connecting To Our Database
mysqlconnect()
```

## OUTPUT



Connected

Today's Date Is 2017-11-14

---

# PYTHON FOR DATA SCIENCE – 23PCS2CC6

STAFF : DN

I M.Sc CS

---

## UNIT III

### TOPIC 1: CONTROLLING THE LINE PROPERTIES OF A CHART

There are many properties of a line that can be set, such as the color, dashes, and several others. There are essentially three ways of doing this. Let's take a simple line chart as an example:

#### Creating a Basic Line Plot in Matplotlib

We will start by creating a basic line plot and then customize the line plot to make it look more presentable and informative.

Using `plt.plot()` to create a line plot

To create a line plot, we will use the `plt.plot()` function. This function takes two parameters; the x-axis values and y-axis values. In our case, the date column will be our x-axis values, while the close column will be our y-axis values. Here is the code:

```
# Extract the date and close price columns

dates = df['Date']

closing_price = df['Close']

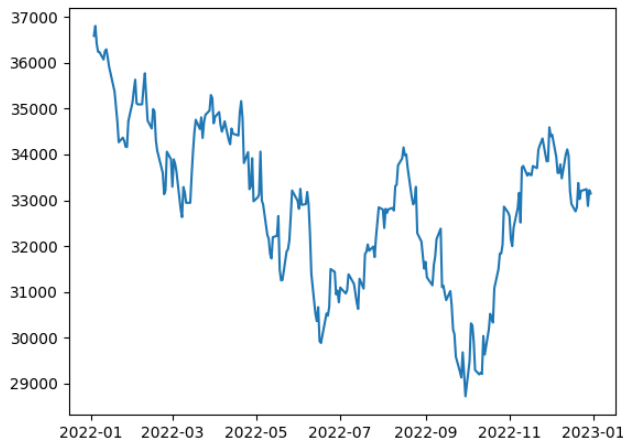
# Create a line plot

plt.plot(dates, closing_price)

# Show the plot

plt.show()
```

When you run the above code, you should see a basic line plot of the DJIA stock.



## Customizing the Line Plot

Matplotlib presents us with plenty of further customizations, which we can utilize per our needs.

### Setting the line color

By default, the `plt.plot()` function plots a blue line. However, you can change the line color by passing a `color` parameter to the function. The `color` parameter can take a string representing the color name or a hexadecimal code.

Here is an example:

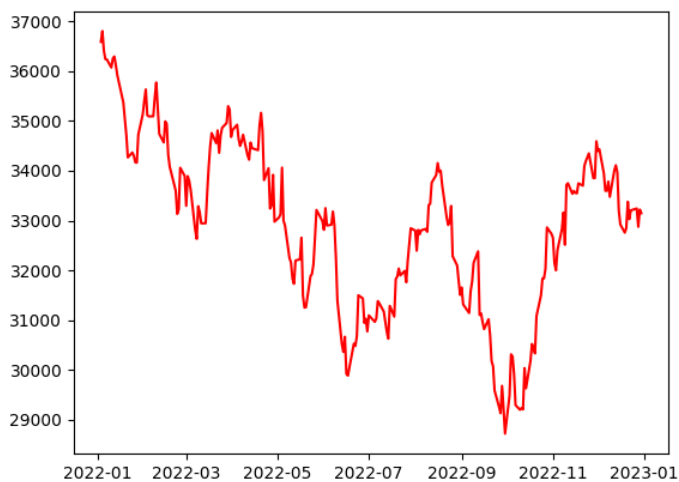
```
# Plot in Red colour

plt.plot(dates, closing_price, color='red')

# Show the plot

plt.show()
```

This code will plot a red line instead of a blue one as shown below:



### Setting the line width

You can also change the line width by passing a `linewidth` parameter to the `plt.plot()` function. The `linewidth` parameter takes a floating-point value representing the line's width.

Here is an example:

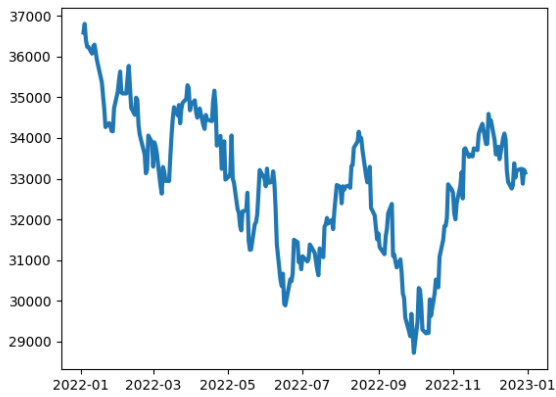
```
# Increasing the linewidth

plt.plot(dates, closing_price, linewidth=3)

# Show the plot

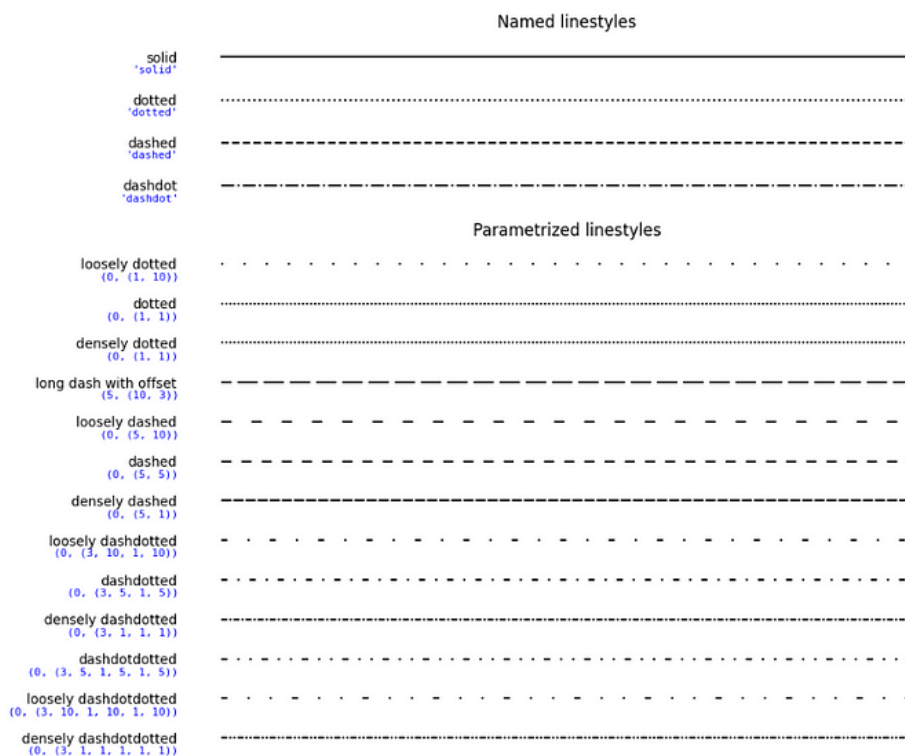
plt.show()
```

This code will plot a line with a width of 3 instead of the default width as shown below:



## Setting the line style

You can change the line style by passing a `linestyle` parameter to the `plt.plot()` function. The `linestyle` parameter takes a string that represents the line style. The [matplotlib documentation](#) provides an extensive list of styles available.



Here's how these can be used in code:

```
# Individually plot lines in solid, dotted, dashed and dashdot
```

```
plt.plot(dates, closing_price, linestyle='solid') # Default line style

plt.plot(dates, closing_price, linestyle='dotted')

plt.plot(dates, closing_price, linestyle='dashed')

plt.plot(dates, closing_price, linestyle='dashdot')

# Show the plot

plt.show()
```

### Adding labels and title

To make the plot more informative, we can add axis labels and a title. We can achieve this by using the `plt.xlabel()`, `plt.ylabel()`, and `plt.title()` functions, respectively.

Here is an example:

```
plt.plot(dates, closing_price, color='red', linewidth=2)

plt.xlabel('Date')

plt.ylabel('Closing Price')

plt.title('DJIA Stock Price')

# Show the plot

plt.show()
```

This code will plot a red line with a width of 2, with the x-axis labeled 'Date,' the y-axis labeled 'Closing Price,' and the title 'DJIA Stock Price.'

## Adding grid lines

We can also add grid lines to our plot to make it more readable. We can achieve this by using the `plt.grid()` function. The `plt.grid()` function takes a boolean value representing whether the grid should be shown.

Here is an example:

```
plt.plot(dates, closing_price, color='red', linewidth=2)
```

```
plt.xlabel('Date')
```

```
plt.ylabel('Closing Price')
```

```
plt.title('DJIA Stock Price')
```

```
# Add the grid
```

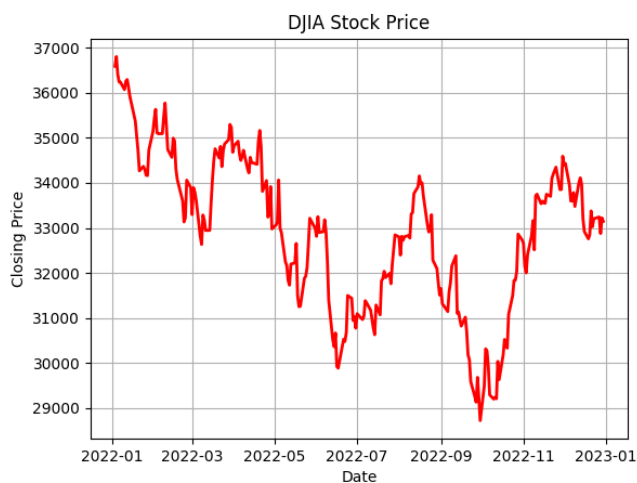
```
plt.grid(True)
```

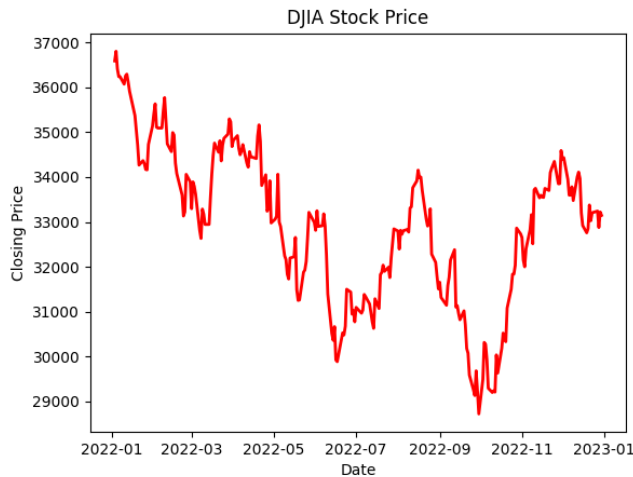
```
# Show the plot
```

```
plt.show()
```

OpenAI

You'd see added grids to the plot:





## TOPIC 2: CREATING MULTIPLE PLOTS

It might need to plot multiple lines on the same graph. To do this, you can call the `plt.plot()` function multiple times with different data for each call. Here is an example:

```
# Line plot of Open and Close prices
```

```
plt.plot(df['Date'], df['Open'])
```

```
plt.plot(df['Date'], df['Close'])
```

```
plt.title('DJIA Open and Close Prices')
```

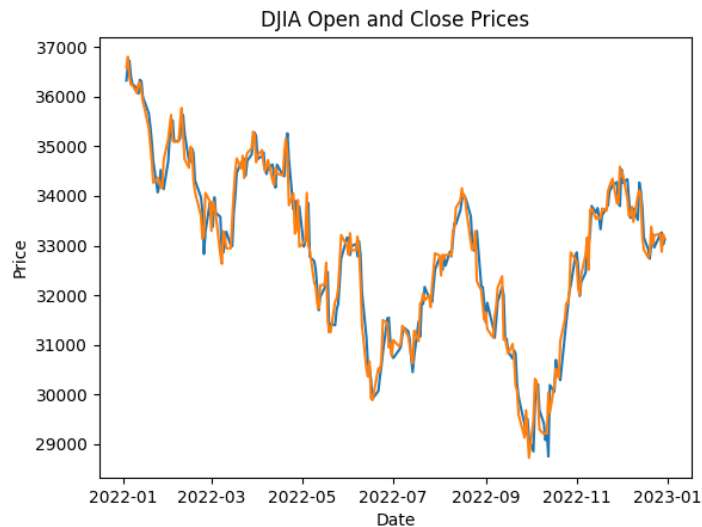
```
plt.xlabel('Date')
```

```
plt.ylabel('Price')
```

```
plt.show()
```

In the above code, we are plotting both the `Open` and `Close` prices of the DJIA stock on the same graph.





### TOPIC 3: PLAYING WITH TEXT

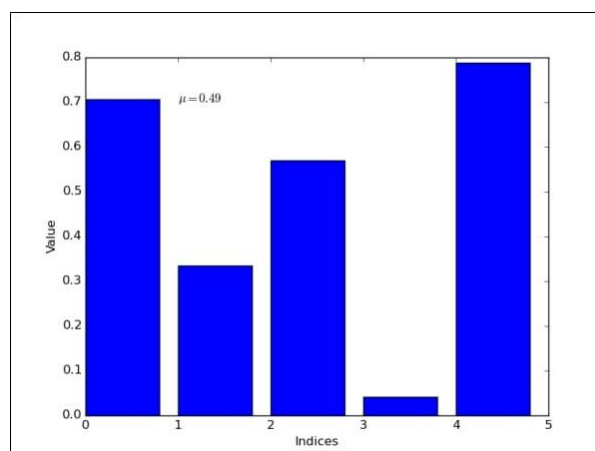
Adding text to your chart can be done by using a simple matplotlib function. You only have to use the text() command to add it to the chart:

```
>>> # Playing with text
>>> n = np.random.random_sample((5,))

>>> plt.bar(np.arange(len(n)), n)
>>> plt.xlabel('Indices')
>>> plt.ylabel('Value')
>>> plt.text(1, .7, r'$\mu=' + str(np.round(np.mean(n), 2)) + '$')

>>> plt.show()
```

In the preceding code, the text() command is used to add text within the plot:



The first parameter takes the  $x$  axis value and the second parameter takes the  $y$  axis value. The third parameter is the text that needs to be added to the plot. The latex expression has been used to plot the  $\mu$  mean within the plot.

A certain section of the chart can be annotated by using the `annotate` command. The `annotate` command will take the text, the position of the section of plot that needs to be pointed at, and the position of the text.

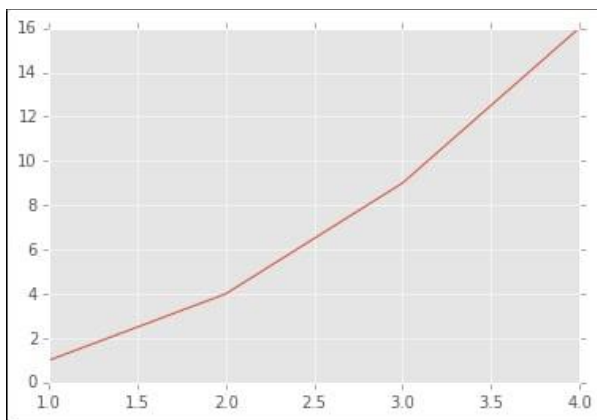
---

## TOPIC 4: STYLING YOUR PLOTS

The `style` package within the `matplotlib` library makes it easier to change the style of the plots that are being plotted. It is very easy to change to the famous `ggplot` style of the R language or use the Nate Silver's website <http://fivethirtyeight.com/> for `fivethirtyeight` style. The following example shows the plotting of a simple line chart with the `ggplot` style:

```
>>> plt.style.use('ggplot')
>>> plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
>>> plt.show()
```

After the preceding code is executed we'll get the following output:



In the preceding code, `plt.style.use()` is used to set the style of the plot. It is a global set, so after it is executed, all the plots that follow will have the same style.

`Matplotlib` is the most popular package or library in Python which is used for data visualization. By using this library we can generate plots and figures, and can easily create raster and vector files without using any other GUIs. With `matplotlib`, we can style the plots like, an HTML webpage is styled by using CSS styles. We just need to import style package of `matplotlib` library.

There are various built-in styles in style package, and we can also write customized style files and, then, to use those styles all you need to import them and apply on the graphs and plots. In this way, we need not write various lines

of code for each plot individually again and again i.e. the code is reusable whenever required.

First, we will import the module:

```
from matplotlib import style
```

To list all the available styles:

```
from matplotlib import style
```

```
print(plt.style.available)
```

### Output:

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

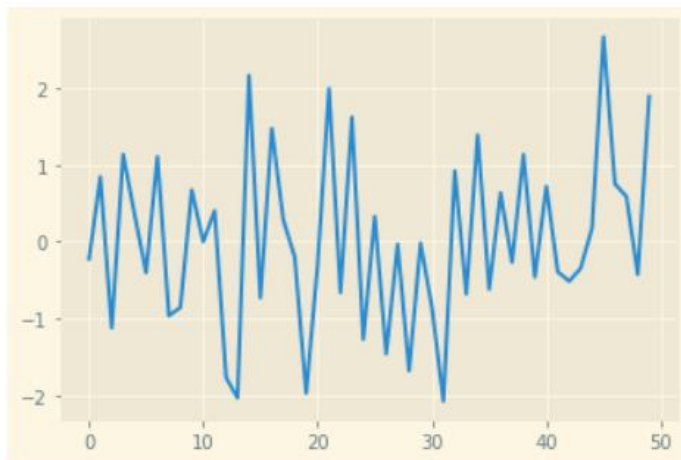
```
import numpy as np  
  
import matplotlib.pyplot as plt  
  
# importing the style package  
  
from matplotlib import style  
  
# creating an array of data for plot  
  
data = np.random.randn(50)  
  
# using the style for the plot  
  
plt.style.use('Solarize_Light2')  
  
  
  
# creating a plot
```

```
plt.plot(data)
```

```
# show plot
```

```
plt.show()
```

### Output:



```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
# importing the style package
```

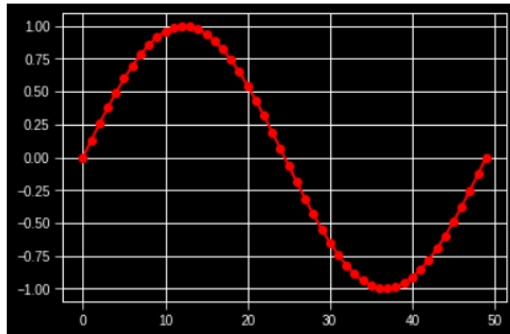
```
from matplotlib import style
```

```
with plt.style.context('dark_background'):
```

```
    plt.plot(np.sin(np.linspace(0, 2 * np.pi)), 'r-o')
```

```
plt.show()
```

### Output:



## TOPIC 5: BOX PLOTS

A **Box Plot** is also known as **Whisker plot** is created to display the summary of the set of data values having properties like minimum, first quartile, median, third quartile and maximum. In the box plot, a box is created from the first quartile to the third quartile, a vertical line is also there which goes through the box at the median. Here x-axis denotes the data to be plotted while the y-axis shows the frequency distribution.

### Creating Box Plot

The [matplotlib.pyplot](#) module of matplotlib library provides `boxplot()` function with the help of which we can create box plots.

#### Syntax:

```
matplotlib.pyplot.boxplot(data, notch=None, vert=None, patch_artist=None, widths=None)
```

#### Parameters:

Attribute	Value
data	array or sequence of array to be plotted
notch	optional parameter accepts boolean values
vert	optional parameter accepts boolean values false and true for horizontal and vertical plot respectively

Attribute	Value
bootstrap	optional parameter accepts int specifies intervals around notched boxplots
usermedians	optional parameter accepts array or sequence of array dimension compatible with data
positions	optional parameter accepts array and sets the position of boxes
widths	optional parameter accepts array and sets the width of boxes
patch_artist	optional parameter having boolean values
labels	sequence of strings sets label for each dataset
meanline	optional having boolean value try to render meanline as full width of box
order	optional parameter sets the order of the boxplot

The data values given to the `ax.boxplot()` method can be a Numpy array or Python list or Tuple of arrays. Let us create the box plot by using `numpy.random.normal()` to create some random data, it takes mean, standard deviation, and the desired number of values as arguments.

### Example:

```
# Import libraries

import matplotlib.pyplot as plt

import numpy as np
```

```
# Creating dataset

np.random.seed(10)

data = np.random.normal(100, 20, 200)

fig = plt.figure(figsize =(10, 7))

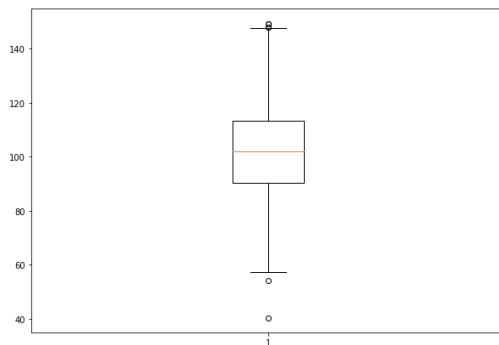
# Creating plot

plt.boxplot(data)

# show plot

plt.show()
```

### Output:



## Customizing Box Plot

### Example 2

```
# Import libraries

import matplotlib.pyplot as plt

import numpy as np

# Creating dataset

np.random.seed(10)

data_1 = np.random.normal(100, 10, 200)
data_2 = np.random.normal(90, 20, 200)
data_3 = np.random.normal(80, 30, 200)
data_4 = np.random.normal(70, 40, 200)
data = [data_1, data_2, data_3, data_4]

fig = plt.figure(figsize =(10, 7))

# Creating axes instance

ax = fig.add_axes([0, 0, 1, 1])

# Creating plot
```

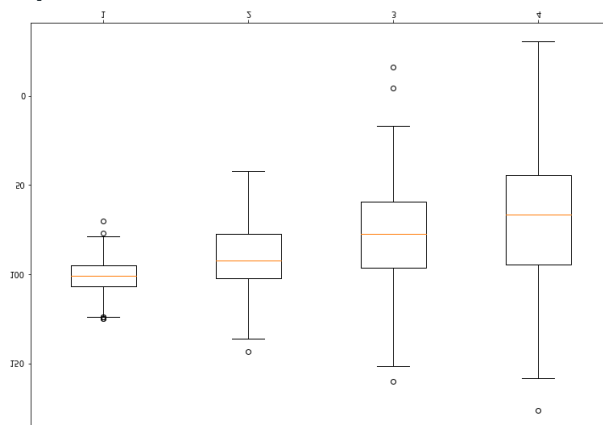


```
bp = ax.boxplot(data)
```

```
# show plot
```

```
plt.show()
```

Output:



## TOPIC 6: HEATMAPS

**Heatmap** is defined as a graphical representation of data using colors to visualize the value of the matrix. In this, to represent more common values or higher activities brighter colors basically reddish colors are used and to represent less common or activity values, darker colors are preferred. Heatmap is also defined by the name of the shading matrix. Heatmaps in Seaborn can be plotted by using the `seaborn.heatmap()` function.

`seaborn.heatmap()`

**Syntax:** `seaborn.heatmap(data, *, vmin=None, vmax=None, cmap=None, center=None, annot_kws=None, linewidths=0, linecolor='white', cbar=True, **kwargs)`

**Important Parameters:**

- **data:** 2D dataset that can be coerced into an ndarray.
- **vmin, vmax:** Values to anchor the colormap, otherwise they are inferred from the data and other keyword arguments.
- **cmap:** The mapping from data values to color space.
- **center:** The value at which to center the colormap when plotting divergent data.
- **annot:** If True, write the data value in each cell.
- **fmt:** String formatting code to use when adding annotations.

- **linewidths:** Width of the lines that will divide each cell.
- **linecolor:** Color of the lines that will divide each cell.
- **cbar:** Whether to draw a colorbar.

All the parameters except data are optional.

## Basic Heatmap

Making a heatmap with the default parameters. We will be creating a 10×10 2-D data using the **randint()** function of the NumPy module.

```
# importing the modules

import numpy as np

import seaborn as sn

import matplotlib.pyplot as plt

# generating 2-D 10x10 matrix of random numbers

# from 1 to 100

data = np.random.randint(low = 1,

                        high = 100,

                        size = (10, 10))

print("The data to be plotted:\n")

print(data)

# plotting the heatmap

hm = sn.heatmap(data = data)
```

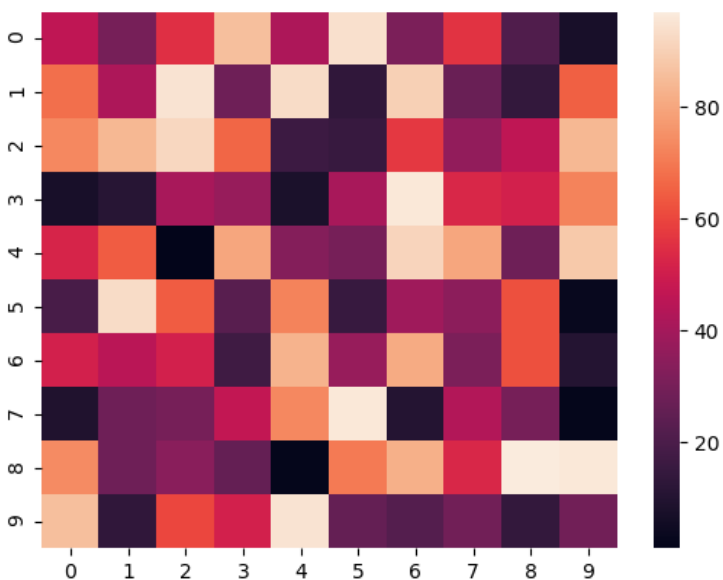
```
# displaying the plotted heatmap
```

```
plt.show()
```

### Output:

The data to be plotted:

```
[[46 30 55 86 42 94 31 56 21 7]  
 [68 42 95 28 93 13 90 27 14 65]  
 [73 84 92 66 16 15 57 36 46 84]  
 [ 7 11 41 37 8 41 96 53 51 72]  
 [52 64 1 80 33 30 91 80 28 88]  
 [19 93 64 23 72 15 39 35 62 3]  
 [51 45 51 17 83 37 81 31 62 10]  
 [ 9 28 30 47 73 96 10 43 30 2]  
 [74 28 34 26 2 70 82 53 97 96]  
 [86 13 60 51 95 26 22 29 14 29]]
```



Scatter Plot with Marginal Histograms is basically a joint distribution plot with the marginal distributions of the two variables. In data visualization, we often plot the joint behavior of two random variables (bi-variate distribution) or any number of random variables. But if data is too large, overlapping can be an issue. Hence, to distinguish between variables it is useful to have the probability distribution of each variable on the side along with the joint plot. This individual probability distribution of a random variable is referred to as its marginal probability distribution.

In seaborn, this is facilitated with **jointplot()**. It represents the bi-variate distribution using **scatterplot()** and the marginal distributions using **histplot()**.

```
# importing and creating alias for seaborn

import seaborn as sns

# loading tips dataset

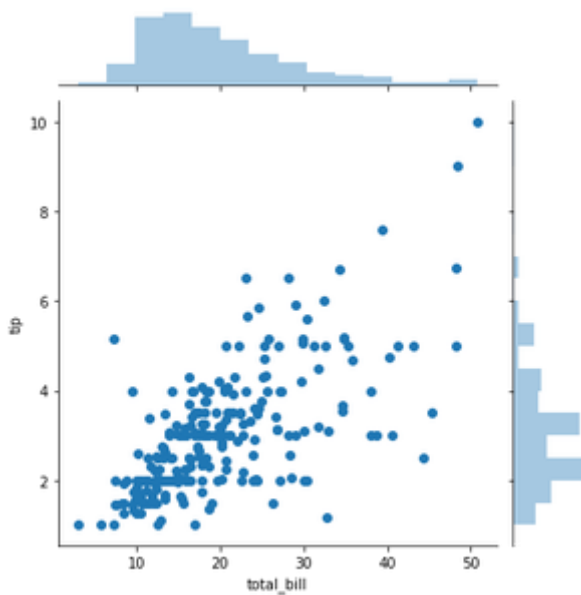
tips = sns.load_dataset("tips")

# plotting scatterplot with histograms for features total bill and tip.

sns.jointplot(data=tips, x="total_bill", y="tip")
```

**Output :**

<seaborn.axisgrid.JointGrid at 0x26203152688>



## TOPIC 8: A SCATTER PLOT MATRIX

In a dataset, for  $k$  set of variables/columns ( $X_1, X_2, \dots, X_k$ ), the scatter plot matrix plot all the pairwise scatter between different variables in the form of a matrix.

Scatter plot matrix answer the following questions:

- Are there any pair-wise relationships between different variables? And if there are relationships, what is the nature of these relationships?
- Are there any outliers in the dataset?
- Is there any clustering by groups present in the dataset on the basis of a particular variable?

For  $k$  variables in the dataset, the scatter plot matrix contains  $k$  rows and  $k$  columns. Each row and column represents as a single scatter plot. Each individual plot  $(i, j)$  can be defined as:

- Vertical Axis: Variable  $X_j$
- Horizontal Axis: Variable  $X_i$

Below are some important factors we consider when plotting the Scatter plot matrix:

- The plot lies on the diagonal is just a 45 line because we are plotting here  $X_i$  vs  $X_i$ . However, we can plot the histogram for the  $X_i$  in the diagonals or just leave it blank.
- Since  $X_i$  vs  $X_j$  is equivalent to  $X_j$  vs  $X_i$  with the axes reversed, we can also omit the plots below the diagonal.
- It can be more helpful if we overlay some line plot on the scattered points in the plots to give more understanding of the plot.
- The idea of the pair-wise plot can also be extended to different other plots such as quantile-quantile plots or bihistogram.

Example:

```
import plotly.express as px
df = px.data.iris()
fig = px.scatter_matrix(df)
fig.show()
```

---

## TOPIC 9: AREA PLOTS

An [area chart](#) is really similar to a [line chart](#), except that the area between the x axis and the line is filled in with color or shading. It represents the evolution of a numeric variable. This section starts by considering matplotlib and seaborn as tools to build area charts. It then shows a few other options for timeseries.

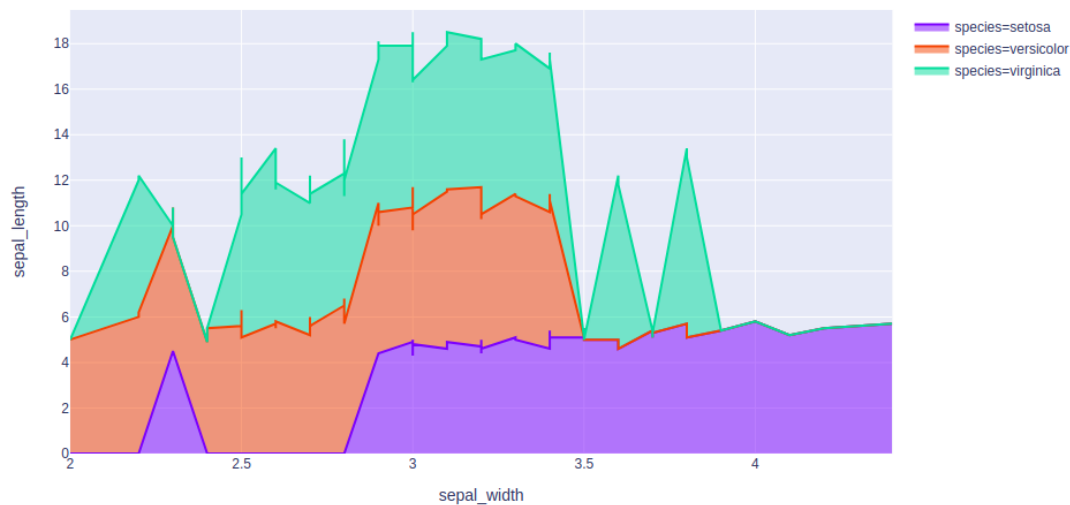
```
import plotly.express as px

df = px.data.iris()

fig = px.area(df, x="sepal_width", y="sepal_length",
              color="species",
              hover_data=['petal_width'],)

fig.show()
```

**Output:**



## TOPIC 10: BUBBLE CHARTS

The bubble chart in Plotly is created using the scatter plot. It can be created using the `scatter()` method of `plotly.express`. A bubble chart is a data visualization which helps to display multiple circles (bubbles) in a two-dimensional plot as same in scatter plot. A bubble chart is primarily used to depict and show relationships between numeric variables.

**Example:**

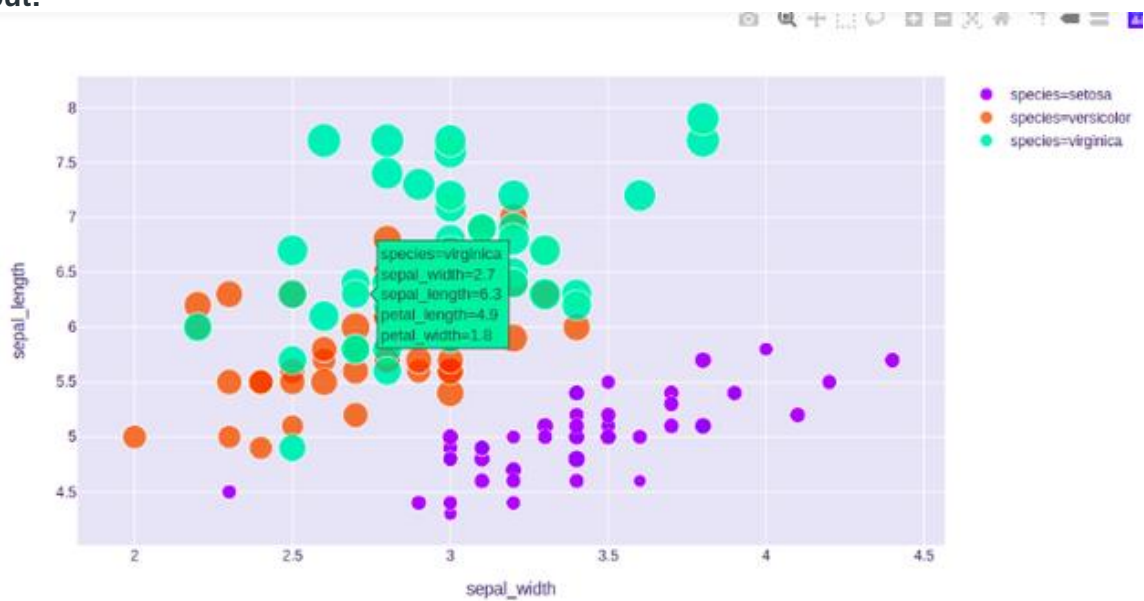
```
import plotly.express as px

df = px.data.iris()

fig = px.scatter(df, x="sepal_width", y="sepal_length",
                color="species",
                size='petal_length',
                hover_data=['petal_width'])
```

```
fig.show()
```

Output:



## Set Marker Size

Marker size and color are used to control the overall size of the marker. Marker size helps to maintain the color inside the bubble in the graph. Scatter is used to actually scale the marker sizes and color based on data.

Example:

```
import plotly.graph_objects as px

import numpy as np

# creating random data through randomint

# function of numpy.random

np.random.seed(42)
```



```
random_x= np.random.randint(1,101,100)
```

```
random_y= np.random.randint(1,101,100)
```

```
plot = px.Figure(data=[px.Scatter(
```

```
    x = random_x,
```

```
    y = random_y,
```

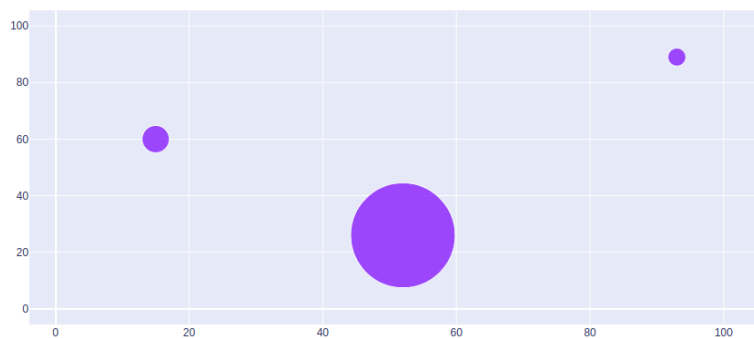
```
    mode = 'markers',
```

```
    marker_size = [115, 20, 30])
```

```
])
```

```
plot.show()
```

**Output:**



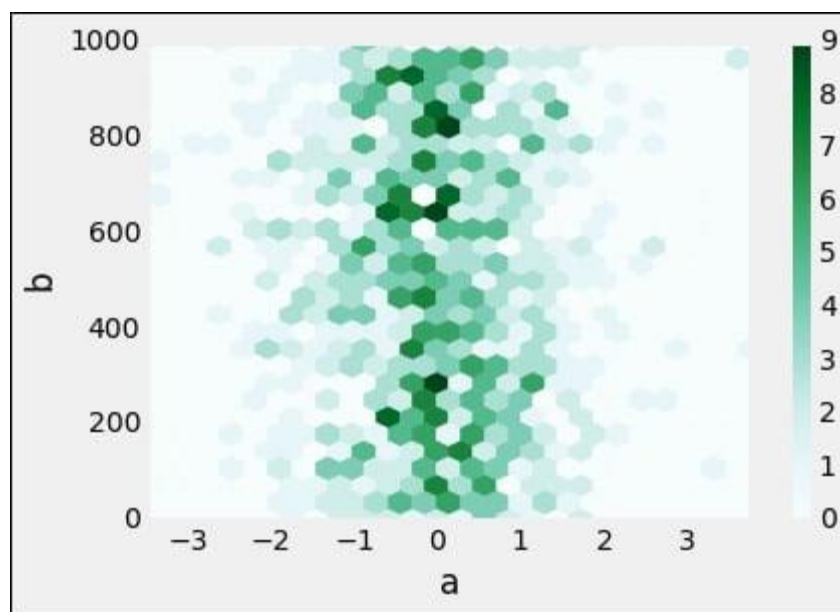
## TOPIC 11: HEXAGON BIN PLOTS

Hexagon bin plot is a 2D histogram plot, in which the bins are hexagons and the color represents the number of data points within each bin. A hexagon bin plot can be created using the `DataFrame.plot()` function and `kind = 'hexbin'`. This kind of plot is really useful if your scatter plot is too dense to interpret. It helps in binning the spatial area of the chart and the intensity of the color that a hexagon can be interpreted as points being more concentrated in this area.

The following code helps in plotting the hexagon bin plot, and the structure of the code is similar to the previously discussed plots:

```
>>> df = pd.DataFrame(np.random.randn(1000, 2), columns=['a', 'b'])  
  
>>> df['b'] = df['b'] + np.arange(1000)  
  
>>> df.plot(kind='hexbin', x='a', y='b', gridsize=25)
```

After the preceding code is executed we'll get the following output:



---

## TOPIC 12: TRELLIS PLOTS

A Trellis plot is a layout of smaller charts in a grid with consistent scales. Each smaller chart represents an item in a category, named conditions. The data displayed on each smaller chart is conditional for the items in the category.

Trellis plots are useful for finding structures and patterns in complex data. The grid layout looks similar to a garden trellis, hence the name Trellis plots.

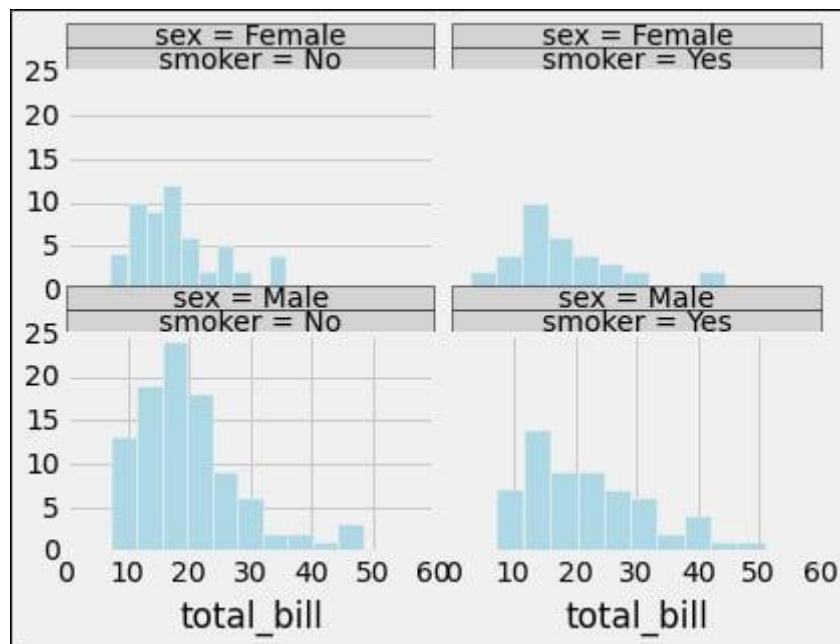
The following code helps in plotting a trellis chart where for each combination of sex and smoker/nonsmoker:

```

>>> tips_data = pd.read_csv('Data/tips.csv')
>>> plt.figure()
>>> plot = rplot.RPlot(tips_data, x='total_bill', y='tip')
>>> plot.add(rplot.TrellisGrid(['sex', 'smoker']))
>>> plot.add(rplot.GeoHistogram())
>>> plot.render(plt.gcf())

```

After the preceding code is executed we'll get the following output:



### TOPIC 13: \*A 3D PLOT OF A SURFACE

A **Surface Plot** is a representation of three-dimensional dataset. It describes a functional relationship between two independent variables X and Z and a designated dependent variable Y, rather than showing the individual data points. It is a companion plot of the contour plot. It is similar to the wireframe plot, but each face of the wireframe is a filled polygon. This helps to create the topology of the surface which is being visualized.

#### Creating 3D surface Plot

The axes3d present in Matplotlib's mpl\_toolkits.mplot3d toolkit provides the necessary functions used to create 3D surface plots. Surface plots are created by using `ax.plot_surface()` function.

#### Syntax:

```
ax.plot_surface(X, Y, Z)
```

where X and Y are 2D array of points of x and y while Z is 2D array of heights. Some more attributes of `ax.plot_surface()` function are listed below:

<b>Attribute</b>	<b>Description</b>
X, Y, Z	2D arrays of data values
cstride	array of column stride(step size)
rstride	array of row stride(step size)
ccount	number of columns to be used, default is 50
rcount	number of row to be used, default is 50
color	color of the surface
cmap	colormap for the surface
norm	instance to normalize values of color map
vmin	minimum value of map
vmax	maximum value of map
facecolors	face color of individual surface
shade	shades the face color

**Example:** Let's create a 3D surface by using the above function

```
# Import libraries

from mpl_toolkits import mplot3d

import numpy as np

import matplotlib.pyplot as plt
```

```
# Creating dataset

x = np.outer(np.linspace(-3, 3, 32), np.ones(32))

y = x.copy().T # transpose

z = (np.sin(x **2) + np.cos(y **2) )

# Creating figure

fig = plt.figure(figsize =(14, 9))

ax = plt.axes(projection ='3d')

# Creating plot

ax.plot_surface(x, y, z)

# show plot

plt.show()
```

**Output:**

