Academic Year:2023-24 SAP ID:

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: DJS22ITL405 **DATE: 8 February 2024**

COURSE NAME: Programing Laboratory 2 (Python) **CLASS:** S.Y. B.Tech

NAME: ANISH SHARMA **SAP**: 60003220045

EXPERIMENT NO. 2

CO/LO: CO1, CO2.

AIM / OBJECTIVE:

Write a Python program to implement functions of List, Tuples, Dictionaries, Arrays / Numpy Array (1D, 2D) applications.

DESCRIPTION OF EXPERIMENT:

Python Data Structures: Lists, Dictionaries, Sets, Tuples

What Is a Data Structure?

A data structure is a way of organizing data in computer memory, implemented in a programming language. This organization is required for efficient storage, retrieval, and modification of data. It is a fundamental concept as data structures are one of the main building blocks of any modern software. Learning what data structures exist and how to use them efficiently in different situations is one of the first steps toward learning any programming language.

Data Structures in Python

- Built-in data structures in Python can be divided into two broad categories: **mutable** and **immutable**.
- **Mutable** (from Latin *mutabilis*, "changeable") data structures are those which we can modify — for example, by adding, removing, or changing their elements.
- Python has three mutable data structures: **lists**, **dictionaries**, and **sets**.
- **Immutable data structures**, on the other hand, are those that we cannot modify after their creation. The only basic built-in immutable data structure in Python is a **tuple**.
- Python also has some advanced data structures, such as stacks or queues, which can be implemented with basic data structures. However, these are rarely used in data science and are more common in the field of software engineering and implementation of complex algorithms.

Lists

- Lists in Python are implemented as **dynamic mutable arrays** which hold an **ordered** collection of items.
- First, in many programming languages, arrays are data structures that contain a collection of elements of the same data types (for instance, all elements are integers). However, in Python**, lists can contain heterogeneous data types and objects**.
- For instance, **integers, strings, and even functions can be stored within the same list**.
- Different **elements of a list can be accessed by integer indices** where the first element of a list has the index of 0.

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

- This property derives from the fact that in Python, **lists are ordered**, which means they retain the order in which you insert the elements into the list.
- Next, **we can arbitrarily add, remove, and change elements in the list**. For instance, the .append() method adds a new element to a list, and the .remove() method removes an element from a list. Furthermore, by accessing a list's element by index, we can change it to another element.
- Finally, when creating a list, **we do not have to specify in advance the number of elements it will contain**; therefore, it can be expanded as we wish, making it dynamic.
- Lists are useful when we want to store a collection of different data types and subsequently add, remove, or perform operations on each element of the list (by looping through them).
- Furthermore, lists are useful to store other data structures (and even other lists) by creating, for instance, lists of dictionaries, tuples, or lists.
- It is very common to store a table as a list of lists (where each inner list represents a table's column) for subsequent data analysis.

Thus, the **pros of lists** are:

- They represent the easiest way to store a collection of related objects.
- They are easy to modify by removing, adding, and changing elements.
- They are useful for creating nested data structures, such as a list of lists/dictionaries.

However, they also have **cons**:

- They can be pretty slow when performing arithmetic operations on their elements. (For speed, use NumPy's arrays.)
- They use more disk space because of their under-the-hood implementation.

Examples

Finally, let's have a look at a few examples.

We can create a list using either square brackets ([]) with zero or more elements between them, separated by commas, or the constructor list(). The latter can also be used to transform certain other data structures into lists.

```
# Create an empty list using square brackets
l1 = []

# Create a four-element list using square brackets
l2 = [1, 2, "3", 4]  # Note that this lists contains two different data types: integers and strings

# Create an empty list using the list() constructor
l3 = list()

# Create a three-element list from a tuple using the list() constructor
# We will talk about tuples later in the tutorial
l4 = list((1, 2, 3))

# Print out lists
print(f"List l1: {l1}") //formatted string literals
```

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

print(f"List l2: {l2}")
print(f"List l3: {l3}")
print(f"List l4: {l4}")
List l1: []
List l2: [1, 2, '3', 4]
List l3: []
List l4: [1, 2, 3]
We can access a list's elements using indices, where the first element of a list has the index of 0:
# Print out the first element of list l2
print(f"The first element of the list l2 is {l2[0]}.")

# Print out the third element of list l4
print(f"The third element of the list l4 is {l4[2]}.")

O/P:
The first element of the list l2 is 1.

The third element of the list l4 is 3.
We can also **slice** lists and access multiple elements simultaneously:
# Assign the third and the fourth elements of l2 to a new list
l5 = l2[2:]

Print out the resulting list
print(l5)['3', 4]
Note that we did not have to specify the index of the last element we wanted to access if we wanted all elements from index 2 (included) to the end of the list. Generally speaking, the list slicing works as follows:

1. Open square brackets.
2. Write the first index of the first element we want to access. This element will be included in the output. Place the colon after this index.
3. Write the index, plus one of the last elements we want to access. The addition of 1 here is required because the element under the index we write **will not be included in the output**.

Let's show this behavior with an example:
print(f"List l2: {l2}")

Access the second and the third elements of list l2 (these are the indices 1 and 2)
print(f"Second and third elements of list l2: {l2[1:3]}")List l2: [1, 2, '3', 4]
Second and third elements of list l2: [2, '3']
Note that the last index we specified is 3, not 2, even though we wanted to access the element under index 2. Thus, the last index we write is not included.
You can experiment with different indices and bigger lists to understand how indexing works.

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

Now let's demonstrate that lists are mutable. For example, we can append() a new element to a list or remove() a specific element from it:

```
# Append a new element to the list l1
l1.append(5)

# Print the modified list
print("Appended 5 to the list l1:")
print(l1)

# Remove element 5 from the list l1
l1.remove(5)

# Print the modified list
print("Removed element 5 from the list l1:")
print(l1)
```

O/P:

```
Appended 5 to list l1:[5]
Removed element 5 from the list l1:[]
```

Additionally, we can modify the elements that are already in the list by accessing the required index and assigning a new value to that index:

```
# Print the original list l2
print("Original l2:")
print(l2)

# Change value at index 2 (third element) in l2
l2[2] = 5

# Print the modified list l2
print("Modified l2:")
print(l2)
```

O/P:

```
Original l2:
[1, 2, '3', 4]
Modified l2:
[1, 2, 5, 4]
```

Dictionaries

- Dictionaries in Python are very similar to real-world dictionaries.
- These are **mutable** data structures that contain a collection of **keys** and, associated with them, **values**.
- This structure makes them very similar to word-definition dictionaries. For example, the word *dictionary* (our key) is associated with its definition (value) in Oxford online

dictionary: *a book or electronic resource that gives a list of the words of a language in alphabetical order and explains what they mean, or gives a word for them in a foreign language*.

- Dictionaries are used to quickly access certain data associated with a **unique** key.
- Uniqueness is essential, as we need to access only certain pieces of information and not confuse it with other entries.
- Imagine we want to read the definition of *Data Science*, but a dictionary redirects us to two different pages: which one is the correct one? Note that technically we can create a dictionary with two or more identical keys, although due to the nature of dictionaries, it is not advisable.

```
# Create dictionary with duplicate keys
d1 = {"1": 1, "1": 2}
print(d1)

# It will only print one key, although no error was thrown
# If we  try to access this key, then it'll return 2, so the value of the second key
print(d1["1"])

# It is technically possible to create a dictionary, although this dictionary will not support them,
# and will contain only one of the key
```

- We use dictionaries when we are able to associate (in technical terms, **to map**) a unique key to certain data, and we want to access that data **very quickly** (in constant time, no matter the dictionary size).
- Moreover, dictionary values can be pretty complex. For example, our keys can be customer names, and their personal data (values) can be dictionaries with the keys like "Age," "Hometown," etc.

Thus, the **pros of dictionaries** are:

- They make code much easier to read if we need to generate key:value pairs. We can also do the same with a list of lists (where inner lists are pairs of "keys" and "values"), but this looks more complex and confusing.
- We can look up a certain value in a dictionary very quickly. Instead, with a list, we would have to read the list before we hit the required element. This difference grows drastically if we increase the number of elements.

However, **their cons** are:

- **They occupy a lot of space.** If we need to handle a large amount of data, this is not the most suitable data structure.
- In Python 3.6.0 and later versions, dictionaries remember the order of element insertions. Keep that in mind to avoid compatibility issues when using the same code in different versions of Python.

Examples

Let's now take a look at a few examples. First, we can create a dictionary with curly brackets ({}) or the dict() constructor:

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

```
# Create an empty dictionary using curly brackets
d1 = {}

# Create a two-element dictionary using curly brackets
d2 = {"John": {"Age": 27, "Hometown": "Boston"}, "Rebecca": {"Age": 31, "Hometown": "Chicago"}}
# Note that the above dictionary has a more complex structure as its values are dictionaries themselves!

# Create an empty dictionary using the dict() constructor
d3 = dict()

# Create a two-element dictionary using the dict() constructor
d4 = dict([["one", 1], ["two", 2]])  # Note that we created the dictionary from a list of lists

# Print out dictionaries
print(f"Dictionary d1: {d1}")
print(f"Dictionary d2: {d2}")
print(f"Dictionary d3: {d3}")
print(f"Dictionary d4: {d4}")
Dictionary d1: {}
Dictionary d2: {'John': {'Age': 27, 'Hometown': 'Boston'}, 'Rebecca': {'Age': 31, 'Hometown': 'Chicago'}}
Dictionary d3: {}
Dictionary d4: {'one': 1, 'two': 2}
```

Now let's access an element in a dictionary. We can do this with the same method as lists:

```
# Access the value associated with the key 'John'
print("John's personal data is:")
print(d2["John"])John's personal data is:
{'Age': 27, 'Hometown': 'Boston'}
```

Next, we can also modify dictionaries — for example, by adding new key:value pairs:

```
# Add another name to the dictionary d2
d2["Violet"] = {"Age": 34, "Hometown": "Los Angeles"}

# Print out the modified dictionary
print(d2){'John': {'Age': 27, 'Hometown': 'Boston'}, 'Rebecca': {'Age': 31, 'Hometown': 'Chicago'}, 'Violet': {'Age': 34, 'Hometown': 'Los Angeles'}}
```

As we can see, a new key, "Violet", has been added.

Sets

- Sets in Python can be defined as mutable dynamic collections of **immutable unique** elements.

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

- The **elements contained in a set must be immutable**. Sets may seem very similar to lists, but in reality, they are very different.
- First, they may **only contain unique elements**, so no duplicates are allowed. Thus, sets can be used to remove duplicates from a list. Next, like sets in mathematics, they have unique operations which can be applied to them, such as set union, intersection, etc. Finally, they are very efficient in checking whether a specific element is contained in a set.

Thus, the pros of sets are:

- We can perform unique (but similar) operations on them.
- They are significantly faster than lists if we want to check whether a certain element is contained in a set.

But their cons are:

- Sets are intrinsically unordered. If we care about keeping the insertion order, they are not our best choice.
- We cannot change set elements by indexing as we can with lists.

Examples

To create a set, we can use either curly brackets ({}) or the set() constructor. Do not confuse sets with dictionaries (which also use curly brackets), as sets do not contain key:value pairs. Note, though, that like with dictionary keys, only immutable data structures or types are allowed as set elements. This time, let's directly create populated sets:

```
# Create a set using curly brackets
s1 = {1, 2, 3}

# Create a set using the set() constructor
s2 = set([1, 2, 3, 4])

# Print out sets
print(f"Set s1: {s1}")
print(f"Set s2: {s2}")Set s1: {1, 2, 3}
Set s2: {1, 2, 3, 4}
```

In the second example, we used an **iterable** (such as a list) to create a set.

However, if we used lists as set elements, Python would throw an error. Why do you think it happens? **Tip**: read the definition of sets.

To practice, you can try using other data structures to create a set.

As with their math counterparts, we can perform certain operations on our sets. For example, we can create a **union** of sets, which basically means merging two sets together. However, if two sets have two or more identical values, the resulting set will contain only one of these values. There are two ways to create a union: either with the union() method or with the vertical bar (|) operator. Let's make an example:

```
# Create two new sets
names1 = set(["Glory", "Tony", "Joel", "Dennis"])
names2 = set(["Morgan", "Joel", "Tony", "Emmanuel", "Diego"])

# Create a union of two sets using the union() method
```

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

```
names_union = names1.union(names2)

# Create a union of two sets using the | operator
names_union = names1 | names2

# Print out the resulting union
print(names_union){'Glory', 'Dennis', 'Diego', 'Joel', 'Emmanuel', 'Tony', 'Morgan'}
```

In the above union, we can see that Tony and Joel appear only once, even though we merged two sets.
Next, we may also want to find out which names appear in both sets. This can be done with the intersection() method or the ampersand (&) operator.

```
# Intersection of two sets using the intersection() method
names_intersection = names1.intersection(names2)

# Intersection of two sets using the & operator
names_intersection = names1 & names2

# Print out the resulting intersection
print(names_intersection){'Joel', 'Tony'}
```

Joel and Tony appear in both sets; thus, they are returned by the set intersection.
The last example of set operations is the difference between two sets. In other words, this operation will return all the elements that are present in the first set, but not in the second one.

We can use either the difference() method or the minus sign (-):

```
# Create a set of all the names present in names1 but absent in names2 with the difference() method
names_difference = names1.difference(names2)

# Create a set of all the names present in names1 but absent in names2 with the - operator
names_difference = names1 - names2

# Print out the resulting difference
print(names_difference){'Dennis', 'Glory'}
```

Tuples

- Tuples are almost identical to lists, so they contain an ordered collection of elements, except for one property: they are **immutable**.
- We would use tuples if we needed a data structure that, once created, cannot be modified anymore.
- Furthermore, tuples can be used as dictionary keys if all the elements are immutable.
- Other than that, tuples have the same properties as lists. To create a tuple, we can either use round brackets (()) or the tuple() constructor. We can easily transform lists into tuples and vice versa (recall that we created the list l4 from a tuple).

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

The **pros of tuples** are:

- **They are immutable**, so once created, we can be sure that we won't change their contents by mistake.
- **They can be used as dictionary keys** if all their elements are immutable.

The **cons of tuples** are:

- We cannot use them when we have to work with modifiable objects; we have to resort to lists instead.
- Tuples cannot be copied.
- **They occupy more memory than lists**.

Examples

Let's take a look at some examples:

```
# Create a tuple using round brackets
t1 = (1, 2, 3, 4)

# Create a tuple from a list the tuple() constructor
t2 = tuple([1, 2, 3, 4, 5])

# Create a tuple using the tuple() constructor
t3 = tuple([1, 2, 3, 4, 5, 6])

# Print out tuples
print(f"Tuple t1: {t1}")
print(f"Tuple t2: {t2}")
print(f"Tuple t3: {t3}")Tuple t1: (1, 2, 3, 4)
Tuple t2: (1, 2, 3, 4, 5)
Tuple t3: (1, 2, 3, 4, 5, 6)
```

Is it possible to create tuples from other data structures (i.e., sets or dictionaries)? Try it for practice.

Tuples are immutable; thus, we cannot change their elements once they are created. Let's see what happens if we try to do so:

```
# Try to change the value at index 0 in tuple t1
t1[0] = 1Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

It is a TypeError! Tuples do not support item assignments because they are immutable. To solve this problem, we can convert this tuple into a list.

However, we can access elements in a tuple by their indices, like in lists:

```
# Print out the value at index 1 in the tuple t2
print(f"The value at index 1 in t2 is {t2[1]}.")The value at index 1 in t2 is 2.
```

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

Tuples can also be used as dictionary keys. For example, we may store certain elements and their consecutive indices in a tuple and assign values to them:

# Use tuples as dictionary keys
working_hours = {("Rebecca", 1): 38, ("Thomas", 2): 40}
If you use a tuple as a dictionary key, then the tuple must contain immutable objects:

# Use tuples containing mutable objects as dictionary keys
working_hours = {(["Rebecca", 1]): 38, (["Thomas", 2]): 40}----------
---

TypeError Traceback (most recent call last)

Input In [20], in ()
1 # Use tuples containing mutable objects as dictionary keys
----> 2 working_hours = {(["Rebecca", 1]): 38, (["Thomas", 2]): 40}

TypeError: unhashable type: 'list'
We get a TypeError if our tuples/keys contain mutable objects (lists in this case).

Arrays / Numpy Array (1D, 2D)

Numpy:

•NumPy, which stands for Numerical Python, is a library consisting of multidimensional array objects and a collection of routines for processing those arrays.
•Using NumPy, mathematical and logical operations on arrays can be performed on Array
•An array is a data structure that stores values of same data type
•List can contain values corresponding to different data types
•Arrays in python can only contain values corresponding to the same data type.
NumPy Array:
•A NumPy array is a grid of values, all of the same type , and is indexed by a tuple of nonnegative integers
•The number of dimensions is the rank of the array
•The shape of an array is a tuple of integers giving the size of the array along each dimension.

Creating an Array:
import numpy as np
arr = np.array([1, 2, 3, 4, 5]) print(arr)
Output:
[1 2 3 4 5]

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

We can also pass a tuple in the array function to create an array.
import numpy as np
arr = np.array((1, 2, 3, 4, 5)) print(arr)
The output would be similar to the above case.

Dimensions- Arrays:
To identify the dimensions of the array, we can use ndim
0-D Arrays:
The following code will create a zero-dimensional array with a value 36. import numpy as np
arr = np.array(36)
print(arr)
Output:
36

l-Dimensional Array:
The array that has Zero Dimensional arrays as its elements is a uni-dimensional or I-D array. The code below creates a 1-D array,
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
Output:
[1 2 3 4 5]
Two Dimensional Arrays:
2-D Arrays are the ones that have I-D arrays as its element. The following code will create a 2-D array with 1,2,3 and 4,5,6 as its values.
import numpy as np
arrl = np.array([[1, 2, 3], [4, 5, 6]])
print(arrl)
Output:
[[1 2 3]
[4 5 6]]
Three Dimensional Arrays:
Let us see an example of creating a 3-D array with two 2-D arrays:
import numpy as np
arrl = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(arr1)
Output:
[[[1 2 3]
[4 5 6]]
[[1 2 3]
[4 5 6]]]

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

Operations using NumPy

Using NumPy, a developer can perform the following operations —
•Mathematical and logical operations on arrays.
•Fourier transforms and routines for shape manipulation.
•Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.
The basic ndarray is created using an array function in NumPy as follows-
numpy.array
It creates a ndarray from any object exposing an array interface, or from any method that returns an array.

numpy.array(object, dtype = None, copy = True, order = None, subok = False, ndmin = 0)

Sr.No.	Parameter & Description
1	**object** Any object exposing the array interface method returns an array, or any (nested) sequence.
2	**dtype** Desired data type of array, optional
3	**copy** Optional. By default (true), the object is copied
4	**order** C (row major) or F (column major) or A (any) (default)
5	**subok** By default, returned array forced to be a base class array. If true, sub-classes passed through
6	**ndmin** Specifies minimum dimensions of resultant array

The ndarray object consists of a contiguous one-dimensional segment of computer memory, combined with an indexing scheme that maps each item to a location in the memory block.

Example 1

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

The output is as follows — [1, 2, 3]
Example 2

```
# more than one dimensions import numpy as np
a = np.array([[1, 2], [3, 4]])
```

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
DEPARTMENT OF INFORMATION TECHNOLOGY

print (a)
The output is as follows —
[[1, 2]
[3, 4]]

Example 3
# minimum dimensions import numpy as np
a = np.array([1, 2, 3,4,5], ndmin = 2)
print (a)
The output is as follows — [[1, 2, 3, 4, 5]]
Example 4

# dtype parameter import numpy as np
a = np.array([1, 2, 3], dtype= complex)
print (a)
The output is as follows —
[1.+0.j, 2.+0.j, 3.+0.j]

#Create Random Array
g = np.random.random((3,3))

 # Indexing
import numpy as np
arr = np.array([1, 2, 3, 4]) print(arr[0])

#Data Types in Array
The NumPy array object has a property called dtype that returns the datatype of the array

#S1icing
We pass slice instead of index like this: *[start:end.*
We can also define the step, like this: *[start:end:step]*.

#Copy
The copy *owns* the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.
#Interating

As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python

QUESTIONS:

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

1. Write a Python program to select an item randomly from a list.
2. Write a Python program to count the elements in a list until an element is a tuple
3. Create a dictionary of 5 countries with their currency details and display them.
4. Write a Python program to reverse a tuple.
5. Create a Numpy array filled with all ones
6. Check whether a Numpy array contains a specified row
7. Compute mathematical operations on Array, Add & Multiply two matrices
8. Find the most frequent value in a NumPy array
9. Flatten a 2d numpy array into ld array
10. Calculate the sum of all columns in a 2D NumPy array
11. Calculate the average, variance and standard deviation in Python using NumPy
12. Insert a space between characters of all the elements of a given NumPy array?
13. Sort the values in a matrix

QUESTIONS FOR WRITE-UP:

1. **Explain the difference between a List, Tuple and Set.**
2. **Explain the array functions used in the experiment with syntax.**

OBSERVATIONS / DISCUSSION OF RESULT:

This section should interpret the outcome of the experiment. The observations can be visually represented using images, tables, graphs, etc. This section should answer the question "What do the result tell us?" Compare and interpret your results with expected behavior. Explain unexpected behavior, if any.

CONCLUSION:

Base all conclusions on your actual results; describe the meaning of the experiment and the implications of your results.

REFERENCES:

Website References:

1. https://www.w3schools.com/python
2. https://www.tutorialspoint.com/
3. https://www.programiz.com/

1. Write a Python program to select an item randomly from a list.
2. Write a Python program to count the elements in a list until an element is a tuple
3. Create a dictionary of 5 countries with their currency details and display them.
4. Write a Python program to reverse a tuple.
5. Create a Numpy array filled with all ones
6. Check whether a Numpy array contains a specified row
7. Compute mathematical operations on Array, Add & Multiply two matrices
8. Find the most frequent value in a NumPy array
9. Flatten a 2d numpy array into ld array
10. Calculate the sum of all columns in a 2D NumPy array
11. Calculate the average, variance and standard deviation in Python using NumPy
12. Insert a space between characters of all the elements of a given NumPy array?
13. Sort the values in a matrix

ANISH SHARMA

I011

60003220045

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

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

```
import random as r
```

```
r.choice(lst)
```

```
5
```

```
dict={
    "India":"Rupees",
    "USA":"Dollar",
    "Afghanistan":"Afghani",
    "Andorra":"Euro",
    "Albania":"Dinar"
}
dict
```

```
{'India': 'Rupees',
 'USA': 'Dollar',
 'Afghanistan': 'Afghani',
 'Andorra': 'Euro',
 'Albania': 'Dinar'}
```

```
t = (1,2,3,4,5)
t[::-1]
```

```
(5, 4, 3, 2, 1)
```

```
import numpy as np

arr = np.ones(5,int)

arr
```

```
array([1, 1, 1, 1, 1])
```

```
m1 = np.array(
    [
        [1,2,3],
        [4,5,6],
        [7,8,9]
    ]
)
m2= np.array(
    [
        [1,2,3],
        [4,5,6],
        [7,8,9]
    ]
)

m1+m2
```

```
array([[ 2,  4,  6],
       [ 8, 10, 12],
       [14, 16, 18]])
```

```
m1*m2
```

```
array([[ 1,  4,  9],
       [16, 25, 36],
       [49, 64, 81]])
```

```
m1.flatten()
```

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

```
m1.sum()
```

```
45
```

```
arr = np.array([1,2,3,4,5],int)
np.average(arr)
```

```
3.0
```

```
np.std(arr)
```

```
1.4142135623730951
```

```
np.var(arr)
```

```
2.0
```

```
x = np.array(["geeks", "for", "geeks"])

r = np.char.join(" ", x)
r
```

```
array(['g e e k s', 'f o r', 'g e e k s'], dtype='<U9')
```

```
np.sort(m2)
```

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

```
arr = np.array([[1, 2, 3, 4, 5],
                [6, 7, 8, 9, 10],
                [11, 12, 13, 14, 15],
                [16, 17, 18, 19, 20]
                ])

print([1, 2, 3, 4, 5] in arr.tolist())
```

```
True
```

```
print([1, 2,5 , 3, 4, 5] in arr.tolist())
```

```
False
```

```
x = np.array([1,2,3,4,5,1,2,1,1,1])
print(np.bincount(x).argmax())
```

```
1
```

```
a = [[1,2,3],[4,5,6],tuple(7,8,9)]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-61-02d7a42561ab> in <module>
----> 1 a = [[1,2,3],[4,5,6],tuple(7,8,9)]

TypeError: tuple expected at most 1 argument, got 3
```

```
for i in a:
    if a is not tuple:
        print(i)
```

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

```
def Count(li):

    for num in li:
        if isinstance(num, tuple):
            break

        print(num)

li = [4, 5, 6, 10, (1, 2, 3), 11, 2, 4]
```

```
4
5
6
10
4
```