11. Discuss in detail about the Python Variables and Tuples

Python Variables:

Variables in Python are names that represent values. They are used to store data and can be changed as needed. In Python, variables are dynamically typed, which means that the data type of a variable is determined at runtime, based on the type of value assigned to it. This allows for greater flexibility, but also requires more careful programming to avoid errors.

To create a variable in Python, you simply assign a value to a name. For example:

```
x = 5
```

In this example, the name x is assigned the value 5. Once a variable has been assigned a value, you can use the variable in your code to refer to that value. For example:

```
y = x + 10
```

In this example, the variable y is assigned the value of x + 10, which is 15.

Python Tuples:

Tuples in Python are another type of data structure used to store a collection of values. Tuples are similar to lists, but they are immutable, which means that once a tuple is created, its contents cannot be changed. Tuples are useful when you need to store a fixed set of values that won't change, such as coordinates or dates.

To create a tuple in Python, you use parentheses () and separate the values with commas. For example:

```
my_tuple = (1, 2, 3)
```

In this example, my_tuple is a tuple containing the values 1, 2, and 3. Once a tuple has been created, you can access its values using indexing, just like with lists. For example:

```
print(my_tuple[0]) # prints 1

print(my_tuple[1]) # prints 2

print(my_tuple[2]) # prints 3
```

You can also use tuples to return multiple values from a function. For example:

```
def get_name_and_age():

   name = "Alice"

   age = 30

   return (name, age)

result = get_name_and_age()

print(result) # prints ("Alice", 30)
```

In this example, the get_name_and_age() function returns a tuple containing the values name and age. The tuple is then assigned to the result variable, which can be used to access the values returned by the function.

12. Discuss in detail about Dictionaries, Sets and write suitable syntax

Python Dictionaries:

Dictionaries in Python are used to store key-value pairs. Each value in a dictionary is associated with a unique key, which can be used to access the value. Dictionaries are useful when you need to store a large amount of data that can be accessed quickly using a key.

To create a dictionary in Python, you use curly braces {} and separate the keys and values with colons :. For example:

my_dict = {'name': 'Alice', 'age': 30}

In this example, my_dict is a dictionary containing two key-value pairs: 'name': 'Alice' and 'age': 30. You can access the values in a dictionary using the keys. For example:

print(my_dict['name']) # prints 'Alice'

print(my_dict['age']) # prints 30

You can also add, remove, and modify key-value pairs in a dictionary using the syntax:

- Adding a new key-value pair: my_dict['key'] = value
- Modifying an existing key-value pair: my_dict['key'] = new_value
- Removing a key-value pair: del my_dict['key']

Python Sets:

Sets in Python are used to store unique elements. Each element in a set is unique, which means that duplicates are not allowed. Sets are useful when you need to store a collection of unique values and perform operations such as union, intersection, and difference.

To create a set in Python, you use curly braces {} and separate the values with commas. For example:

my_set = {1, 2, 3}

In this example, my_set is a set containing the values 1, 2, and 3. You can perform operations on sets using the built-in set methods. For example:

- Adding an element to a set: my_set.add(element)
- Removing an element from a set: my_set.remove(element)
- Union of two sets: set1.union(set2)
- Intersection of two sets: set1.intersection(set2)
- Difference of two sets: set1.difference(set2)

Syntax for creating a dictionary:

# Creating an empty dictionary

my_dict = {}

# Creating a dictionary with key-value pairs

my_dict = {'key1': 'value1', 'key2': 'value2'}

# Adding a key-value pair to a dictionary

```
my_dict['key3'] = 'value3'

# Modifying a key-value pair in a dictionary

my_dict['key1'] = 'new_value1'

# Removing a key-value pair from a dictionary

del my_dict['key2']
```

Syntax for creating a set:

```
# Creating an empty set

my_set = set()

# Creating a set with values

my_set = {1, 2, 3}

# Adding an element to a set

my_set.add(4)

# Removing an element from a set

my_set.remove(2)

# Union of two sets

set1 = {1, 2, 3}

set2 = {2, 3, 4}

union = set1.union(set2)

# Intersection of two sets

intersection = set1.intersection(set2)

# Difference of two sets

difference = set1.difference(set2)
```

13. Interpret on any TEN built in functions with suitable syntax.

Here are ten built-in functions in Python and their syntax:

len() - This function returns the length of an object (i.e. the number of elements in the object).

Syntax:

```
len(object)
```

Example:

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

print(len(my_list)) # Output: 5
```

print() - This function prints the specified values to the standard output.

Syntax:

print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

Example:

print('Hello, world!') # Output: Hello, world!

type() - This function returns the type of an object.

Syntax:

type(object)

Example:

my_list = [1, 2, 3, 4, 5]

print(type(my_list)) # Output: <class 'list'>

str() - This function converts an object to a string.

Syntax:

str(object)

Example:

my_number = 42

my_string = str(my_number)

print(my_string) # Output: '42'

max() - This function returns the largest element in an iterable.

Syntax:

max(iterable, *iterables, key=None, default=None)

Example :

max() - This function returns the largest element in an iterable.

Syntax:

min() - This function returns the smallest element in an iterable.

Syntax:

min(iterable, *iterables, key=None, default=None)

Example:

my_list = [1, 2, 3, 4, 5]

print(min(my_list)) # Output: 1

abs() - This function returns the absolute value of a number.

Syntax:

```
abs(x)
```

Example:

```
my_number = -42
print(abs(my_number)) # Output: 42
```

sorted() - This function returns a new sorted list from the elements in an iterable.

Syntax:

```
sorted(iterable, key=None, reverse=False)
```

Example:

```
my_list = [5, 3, 1, 4, 2]
print(sorted(my_list)) # Output: [1, 2, 3, 4, 5]
```

sum() - This function returns the sum of all the elements in an iterable.

Syntax:

```
sum(iterable, start=0)
```

Example:

```
my_list = [1, 2, 3, 4, 5]
print(sum(my_list)) # Output: 15
```

range() - This function returns a sequence of numbers, starting from 0 and incrementing by 1 (or by a specified step), up to a specified number.

Syntax:

```
range(stop)
range(start, stop[, step])
```

Example:

```
my_range = range(5)
print(list(my_range)) # Output: [0, 1, 2, 3, 4]
```

15. Illustrate the steps involved in Sort NumPy Array in Descending Order and write suitable coding

NumPy is a Python library used for working with arrays. It provides a powerful N-dimensional array object, as well as functions for working with these arrays. NumPy is designed to be efficient for scientific computation, and is widely used in fields such as data science, machine learning, and engineering

To sort a NumPy array in descending order, you can use the numpy.sort() function with the parameter kind='quicksort' and order=None, and then reverse the sorted array using the [::-1] slicing notation. Here are the steps involved:

1. Import the NumPy library
2. Create a NumPy array
3. Use the numpy.sort() function to sort the array in descending order
4. Reverse the sorted array using slicing

Here is the Python code that implements these steps:

```
import numpy as np

# Create a NumPy array

arr = np.array([3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5])

# Sort the array in descending order

arr_desc = np.sort(arr, kind='quicksort')[::-1]

print(arr_desc)
```

Output:

```
[9 6 5 5 5 4 3 3 2 1 1]
```

In the code above, we first import the NumPy library using the import statement. Then, we create a NumPy array arr with some values. We use the numpy.sort() function to sort the array in descending order, and store the sorted array in arr_desc. Finally, we print the sorted array arr_desc to the console.

16. Write the detail explanation of NumPy arrays and create a program of compare two NumPy arrays

NumPy arrays are a central data structure in the NumPy library, which is used for working with arrays and performing mathematical operations on them. NumPy arrays are similar to Python lists, but they are more powerful and efficient, especially for large arrays.

NumPy arrays can be created using the numpy.array() function, which takes a sequence-like object (e.g. list, tuple) as input and returns a NumPy array. NumPy arrays are homogeneous, meaning that all elements must be of the same data type.

Types of Array:

- One Dimensional Array
- Multi-Dimensional Array

One Dimensional Array:

A one-dimensional array is a type of linear array.

```
# importing numpy module
import numpy as np
# creating list
list = [1, 2, 3, 4]
# creating numpy array
sample_array = np.array(list1)
print("List in python : ", list)
print("Numpy Array in python :",sample_array)
```

Output:

```
List in python :  [1, 2, 3, 4]
Numpy Array in python :  [1 2 3 4]
```

Multi-Dimensional Array:

Data in multidimensional arrays are stored in tabular form.

```
# importing numpy module
import numpy as np
# creating list
list_1 = [1, 2, 3, 4]
list_2 = [5, 6, 7, 8]
list_3 = [9, 10, 11, 12]
# creating numpy array
sample_array = np.array([list_1,list_2,list_3])
print("Numpy multi dimensional array in python\n",sample_array)
```

Output:

```
Numpy multi dimensional array in python
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

example program to compare two NumPy arrays:

```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

# Compare two NumPy arrays element-wise

result = np.equal(arr1, arr2)

print(result)
```

Output:

```
[False False False]
```

18. Explain in detail about Pandas data structures and write suitable example

Pandas is a Python library that provides powerful data structures for working with structured data. The two main data structures in Pandas are Series and DataFrame.

A Series is a one-dimensional array-like object that can hold any data type, such as integers, floats, strings, or even Python objects. Each element in a Series has a label or index that identifies its position within the Series. Series can be created using the pandas.Series() function, which takes a list or array of values as input.

Here is an example of creating a Series in Pandas:

```
import pandas as pd

data = [1, 2, 3, 4, 5]

series = pd.Series(data)

print(series)
```

Output:

```
0    1

1    2

2    3

3    4

4    5

dtype: int64
```

A DataFrame is a two-dimensional table-like data structure that consists of rows and columns. Each column in a DataFrame can have a different data type, such as integers, floats, strings, or Python objects. A DataFrame can be created using the pandas.DataFrame() function, which takes a dictionary, list of lists, or NumPy array as input.

Here is an example of creating a DataFrame in Pandas:

```
import pandas as pd

data = {

   'name': ['Alice', 'Bob', 'Charlie', 'David'],

   'age': [25, 30, 35, 40],

   'country': ['USA', 'Canada', 'UK', 'Australia']

}

df = pd.DataFrame(data)

print(df)
```

Output:

```
      name  age    country

0    Alice   25       USA

1      Bob   30    Canada

2   Charlie   35        UK

3    David   40  Australia
```

In the example above, we created a dictionary data with three keys ('name', 'age', and 'country') and their corresponding values. We then passed this dictionary to the pandas.DataFrame() function to create a DataFrame df. The resulting DataFrame has four rows (one for each person) and three columns (name, age, and country).

Pandas provides many powerful functions and methods for working with Series and DataFrame objects, such as selecting, filtering, grouping, aggregating, and merging data. With these tools, Pandas makes it easy to manipulate and analyze structured data in Python.

19.Write a code for Selecting a single columns and multiple columns using indexing operator

Pandas is a popular data analysis library in Python that provides various functionalities for data manipulation and analysis. One of the essential tasks in data analysis is selecting specific columns from a DataFrame. In this detailed guide, we'll explore various methods of selecting columns in Pandas.

Assume we have a DataFrame named df as follows:

```
import pandas as pd

data = {'name': ['John', 'Alice', 'Bob'],
        'age': [25, 28, 32],
        'gender': ['M', 'F', 'M'],
        'salary': [50000, 60000, 70000]}
df = pd.DataFrame(data)
```

Selecting a single column

To select a single column from a DataFrame, we can use either of the following methods:

- Using bracket notation:
  - df['name']
- Using dot notation:
  - df.name

Both of these methods return a Pandas Series object containing the values of the selected column.

Selecting multiple columns

To select multiple columns, we can pass a list of column names to the bracket notation:

df[['name', 'age']]

This returns a new DataFrame with the selected columns.

- Selecting columns by position

We can also select columns by their position in the DataFrame using the iloc method. For example, to select the first two columns, we can do:

df.iloc[:, :2]

This returns a new DataFrame containing the first two columns of the original DataFrame. The : operator specifies that we want to select all rows, and :2 specifies that we want to select the first two columns.

- Selecting columns by condition

We can select columns based on a condition using boolean indexing. For example, to select columns where the age is greater than 26, we can do:

df.loc[:, df.mean() > 26]

This returns a new DataFrame with the columns where the mean value of the column is greater than 26.

Dropping columns

Finally, we can drop columns from a DataFrame using the drop method. For example, to drop the gender column, we can do:

df.drop('gender', axis=1)

This returns a new DataFrame with the gender column removed. The axis=1 argument specifies that we want to drop a column (as opposed to a row, which would be axis=0).

These are some of the common methods to select columns in Pandas. By using these methods, we can manipulate the DataFrame to extract the information we need for analysis and visualization.