List arguments

```
t1 = [1, 2]
t2 = t1.append(3)
print(t1)
#[1, 2, 3]
print(t2)
#None
t3 = t1 + [3]
print(t3)
#[1, 2, 3]
t2 is t3
#False
```

Activity - 14

- **Write a function called chop that takes a list and modifies it, removing the first and last elements, and returns None**

- **Then write a function called middle that takes a list and returns a new list that contains all but the first and last elements**

Tuples

- A tuple in Python is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas we can change the elements of a list.

Creating a Tuple

- A tuple is created by placing all the items (elements) inside parentheses (), separated by commas. The parentheses are optional, however, it is a good practice to use them.

- A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).

```python
# Different types of tuples

# Empty tuple
my_tuple = ()
print(my_tuple)

# Tuple having integers
my_tuple = (1, 2, 3)
print(my_tuple)

# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)
print(my_tuple)

# nested tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)
```

create tuples without using parentheses

```
my_tuple = 1, 2, 3
my_tuple = 1, "Hello", 3.4
```

create tuples with one Element

```
var1 = ("Hello") # string
var2 = ("Hello",) # tuple
```

Use type() function

```
var1 = ("hello")
print(type(var1))  # <class 'str'>

# Creating a tuple having one element
var2 = ("hello",)
print(type(var2))  # <class 'tuple'>

# Parentheses is optional
var3 = "hello",
print(type(var3))  # <class 'tuple'>
```

Access Tuple Elements

- We can use the index operator [] to access an item in a tuple, where the index starts from 0.
- So, a tuple having n elements will have indices from 0 to n-1.

```
# accessing tuple elements using indexing
letters = ("p", "r", "o", "g", "r", "a", "m", "i", "z")

print(letters[0])   # prints "p"
print(letters[5])   # prints "a"
```

Tuple Negative Indexing

- Python allows negative indexing for its sequences.
- The index of -1 refers to the last item, -2 to the second last item and so on. For example,

```
# accessing tuple elements using negative indexing
letters = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

print(letters[-1])   # prints 'z'
print(letters[-3])   # prints 'm'
```

Tuple Slicing

- We can access a range of items in a tuple by using the slicing operator colon :.

```
# accessing tuple elements using slicing
my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

# elements 2nd to 4th index
print(my_tuple[1:4])  #  prints ('r', 'o', 'g')

# elements beginning to 2nd
print(my_tuple[:-7]) # prints ('p', 'r')

# elements 8th to end
print(my_tuple[7:]) # prints ('i', 'z')

# elements beginning to end
print(my_tuple[:]) # Prints ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

Tuple Methods

- In Python ,methods that add items or remove items are not available with tuple. Only the following two methods are available.

```
my_tuple = ('a', 'p', 'p', 'l', 'e',)

print(my_tuple.count('p'))  # prints 2
print(my_tuple.index('l'))  # prints 3
```

Iterating through a Tuple

- We can use the for loop to iterate over the elements of a tuple. For example,

```
languages = ('Python', 'Swift', 'C++')

# iterating through the tuple
for language in languages:
    print(language)
```

Check if an Item Exists in the Python Tuple

- We use the in keyword to check if an item exists in the tuple or not. For example,

```
languages = ('Python', 'Swift', 'C++')

print('C' in languages)    # False
print('Python' in languages)    # True
```

Objects and values

- If we execute these assignment statements:
  - a = 'banana'
  - b = 'banana‘
- we know that a and b both refer to a string, but we don’t know whether they refer to the *same string. There are two possible states:*

Objects and values

- To check whether two variables refer to the same object, you can use the is operator

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

Objects and values

- But when you create two lists, you get two objects:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

Aliasing

- If a refers to an object and you assign b = a, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

Aliasing

- An object with more than one reference has more than one name, so we say that the object is aliased

- If the aliased object is mutable, changes made with one alias affect the other

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

Dictionaries

- Dictionaries are a collection of key-value pairs.
- Keys are unique identifiers, and values can be any data type.
- Dictionaries are similar to real-world dictionaries that store words and their definitions.

```
# Creating a dictionary
my_dict =  'key1'  'value1'  'key2'  'value2'
'key3'  'value3'

# Accessing dictionary values
print my_dict 'key1'
# Output: value1

# Modifying dictionary values
my_dict 'key2'  = 'new_value'
print my_dict
# Output: {'key1': 'value1', 'key2': 'new_value',
'key3': 'value3'}
```

Creating and Manipulating Dictionaries

- Dictionaries can be created using curly braces {} or the dict() constructor.
- Values can be accessed and modified using keys.
- Dictionary elements can be added, removed, and updated.

```
# Creating a dictionary using curly braces
my_dict =  'name'  'John'  'age'  25  'city'  'New
York'

# Creating a dictionary using dict() constructor
another_dict = dict name='Alice'  age=30
city='London'

# Adding a new key-value pair
my_dict 'occupation'  = 'Engineer'

# Removing a key-value pair
del my_dict 'age'
```

Dictionary Operations

- len() function returns the number of key-value pairs in a dictionary.
- in operator checks if a key exists in the dictionary.
- del keyword removes a key-value pair from the dictionary.

```
my_dict =  'name'  'John'  'age'  25  'city'
'New York'

# Getting the number of key-value pairs
print len my_dict     # Output: 3

# Checking if a key exists
print 'name' in my_dict    # Output: True

# Removing a key-value pair
del my_dict 'age'
```

Dictionary Methods

- keys() method returns a view of all keys in the dictionary.
- values() method returns a view of all values in the dictionary.
- items() method returns a view of all key-value pairs in the dictionary.
- get() method retrieves the value associated with a key.
- update() method updates the dictionary with another dictionary or key-value pairs.

```
my_dict =  'name'  'John'  'age'  25  'city'  'New York'

# Getting all keys
keys = my_dict keys

# Getting all values
values = my_dict values

# Getting all key-value pairs
items = my_dict items

# Retrieving the value associated with a key
value = my_dict get 'age'

# Updating the dictionary
my_dict update  'occupation'  'Engineer'
```

Dictionary Loops

- for loop can be used to iterate over the keys or values of a dictionary.
- items() method can be used to iterate over both keys and values simultaneously.

```
my_dict =  'name'  'John'  'age'  25  'city'  'New York'

# Iterating over keys
for key in my_dict
    print key

# Iterating over values
for value in my_dict values
    print value

# Iterating over key-value pairs
for key  value in my_dict items
    print key  value
```

Dictionary and Lists

- Dictionaries can contain lists as values and vice versa.
- Lists can be stored as values against a key in a dictionary.
- Lists can be used to store multiple values for a key.

```
my_dict =  'fruits'   'apple'  'banana'
'orange'   'colors'   'red'  'green'  'blue'

# Accessing a list within a dictionary
fruits = my_dict 'fruits'
print fruits    # Output: ['apple', 'banana',
'orange']

# Storing a list as a value
my_dict 'numbers'  =  1  2  3
```

Sorting Dictionaries

- Dictionaries are inherently unordered, but they can be sorted based on keys or values using the sorted() function.

```
my_dict =  'name'  'John'  'age'  25  'city'  'New
York'

# Sorting by keys
sorted_dict_keys = sorted my_dict

# Sorting by values
sorted_dict_values = sorted my_dict items
key=lambda x  x 1
```

Activity: 14.1

Question 1: Create a dictionary called "student" with the following key-value pairs:

"name" with the value "John"
"age" with the value 20
"major" with the value "Computer Science"

Question 2: Add a new key-value pair to the "student" dictionary:

"gpa" with the value 3.8

Question 3: Retrieve the value of the "major" key from the "student" dictionary and store it in a variable called "student_major". Print the value.

Cont.

Question 4: Create a dictionary called "grades" with the following key-value pairs:

"math" with the value 90
"science" with the value 85
"history" with the value 95

Question 5: Add the "grades" dictionary as a value to the "student" dictionary, using the key "courses".

Question 6: Iterate over the "courses" dictionary and print each key-value pair.

```
# Q1
student =
    "name"  "John"
    "age"  20
    "major"  "Computer Science"

# Q2
student "gpa"  = 3.8
# Q3
student_major = student "major"
print student_major
# Q4
grades =
    "math"  90
    "science"  85
    "history"  95

# Q5
student "courses"  = grades
# Q6
for course  grade in student "courses"  items
    print course  ":"  grade
```

Comparing Lists and Dictionaries

Dictionaries are like lists except that they use keys instead of numbers to look up values

```
>>> lst = list()
>>> lst.append(23)
>>> lst.append(183)
>>> print(lst)
[23, 183]
>>> lst[0] = 23
>>> print(lst)
[23, 183]
```

```
>>> ddd = dict()
>>> ddd['age'] = 21
>>> ddd['course'] = 182
>>> print(ddd)
{'course': 182, 'age': 21}
>>> ddd['age'] = 23
>>> print(ddd)
{'course': 182, 'age': 23}
```

Activity 14.2

- How we can Count similar Words in Text by reading the text from file.

- Output should be in dict like this:
  - wordcounts = { ‘python’ : 1 , ‘training’ : 42, ‘exam’: 100}

Python Arrays

- Python does not have built-in support for Arrays, but Python Lists can be used instead.

- However, to work with arrays in Python you will have to import a library, like the **NumPy** library.

Python Modules

- As our program grows bigger, it may contain many lines of code. Instead of putting everything in a single file, we can use modules to separate codes in separate files as per their functionality. This makes our code organized and easier to maintain.

- Module is a file that contains code to perform a specific task. A module may contain variables, functions, classes etc. Let's see an example,

Custom Python Modules

- Let us create a module. Type the following and save it as example.py.

```
# Python Module addition

def add(a, b):

   result = a + b
   return result
```

- Here, we have defined a function **add()** inside a module named **example**. The function takes in two numbers and returns their sum.

Import Custom modules in Python

- We can import the definitions inside a module to another module in Python.

- We use the import keyword to do this. To import our previously defined module example, we type the following in the Python prompt.

```
import example
```

Import Custom modules in Python

- This does not import the names of the functions defined in example directly in the current symbol table. It only imports the module name example there.
- Using the module name we can access the function using the dot . operator in another module called calculatesum.py. For example:

```
p=example.add(4,5)  # returns 9
print(p)
```

Import Standard Built-in Python modules

- Python has tons of standard modules. Standard modules can be imported the same way as we import our user-defined modules.
- Suppose we want to get the value of pi, first we import the math module and use math.pi. For example,

```
# import standard math module
import math

# use math.pi to get value of pi
print("The value of pi is", math.pi)
```

Output:

```
The value of pi is 3.141592653589793
```

Python import with Renaming

- In Python, we can also import a module by renaming it. For example,

```
# import module by renaming it
import math as m

print(m.pi)

# Output: 3.141592653589793
```

- Here, We have renamed the **math** module as **m**. This can save us typing time in some cases.

- Note that the name **math** is not recognized in our scope. Hence, **math.pi** is invalid, and **m.pi** is the correct implementation.

Python from...import statement

- We can import specific names from a module without importing the module as a whole. For example:

```
# import only pi from math module
from math import pi

print(pi)

# Output: 3.141592653589793
```

- Here, we imported only the pi attribute from the math module.

Import all names

- In Python, we can import all names(definitions) from a module using the following construct:

```
# import all names from the standard module math
from math import *

print("The value of pi is", pi)
```

Activity - 15

- **Write a module called student_data in which user will input the student data(name, rollnumber and discipline) in a single list.**
- **Create another module called print_studentdata which will print the data of each student on a separate line by identifying it using delimiter.**

NumPy

- NumPy is a Python library created in 2005 that performs numerical calculations. It is generally used for working with arrays.
- NumPy is an important library generally used for:
  - Machine Learning
  - Data Science
  - Image and Signal Processing
  - Scientific Computing
  - Quantum Computing

Why Use NumPy?

- Major Reasons
  - Faster Execution (50x faster)
  - Used with Various Libraries
    - Pandas
    - Scipy
    - Scikit-learn

```
import numpy as np
```

NumPy Array Creation

- Create a NumPy array using a Python List. For example,

```
import numpy as np

# create a list named list1
list1 = [2, 4, 6, 8]

# create numpy array using list1
array1 = np.array(list1)

print(array1)

# Output: [2 4 6 8]
```

- Instead of creating a list and using the list variable with the np.array() function, we can directly pass list elements as an argument. For example,

```
import numpy as np

# create numpy array using a list
array1 = np.array([2, 4, 6, 8])
print(array1)

# Output: [2  4  6 8]
```

Create an Array Using np.zeros()

- The np.zeros() function allows us to create an array filled with all zeros. For example,

```
import numpy as np

# create an array with 4 elements filled with zeros
array1 = np.zeros(4)

print(array1)

# Output: [0. 0. 0. 0.]
```

Create an Array With np.arange()

- The np.arange() function returns an array with values within a specified interval. For example,

```
import numpy as np

# create an array with values from 0 to 4
array1 = np.arange(5)

print("Using np.arange(5):", array1)

# create an array with values from 1 to 8 with a step of 2
array2 = np.arange(1, 9, 2)

print("Using np.arange(1, 9, 2):",array2)
```

Create an Array With np.random.rand()

```
import numpy as np

# generate an array of 5 random numbers
array1 = np.random.rand(5)

print(array1)
```

Create an Empty NumPy Array

```
import numpy as np

# create an empty array of length 4
array1 = np.empty(4)

print(array1)
```

Python Namespace and

N-D Array Creation

- NumPy is not restricted to 1-D arrays, it can have arrays of multiple dimensions, also known as N-dimensional arrays or ndarrays.
- An N-dimensional array refers to the number of dimensions in which the array is organized.
- For example, a 2D array represents a table with rows and columns

Create a 2-D NumPy Array

- Let's create a 2D NumPy array with **2** rows and **4** columns using lists.

```
import numpy as np

# create a 2D array with 2 rows and 4 columns
array1 = np.array([[1, 2, 3, 4],
                   [5, 6, 7, 8]])

print(array1)
```

Create a 3-D NumPy Array

- How we can create a 3-D NumPy array consisting of two "slices" where each slice has
  - 3 rows and 4 columns.

```python
import numpy as np

# create a 3D array with 2 "slices", each of 3 rows and 4 columns
array1 = np.array([[[1, 2, 3, 4],
                    [5, 6, 7, 8],
                    [9, 10, 11, 12]],

                   [[13, 14, 15, 16],
                    [17, 18, 19, 20],
                    [21, 22, 23, 24]]])

print(array1)
```

[9] ✓ 0.0s

```
[[[ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10 11 12]]

 [[13 14 15 16]
  [17 18 19 20]
```

Creating N-d Arrays From Scratch

- To create multidimensional arrays from scratch we use functions such as

  - np.zeros()
  - np.arange()
  - np.random.rand()

Using np.zeros

```
import numpy as np

# create 2D array with 2 rows and 3 columns filled with zeros
array1 = np.zeros((2, 3))

print("2-D Array: ")
print(array1)

# create 3D array with dimensions 2x3x4 filled with zeros
array2 = np.zeros((2, 3, 4))

print("\n3-D Array: ")
print(array2)
```

Using np.arange()

```python
import numpy as np

# Create a 1D array with values from 0 to 11 (exclusive)
arr_1d = np.arange(12)
print("1D Array:")
print(arr_1d)

# Reshape the 1D array into a 3x4 2D array (3 rows and 4 columns)
arr_2d = arr_1d.reshape(3, 4)
print("\n2D Array:")
print(arr_2d)
```

Using np.random.rand()

```
import numpy as np

# Create a 1D array with values from 0 to 11 (exclusive)
arr_1d = np.random.rand(12)
print("1D Array:")
print(arr_1d)

# Reshape the 1D array into a 3x4 2D array (3 rows and 4 columns)
arr_2d = arr_1d.reshape(3, 4)
print("\n2D Array:")
print(arr_2d)
```

Cont.

```
import numpy as np

# create a 2D array of 2 rows and 2 columns of random numbers
array1 = np.random.rand(2, 2)

print("2-D Array: ")
print(array1)

# create a 3D array of shape (2, 2, 2) of random numbers
array2 = np.random.rand(2, 2, 2)

print("\n3-D Array: ")
print(array2)
```

N-D Array with a Specified Value

- We can use the np.full() function to create a multidimensional array with a specified value.

```
import numpy as np

# Create a 2-D array with elements initialized to 5
numpy_array = np.full((2, 2), 5)

print("Array:", numpy_array)
```

3-D Array using np.full()

```
import numpy as np

# Create a 3-D array with elements initialized to 5
numpy_array = np.full((2, 2, 2), 5)

print("Array:", numpy_array)
```

Activity 15.1

- How can we create empty 3-4 NumPy Array?

NumPy Data Types

- NumPy offers a wider range of numerical data types than what is available in Python. Here's the list of most commonly used numeric data types in NumPy:

  - int8, int16, int32, int64 - signed integer types with different bit sizes
  - uint8, uint16, uint32, uint64 - unsigned integer types with different bit sizes
  - float32, float64 - floating-point types with different precision levels
  - complex64, complex128 - complex number types with different precision levels

Data Type of NumPy Array

```
import numpy as np

# create an array of  integers
int_array = np.array([-3, -1, 0, 1])

# create an array of floating-point numbers
float_array = np.array([0.1, 0.2, 0.3])

# create an array of complex numbers
complex_array = np.array([1+2j, 2+3j, 3+4j])

# check the data type of int_array
print(int_array.dtype)  # prints int64

# check the data type of float_array
print(float_array.dtype)  # prints float64

# check the data type of complex_array
print(complex_array.dtype)  # prints complex128
```

Creating NumPy Arrays With a Defined Data Type

```
import numpy as np

# create an array of 32-bit integers
array1 = np.array([1, 3, 7], dtype='int32')

print(array1, array1.dtype)
```

NumPy Type Conversion

- We can convert the data type of an array using the astype() method. For example,

```
import numpy as np

# create an array of integers
int_array = np.array([1, 3, 5, 7])

# convert data type of int_array to float
float_array = int_array.astype('float')

# print the arrays and their data types
print(int_array, int_array.dtype)
print(float_array, float_array.dtype)
```

NumPy Array Attributes

ndim	returns number of dimension of the array
size	returns number of elements in the array
dtype	returns data type of elements in the array
shape	returns the size of the array in each dimension.
itemsize	returns the size (in bytes) of each elements in the array
data	returns the buffer containing actual elements of the array in memory

NumPy I/O Functions

- save() Function
  - np.save('file1.npy', array1)
- load() Function
  - loaded_array = np.load('file1.npy’)
- savetxt() Function
  - np.savetxt('file2.txt', array2)
- loadtxt() Function
  - loaded_array = np.loadtxt('file2.txt')

NumPy Array Indexing

```
import numpy as np

array1 = np.array([1, 3, 5, 7, 9])

# access numpy elements using index
print(array1[0])    # prints 1
print(array1[2])    # prints 5
print(array1[4])    # prints 9
```

Modify Array Elements Using Index

```
import numpy as np

# create a numpy array
numbers = np.array([2, 4, 6, 8, 10])

# change the value of the first element
numbers[0] = 12
print("After modifying first element:",numbers)    # prints [12 4 6 8 10]

# change the value of the third element
numbers[2] = 14
print("After modifying third element:",numbers)    # prints [12 4 14 8 10]
```

NumPy Array Negative Indexing

```
import numpy as np

# create a numpy array
numbers = np.array([1, 3, 5, 7, 9])

# access the last element
print(numbers[-1])    # prints 9

# access the second-to-last element
print(numbers[-2])    # prints 7
```

2-D NumPy Array Indexing

```
array1 = np.array([[1, 3, 5],
                   [7, 9, 2],
                   [4, 6, 8]])
```

```
array1[2, 1] # returns 6
```

Cont.

```
import numpy as np

# create a 2D array
array1 = np.array([[1, 3, 5, 7],
                   [9, 11, 13, 15],
                   [2, 4, 6, 8]])


# access the element at the second row and fourth column
element1 = array1[1, 3]  # returns 15
print("4th Element at 2nd Row:",element1)

# access the element at the first row and second column
element2 = array1[0, 1]  # returns 3
print("2nd Element at First Row:",element2)
```

Access Row or Column of 2D Array Using Indexing

```
import numpy as np

# create a 2D array
array1 = np.array([[1, 3, 5],
                   [7, 9, 2],
                   [4, 6, 8]])

# access the second row of the array
second_row = array1[1, :]
print("Second Row:", second_row)  # Output: [7 9 2]

# access the third column of the array
third_col = array1[:, 2]
print("Third Column:", third_col)  # Output: [5 2 8]
```

3-D NumPy Array Indexing

- To access an element of a 3D array, we use three indices separated by commas.

  - The **first** index refers to the **slice**
  - The **second** index refers to the **row**
  - The **third** index refers to the **column**.

Cont.

```
import numpy as np

# create a 3D array with shape (2, 3, 4)
array1 = np.array([[[1, 2, 3, 4],
                    [5, 6, 7, 8],
                    [9, 10, 11, 12]],

                     [[13, 14, 15, 16],
                     [17, 18, 19, 20],
                     [21, 22, 23, 24]]])

# access a specific element of the array
element = array1[1, 2, 1]

# print the value of the element
print(element)

# Output: 22
```

2D NumPy Array Slicing

```
# create a 2D array
array1 = np.array([[1, 3, 5, 7],
                   [9, 11, 13, 15]])

print(array1[:2, :2])

# Output

[[ 1  3]
 [ 9 11]]
```

Cont.

```
import numpy as np

# create a 2D array
array1 = np.array([[1, 3, 5, 7],
                   [9, 11, 13, 15],
                   [2, 4, 6, 8]])


# slice the array to get the first two rows and columns
subarray1 = array1[:2, :2]

# slice the array to get the last two rows and columns
subarray2 = array1[1:3, 2:4]

# print the subarrays
print("First Two Rows and Columns: \n",subarray1)
print("Last two Rows and Columns: \n",subarray2)
```