Unit II: Understanding Basic Data Types, Packages & Control Structures

2.1 Language Data Types & Abstract Data Types

In everyday life, we often categorize things based on their characteristics. Imagine you have a box to store items as given in the above example. You'd label the box based on what it contains, like “Kitchen” to store kitchen related materials like utensils, “Books” to store any reading books or diaries, likewise many more. Due to the label outside, we know what is stored inside the box, likewise data type in programming tells the computer what kind of data is inside or can be stored in variable or storage location.

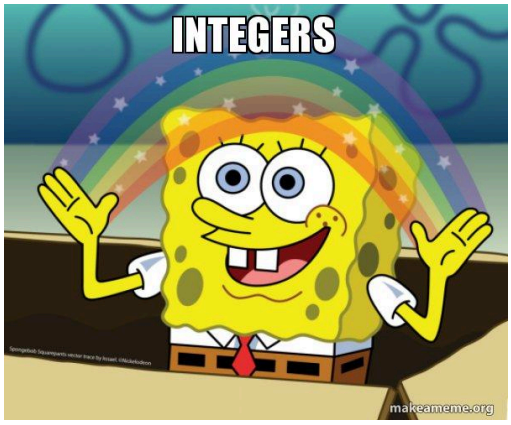
Data types: It defines the type of the data that can be stored and manipulated in a program.

- It defines the nature of the variable.
- It defines operations allowed on those values.

Two common types of data types are:

1. Primitive data types:
    - It is a basic data type that is directly supported by the programming language.
    - It is often used for simple tasks like arithmetic and comparison operations.

    - Some of the common primitive data types are:
        1. Int : Represents the whole number (integer type values).

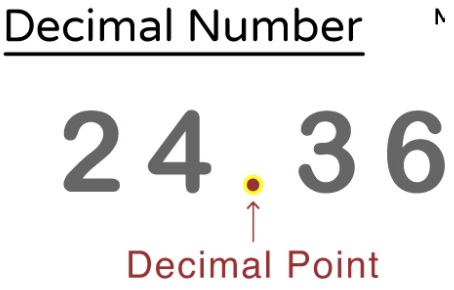


- It can be defined in python by directly assigning an integer value to the variable.

Eg: a= 1

To convert a specific value to integer, an int() function is used. This conversion of one data type to another data type is known as typecasting.

2. Float : Represents the real numbers with decimal.



- A number can be positive or negative of unlimited length.
- It can be defined in python by directly assigning a float value to the variable.

Eg: a=1.0

To convert a specific value to float, a float() function is used.

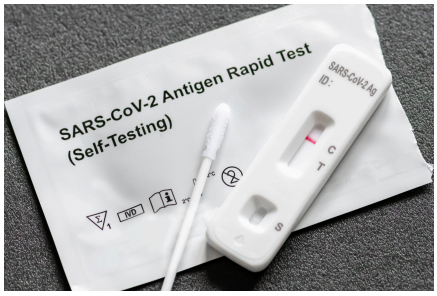
3. String : It is a sequence of characters.

- It can be defined in python by storing value for the variable in quotation marks. You will learn about string in detail when you learn string manipulation.

Eg: a = “Hello”

To convert a specific data into string type, str() is used.

4. Bool: Represents logical values (True or False)



- It is like a test kit whereby you get results either True or False.
- The bool() function helps to evaluate any value.
- Almost any value is evaluated to True if it has some sort of content which means in other words, as long as there is data in that variable or any data structure, it will return True.
- In fact, there are not many values that evaluate to False, except empty values, such as (), [], {}, "", the number 0, and the value None. And of course the value False evaluates to False.

Eg: print(1>0)

5. NONE : Represents the absence of value.

- Think of None data type as an empty parking lot. It is not that there won't be cars at all in that parking lot forever. What it means is that for that instant there ain't any cars but eventually that empty spaces can be filled with cars at any instant in the future.
- Likewise, the None keyword is used to define a null variable or an object or no value at all for that instant.

Class Activity: Code a python program following the instructions given below.
The code begins by prompting the user to enter their name, storing it in the variable name.
It then prompts the user to enter two numbers (num1 and num2) and stores them as floating-point numbers. The entered numbers (num1 and num2) are used to perform basic arithmetic operations: addition, subtraction, multiplication, and division. And the results of these operations are stored in separate variables (addition_result, subtraction_result, etc.). The program prints a personalized greeting along with the results using the print() function. Ensure that the print statements are clear and informative, providing a user-friendly output.

Sample Code:

```
# Prompt the user to enter their name
name = input("Enter your name: ")

# Prompt the user to enter two numbers
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))

# Perform arithmetic operations
addition_result = num1 + num2
subtraction_result = num1 - num2
multiplication_result = num1 * num2

# Check if num2 is not zero to avoid division by zero error
if num2 != 0:
    division_result = num1 / num2
else:
    division_result = "Undefined (division by zero)"
```

```
# Print personalized greeting and results
print(f"Hello, {name}!")
print("Here are the results of the arithmetic operations:")
print("Addition:", num1, "+", num2, "=", addition_result)
print("Subtraction: {} - {} = {}".format(num1, num2, subtraction_result))
print("Multiplication:", num1, "*", num2, "=", multiplication_result)
print("Division:", num1, "/", num2, "=", division_result)
```

2. Abstract Data Type:

- It is a high level description of data structures.
- It provides a more complex structure for storing and manipulating data.
- Primitive data types are often used to model the abstract data types. I.e. primitive data types are used to structure the abstract data type. The group of primitive data types represents the abstract data type.
- Common types of abstract data types are:

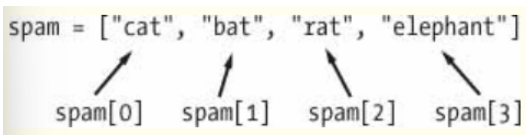
Data Structure	Ordered	Mutable	Constructor	Example
List	Yes	Yes	`[ ]` or `list()`	`[5.7, 4, 'yes', 5.7]`
Tuple	Yes	No	`( )` or `tuple()`	`(5.7, 4, 'yes', 5.7)`
Set	No	Yes	`{}`* or `set()`	`{5.7, 4, 'yes'}`
Dictionary	No	Yes**	`{ }` or `dict()`	`{'Jun': 75, 'Jul': 89}`

1. Lists : data are stored in square brackets([]).
   A list is a value that contains multiple values in an ordered sequence. The values inside lists are also called items. Items are separated by commas. Items in a list can be of different data types.

   Eg: `['hello', 3.1415, True, None, 42]`

   In the list, indexing order is preserved which means the items in the list will be in same order as they were inserted in the list, due to which list items can be accessed with an index.

   Eg:



- It is mutable, which means we can perform operations on that list to update the changes in it.
- Duplication is allowed. Lists can store the same items in it.
- Slicing is allowed. Just as an index can get a single value from a list, a slice can get several values from a list, in the form of a new list. In a slice, the first integer is the index where the slice starts. The second integer is the index where the slice ends.

Eg:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0:4]
['cat', 'bat', 'rat', 'elephant']
>>> spam[1:3]
['bat', 'rat']
>>> spam[0:-1]
['cat', 'bat', 'rat']
```

- Enumerate function : On each iteration of the loop, enumerate() will return two values: the index of the item in the list, and the item in the list itself.

Eg:

```
>>> supplies = ['pens', 'staplers', 'flamethrowers', 'binders']
>>> for index, item in enumerate(supplies):
...     print('Index ' + str(index) + ' in supplies is: ' + item)


Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flamethrowers
Index 3 in supplies is: binders
```

List can be compared to shopping mall racks which will contain multiple grocery items in it. It can store the same item too.

- Some of the operations of lists are:
  1. index() : if the value exists in the list, the index of the value is returned. If the value isn't in the list, then Python produces a ValueError error.

```
my_list = [1,2,3]
print(my_list.index(2)) # Output: 1
```

In the above example, the index of the item '2' is 1, therefore output will be 1.

  2. Append and insert

The append() method call adds the argument to the end of the list. The insert() method can insert a value at any index in the list.

Eg:

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.append('moose')
>>> spam
['cat', 'dog', 'bat', 'moose']
```

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.insert(1, 'chicken')
>>> spam
['cat', 'chicken', 'dog', 'bat']
```

3. Remove

Removes an item from the list.

```
>>> spam
['cat', 1, 'dog', 'bat']
```

```
>>> spam.remove('cat')
>>> spam
[1, 'dog', 'bat']
```

4. Sort & reverse

Lists of number values or lists of strings can be sorted with the sort() method. When passed a true value to reverse, the items will be sorted out in a reverse order, or just use reverse() function.

Eg:

```
>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
>>> spam.sort()
>>> spam
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

```
>>> spam.sort(reverse=True)
>>> spam
['elephants', 'dogs', 'cats', 'badgers', 'ants']
```

```
>>> spam = ['cat', 'dog', 'moose']
>>> spam.reverse()
>>> spam
['moose', 'dog', 'cat']
```

2. Set : Data is stored in braces ({}).

Set is also a collection of heterogeneous data, i.e. it can store multiple values of different data types. Just like lists, sets are also mutable.

In a set, insertion order is not preserved, due to which the order of the value you insert in a set will be randomized. Therefore, the values in a set are not accessible with an index. Unlike lists, slicing is not allowed. Additionally, storing duplicate values in a set is not possible.

```
>>> spam = {'cat', 'dog', 1, 'monkey'}
>>> spam
{1, 'cat', 'dog', 'monkey'}
```

Set can be compared to the follower list of your instagram whereby each follower will have their unique usernames. Therefore, follower list will contain multiple unique usernames.

Some of the operations of set are:

1. Add

```
>>> spam.add(2)
>>> spam
{1, 2, 'cat', 'monkey', 'dog'}
```

To add an item in a set, you will used add() method.

2. Remove

```
>>> spam.remove(1)
>>> spam
{2, 'cat', 'monkey', 'dog'}
```

remove() method is used to remove the items from the set.

3. Union

```
>>> spam
{2, 'cat', 'monkey', 'dog'}
>>> spam1 = {2, True}
>>> unionSpam = spam.union(spam1)
>>> unionSpam
{True, 2, 'cat', 'monkey', 'dog'}
```

Union method will club all the unique items from two sets in a set.

4. Intersection

```
>>> intersecSpam = spam.intersection(spam1)
>>> intersecSpam
{2}
```

From the two sets, it will return common items from them.

5. difference

```
>>> differSpam = spam.difference(spam1)
>>> differSpam
{'cat', 'dog', 'monkey'}
```

The difference method will return the unique items in first set that is not in the second set.

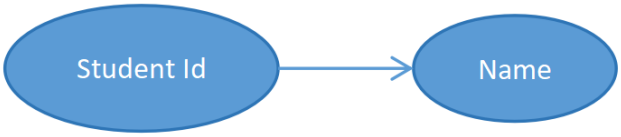
3. Dictionary: The data in the dictionary are stored in braces ({}).
Like a list and set, a dictionary is also a mutable collection of many values. But unlike indexes for lists, indexes for dictionaries can use many different data types, not just integers. Indexes for dictionaries are called keys, and a key with its associated value is called a key-value pair.

Eg:
```
>>> myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

In the above example, a dictionary called “myCat” is storing key-value pair data whereby keys are “Size, color and disposition” and values their values are, “fat, gray and loud” respectively.

The duplication in key is not allowed, however value can be duplicated. In case, if the key is duplicated, the value for the key will be changed to the latest value of the given key. Like sets, items in dictionaries are unordered and dictionaries can’t be sliced too.

Eg:
```
>>> spam = ['cats', 'dogs', 'moose']
>>> bacon = ['dogs', 'moose', 'cats']
>>> spam == bacon
False
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> eggs == ham
True
```



Think of a dictionary as a record sheet containing student Id and the respective names. Each student will have a unique student Id but can have the same names. So, for this reason, the student will be identified by its student id in that record. Here student id is acting as a key and name is its respective values.

Some of the operations of the dictionary are:

For an example, the spam dictionary contains its items in it as shown below:

```
>>> spam = {'name' : 'Alice', 'age' : 20}
```

1. Get

```
>>> spam.get('name')
'Alice'
```

The get method in the dictionary is used to return the particular value of that specific key.

2. Setdefault:

```
>>> spam.setdefault('height', 180)
180
>>> spam.items()
dict_items([('name', 'Alice'), ('age', 20), ('height', 180)])
```

The setdefault method in dictionary helps to add an item in the dictionary.

3. Keys

```
>>> spam = {'name' : 'Alice', 'age' : 20}
>>> spam.keys()
dict_keys(['name', 'age'])
```

The keys method will return all the keys of that dictionary.

4. Values

```
>>> spam.values()
dict_values(['Alice', 20])
```

The values method of the dictionary will return all the values of that dictionary.

5. items

```
>>> spam.items()
dict_items([('name', 'Alice'), ('age', 20), ('height', 180)])
```

It will return all the key-value pairs of that dictionary.

***

NOTE: In order to print just values of the abstract data structure without printing its respective brackets, you can make use of the * in the print function.

Eg:

```
my_list = [1,2,3]
print(my_list) # Output: [1, 2, 3]
print(*my_list) # Output: 1 2 3
```

***

2.2 String Manipulation

By default data will always be in string format i.e. text. String is actually stored in the form of an array. The string begins and ends with a quotation mark. For a multiline string like to print paragraphs, in Python string should be wrapped within either three single quotes or three double quotes. In order to use an apostrophe in a string, you need to wrap the string inside a double quote.

eg:
```
>>> spam = "That is Alice's cat."
```

Different string manipulation in python:

1. len()
   This method returns the length of the string.
```
>>> spam = 'Hello, world!'
>>> print(len(spam))
13
```

2. upper()
   This method returns a new string where all the letters in the original string have been converted to uppercase.

   Eg:
```
>>> spam = 'Hello, world!'

>>> spam = spam.upper()

>>> spam

'HELLO, WORLD!'
```

3. lower():
   This method returns a new string where all the letters in the original string have been converted to lowercase.

   Eg:
```
>>> spam = spam.lower()

>>> spam

'hello, world!'
```

4. isX():
   These methods return a Boolean value that describes the nature of the string. The isX() string methods are helpful when you need to validate user input.
   Some of the commonly used isX() methods are:
   1. islower(): Checks if all the characters in the given string are in lower case or not.
   2. isupper(): Checks if all the characters in the given string are in upper case or not.

Eg:

```
>>> 'HELLO'.isupper()

True

>>> 'abc12345'.islower()

True
```

5. count():
   It returns the number of times a character appears in the string.

Eg:

```
>>> a = "Hello"
>>> print(a.count('l'))
2
>>> print(a.count('H'))
1
```

6. find()
   This method finds the first occurrence of the specified value in the string. It returns the index (position) of that value.

Eg:

```
>>> a = "Hello"
...
>>> print(a.find('e'))
...
1
>>>
```

7. strip()
   It is used to remove extra whitespaces and specific characters from the string.

Eg:

```
>>> a = "   ,  Hello  "
>>> print(a)
   ,  Hello
>>> print(a.strip())
,  Hello
>>> a = a.strip(" ,")
>>> print(a)
Hello
```

8. split()
   The split() function in Python is used to split a string into a list of substrings based on a specified delimiter. By default, the delimiter is a whitespace, but can specify any character.

```
>>> spam = "Hello, Welcome to whereever"
>>> print(spam)
Hello, Welcome to whereever
>>> spam.split()
['Hello,', 'Welcome', 'to', 'whereever']
>>> spam.split(",")
['Hello', ' Welcome to whereever']
```

9. Escape Characters

An escape character consists of a backslash (\) followed by the character you want to add to the string. Even though it consists of two characters, it is commonly referred to as a singular escape character.

Escape character	Prints as
\'	Single quote
\"	Double quote
\t	Tab
\n	Newline (line break)
\\	Backslash

Eg:

```
>>> print("Hello there!\nHow are you?\nI\'m doing fine.")
Hello there!
How are you?
I'm doing fine.
```

However, there's a concept of raw string which completely ignores all escape characters and prints any backslash that appears in the string. Place an r before the beginning quotation mark of a string to make it a raw string.

Eg:

```
>>> print(r'That is Carol\'s cat.')
That is Carol\'s cat.
```

10. Formatted String

It is a way to create strings that include variables or expressions, allowing you to embed values within the string. Certain placeholders are being used in between the original string so that these placeholders can be replaced with certain specific values.

1. F-string

It is created by prefixing the string with the letter 'f' or 'F'.

eg:
```
>>> name = "Karma"
>>> print(f"Hello {name}")
Hello Karma
```

2. format() method:
This formats the specified value(s) and inserts them inside the string's placeholder.

eg:
```
>>> print("Greeting to you, {}".format(name))
Greeting to you, Karma
```

3. % format:
The following string literals are inserted in between the character as placeholder in the string.

- %d and %I: used for integer type of data.
- %f : for float values.
- %s : for string type of values.

Eg:
```
>>> name = "Karma"
>>> Number = 2
>>> print("There are %d %s in the class" %(Number, name))
There are 2 Karma in the class
```

11. Concatenation:
The meaning of operators change based on the data types of the values. The “+” operator is used for addition when the data type is int or float, however in string, it is used to join the strings. This joining of strings is known as string concatenation. In other words, a new single string is formed.

Eg:
```
a = "Hello"
b = "World"
c = a + " " + b + "!"
print(c)

#Output: Hello World!
```

If + operator is used for string and an integer, it will display an error message.

Eg:
```
print("Hello" + 2)
#Output: TypeError: can only concatenate str (not "int") to str
```

join(): We can also make use of this method to join the list of strings.

Eg:

```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
```

12. String Replication:

    The * operator multiplies two integer or floating-point values. But when the * operator is used on one string value and one integer value, it becomes the string replication operator. In other words, the string is getting duplicated the given number of integer times.

Eg:

```
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```

13. Comments:

    Python ignores comments, and you can use them to write notes or remind yourself what the code is trying to do. Any text for the rest of the line following a hash mark (#) is part of a comment. Sometimes, programmers will put a # in front of a line of code to temporarily remove it while testing a program. This is called commenting out code, and it can be useful when you're trying to figure out why a program isn't working. You can remove the # later when you are ready to put the line back in. The triple quote is often used for commenting on multiple lines.

Eg:

```
"""

This  code is under comment.

"""
```

14. Indexing and Slicing:

    Since, string is stored in the form of an array, we can make use of index to get the character of that position, and we can make use of slicing to get certain characters from the String.

    Eg: Let's say a variable name "spam" contains the string "Hello, world".

'	H	e	l	l	o	,		w	o	r	l	d	!	'
	0	1	2	3	4	5	6	7	8	9	10	11	12	

The space and exclamation point are included in the character count, so 'Hello, world!' is 13 characters long, from H at index 0 to ! at index 12.

1. Getting value stored at index 4 and -1:

```
>>> spam[4]
'o'
>>> spam[-1]
'!'
```

2. Slicing till index 5 and also, slicing till index 7:

```
>>> spam[:5]
'Hello'
>>> spam[7:]
'world!'
```

2.3 Multi-Dimensional Arrays & Abstract Data Structures

Array: It is the collection of data that stores finite multiple values of the same type at adjacent (contiguous) memory locations.

The idea is to store multiple items of the same type together. The items that are stored inside an array is called **Element**, whereas the numerical position of each box in the array is known as **Index**. However, take note that index and address aren't the same thing in array. **Address** in other hand is a numerical value of the first byte at which the item is stored. In simple terms, array index is just a label to each box of array, whereas array address is the physical location of the overall array.

Eg: We can make use of an array to store the index number of university students.

In order to use an array in python, an "array"m package needs to be imported.

```
from array import *
```

To define an array:

```
VAR=ARRAY(TYPE CODE, [ELEMENTS])
```

Eg:

```
empty_arr = array('B')
```

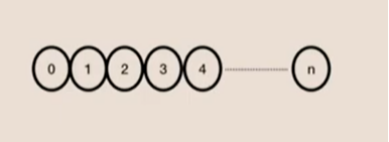
```
arr = array('i', [10, 20, 30, 40, 50])
print(type(arr)) #<class 'array.array'>
```

TypeCode	C Type	Python Type
'b'	signed char	int
'B'	unsigned char	int
'u'	Py_UNICODE	Unicode character
'h'	signed short	int
'H'	unsigned short	int
'i'	signed int	int
'I'	unsigned int	int
'l'	signed long	int
'L'	unsigned long	int
'f'	float	float
'd'	double	float

- Types of arrays:

NOTE: We can make use of list to define an array too.

1. Single dimension (1-D) array



Eg:
```
toys = ["doll", "car", "ball", "blocks"]
```

Methods in array:

1. len() : it returns the number of elements in the array.
```
arr = array('i', [10, 20, 30, 40, 50])
print(len(arr)) # 5
```

2. Indexing: We pass the index of that element and as an output, it will return the value.
```
print(arr[2]) # 30
```

3. index() : Just opposite to the indexing, we pass value of the array as an argument and then as a output it will return index of that particular element.

```
print(arr.index(30)) # 2
```

4. append() : A new specified element will be added at the end of the array.

```
arr.append(60)
```

5. insert() : The specified element will be added to the specified index in the array. The first argument of insert() method represents index and second argument represents an element that needs to be inserted in the array.

```
arr.insert(2, 70)
```

6. extend(): With the help of this method, multiple elements can be added in an array.

```
arr.extend([70, 80, 90])
```

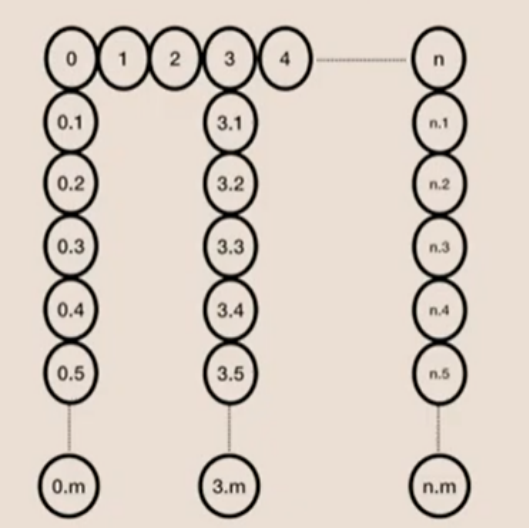
7. remove(): It removes an element from the array. The specific element which needs to be removed is passed as an argument for remove() method.

```
arr.remove(90)
```

8. pop(): It removes the element from the specified index. If no argument is passed in pop() method, it will just remove the last element from the list.

```
arr.pop(0)
```

2. Multi-dimensional arrays (N-D) : It is also known as matrices. To create a multidimensional array, an array is stored in an array.



Eg:

```
candies = [
    ["chocolate", "toffee", "caramel"],
    ["fruit", "sour", "gummy"]
]
```

In order to implement the concepts of multidimensional arrays without using lists, we need to import third party package called numpy.

```
import numpy as np
```

Then array can be defined using the alias name of numpy i.e. np:

```
arr2 = np.array([[10, 20, 30, 40, 50], [60, 70, 80, 90, 100]])
print(arr2) # [[ 10  20  30  40  50] [ 60  70  80  90 100]]
```

We consider that arrays are being stored in rows, whereby each individual array within an array represents rows. As per the above example, since there are two arrays within an array, it means that there are two rows. The element in each array will represent the number of columns. As per the above example, there are five columns in the array.

While indexing in multiple arrays, the first index represents the row number and the second one represents the column number.

```
print(arr2[0, 2]) # 30
```

We can perform mathematical operations in arrays.

Eg:

```
print(arr2.sum()) # 550
```

```
print(arr2.sum(axis=0)) # [ 70  90 110 130 150]
print(arr2.sum(axis=1)) # [150 400]
```

When the axis is “0” represents the column whereas when the axis is “1”, it represents the rows.

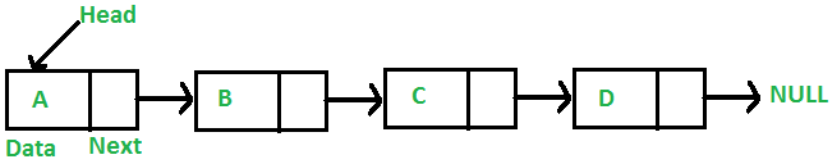
Array concept in python is implemented with the help of lists. Some of the array methods used in python are:

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Data Structures:

- Some of the commonly used data structures are:

1. Linked List : It is a linear data structure in which elements are stored in the form of chains using pointers(Address or reference to the next node). The structure of a linked list is such that each piece of data has a connection to the next one (and sometimes the previous data as well). Each element in a linked list is called a node. The first node is called Head.



There are two types of linked lists, they are:

1. Singly linked list: Each node contains two parts: the data, which holds the value of the element, and a reference (or pointer) to the next node in the sequence. The pointer will contain the address of the next node The last node typically points to a null value to signify the end of the list.

Sample implementation in python code:

To implement the concept of linked list, no other third party packages can be used. Instead everything needs to be coded.

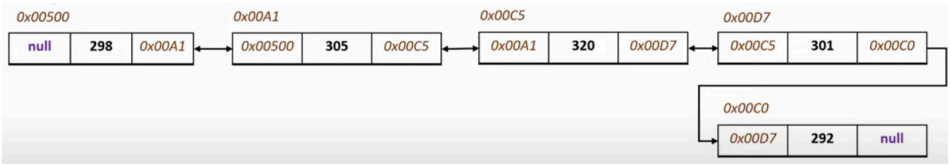
There should be two classes for the linked list, one Node which will contain data and a pointer in it. Another one is linkedlist itself which will contain the methods like add nodes, remove nodes and so on.

```
# Define a class named Node representing a single node in a singly linked list.
class Node:
       #    method for the Node class. Node will have two attributes: data and next.
    def __init__(self, data):
          # Initialize the data attribute of the node with the given data.
        self.data = data
          # Initialize the next attribute of the node as None, indicating that it initially does not point to any other node as its empty.
        self.next = None
```

```
# Define a class named SinglyLinkedList representing a singly linked list.
class SinglyLinkedList:
     # Define the constructor method for the SinglyLinkedList class.
    def __init__(self):
         # Initialize the head attribute of the linked list as None, indicating an empty list.
        self.head = None

    # Define a method named append to add a new node with the given data to the end of the linked list.
    def append(self, data):
        # Create a new node object with the given data.
        new_node = Node(data)
        # Check if the linked list is empty.
        if not self.head:
             # If the linked list is empty, set the new node as the head of the linked list and return.
            self.head = new_node
            return
         # If the linked list is not empty, traverse the linked list to find the last node.
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
         # Once the last node is found, set the next attribute of the last node to the new node, effectively adding it to the end of the linked list.
        last_node.next = new_node

     # Define a method named display to print the elements of the linked list.
    def display(self):
        # Start from the head of the linked list.
        current = self.head
         # Traverse the linked list and print the data of each node.
        while current:
            print(current.data, end=" -> ")
```

```
            # Move to the next node in the linked list.
            current = current.next
        # Print "None" to signify the end of the linked list.
        print("None")

# Example usage:
# Create an instance of the SinglyLinkedList class.
sll = SinglyLinkedList()
# Append some elements to the linked list.
sll.append(1)
sll.append(2)
sll.append(3)
# Display the elements of the linked list.
sll.display()
```

2. Doubly linked list: In a doubly linked list, each node contains an extra pointer that points to the previous node in addition to the next node. This allows traversal in both forward and backward directions. So, in doubly linked list, there will be two pointers and data in a node.



Sample implementation in python code:
Everything is similar to singly linked list code except that class node will have one extra attribute which will point to previous node.

```
# Define a class named Node representing a single node in a
doubly linked list.
class Node:
    # Define the method for the Node class.
    def __init__(self, data):
        # Initialize the data attribute of the node with the
given data.
        self.data = data
        # Initialize the next attribute of the node as None,
indicating that it initially does not point to any other node.
        self.next = None
```

```
        # Initialize the prev attribute of the node as None,
indicating that it initially does not point to any previous
node.
        self.prev = None

# Define a class named DoublyLinkedList representing a doubly
linked list.
class DoublyLinkedList:
    # Define the method for the DoublyLinkedList class.
    def __init__(self):
        # Initialize the head attribute of the linked list as
None, indicating an empty list.
        self.head = None

    # Define a method named append to add a new node with the
given data to the end of the linked list.
    def append(self, data):
        # Create a new node object with the given data.
        new_node = Node(data)
        # Check if the linked list is empty.
        if not self.head:
            # If the linked list is empty, set the new node as
the head of the linked list and return.
            self.head = new_node
            return
        # If the linked list is not empty, traverse the linked
list to find the last node.
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        # Once the last node is found, set the next attribute
of the last node to the new node, effectively adding it to the
end of the linked list.
        last_node.next = new_node
        # Set the prev attribute of the new node to the last
node, establishing the backward link.
        new_node.prev = last_node

    # Define a method named display to print the elements of
the linked list.
```

```
    def display(self):
        # Start from the head of the linked list.
        current = self.head
        # Traverse the linked list forwards and print the data
of each node.
        while current:
            print(current.data, end=" <-> ")
            # Move to the next node in the linked list.
            current = current.next
        # Print "None" to signify the end of the linked list.
        print("None")

# Example usage:
# Create an instance of the DoublyLinkedList class.
dll = DoublyLinkedList()
# Append some elements to the linked list.
dll.append(1)
dll.append(2)
dll.append(3)
# Display the elements of the linked list.
dll.display()
```

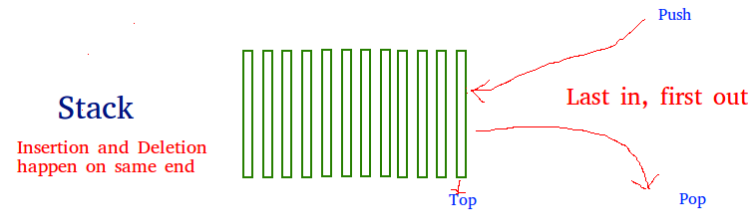
Some of the methods associated with linked lists are:

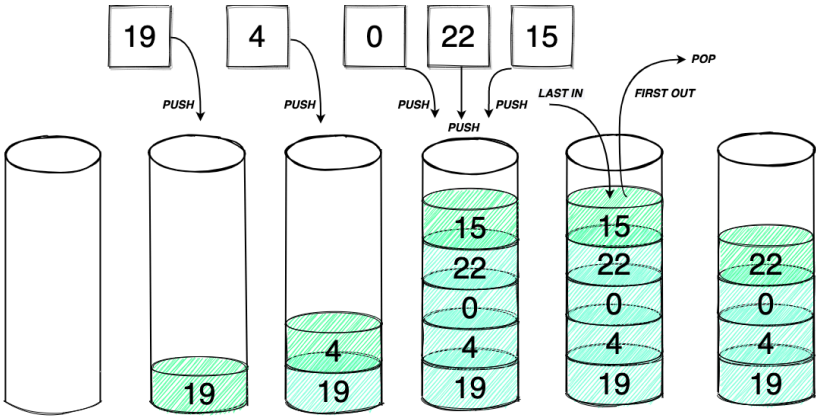
1. insert() : Add an item to the linked list at the head of the list
2. find() : Find an item within the linked list
3. Remove() : Remove a given item with a given value
4. is_empty() : Returns whether the linked list is empty or not
5. get_count() : Returns the number of items in the linked list

2. Stack :

For instance, the waiter cleans the plate and stacks it on top of each other as shown in the above image. Then guests will take the plate which is on the top to serve the food for themselves.

Similarly, stack in python is a linear data structure that stores items in a Last-In/First-Out (LIFO) or First-In/Last-Out (FILO) manner. Whereby the item that is added at the end will be removed first.





- Some of the methods associated with stacks are:
    1. push() : Inserts the element at the top of the stack.
    2. pop() : Deletes the topmost element of the stack.
    3. empty() : Returns whether the stack is empty.
    4. size() : Returns the size of the stack.

- Stack in python can be implemented in three ways:
    1. Lists: implementing stack using lists
       Sample code in python:

```
stack = []
stack.append(10)
stack.append('H')
stack.append(0)
print(stack) # [10, 'H', 0]

stack.pop()
print(stack) # [10, 'H']
```

    2. Collections.queue: Implementing stack using dequeue
       Sample code in python:

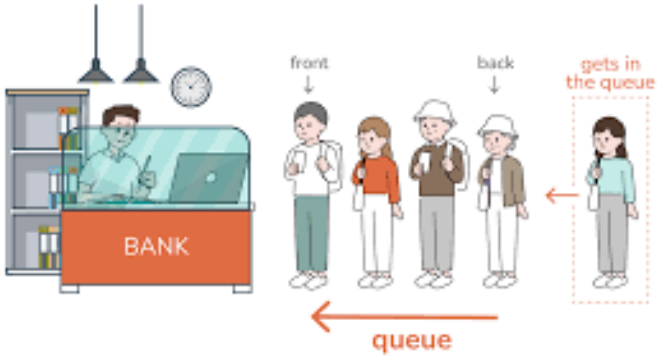
```
#Stack usinfg dequeue
from collections import deque

stack = deque()

# append() function to push
# element in the stack
stack.append('g')
stack.append('f')
stack.append('g')
print(stack) # deque(['g', 'f', 'g'])



print(stack.pop())
print(stack.pop())
print(*stack) # g
```

3. Queue.LifoQueue : Implementing stack using queue package library.
   Sample code in python:

```
# #Stack using queue
from queue import LifoQueue

stack = LifoQueue(maxsize = 3)

# qsize() show the number of elements
print(stack.qsize()) # 0

# put() function to push
stack.put('g')
stack.put('f')
stack.put('g')
print(stack.queue) # ['g', 'f', 'g']

print("Full: ", stack.full()) # True
print("Size: ", stack.qsize())  # 3

# get() function to pop
print(stack.get())
print(stack.queue) # ['g', 'f']
```

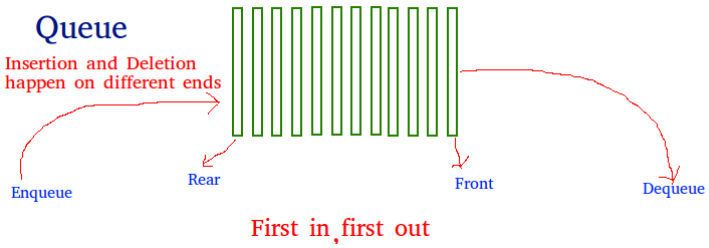
```
print(stack.get())
print(stack.get())

print("\nEmpty: ", stack.empty())
```

3. Queue :



For instance, think of customers in line to access the bank service. In there, whoever is first in the line or whoever reached earlier can access the service faster than the one who has reached later and is in the last line.

Similarly, queue is a linear data structure that stores elements in a First in First Out (FIFO) manner in which the least recently added item is removed first unlike stack.



- Some of the methods associated with queue are;
    1. Enqueue : Adds an item in the queue.
        - Overflow : It's the condition when the queue is full.
    2. Dequeue : Removes an item from the queue.
        - Underflow : Its the condition when the queue is empty.
    3. Front : Gets the front item from the queue.
    4. Rear : Gets the last item from the queue.

- In python, queue can be implemented in three ways:
    1. List:
       Sample Code:

```
# Initializing a queue
queue = []
```

```
# Adding elements to the queue
queue.append('g')
queue.append('f')
queue.append('g')
print(queue) # ['g', 'f', 'g']

# Removing elements from the queue
print(queue.pop(0))
print(queue) # ['f', 'g']
print(queue.pop(0))
print(queue.pop(0))

print("\nQueue after removing elements")
print(queue)
```

2. Collections.deque:
   Sample python code:

```
from collections import deque

q = deque()

# Adding elements to a queue
q.append('g')
q.append('f')
q.append('g')
print(q) # deque(['g', 'f', 'g'])

# Removing elements from a queue
print(q.popleft())
print(q) # deque(['f', 'g'])
print(q.popleft())
print(q.popleft())

print("\nQueue after removing elements")
print(*q) # Empty queue
```

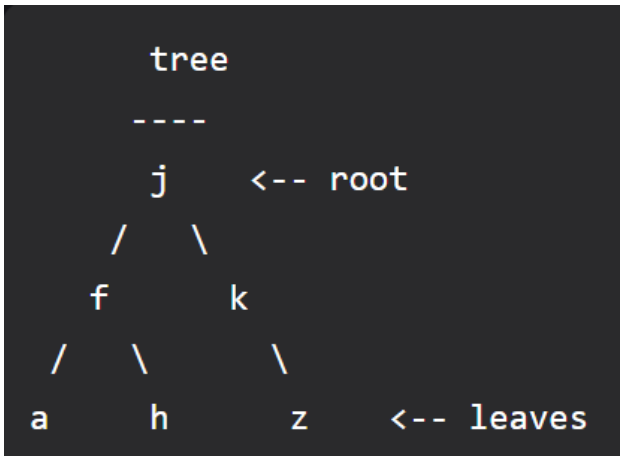
3. queue.Queue:
   Sample Code:

```
#Implementing queue using queue
```

```
from queue import Queue

q = Queue(maxsize = 3)

# qsize() give the maxsize
# of the Queue
print(q.qsize())

# Adding of element to queue
q.put('g')
q.put('f')
q.put('g')
print(q.queue) # ['g', 'f', 'g']


# Return Boolean for Full
# Queue
print("\nFull: ", q.full())

# Removing element from queue
print(q.get())
print(q.queue) # ['f', 'g']
print(q.get())
print(q.get())

# Return Boolean for Empty
# Queue
print("\nEmpty: ", q.empty())

q.put(1)
print("\nEmpty: ", q.empty())
print("Full: ", q.full())
```

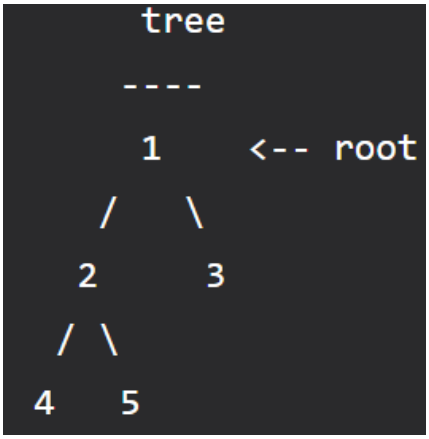
- Priority Queue : It is an abstract data structure where each item in the queue has a certain priority.
- Priority Queue is an extension of queue consisting of following properties:
    1. An element with high priority is dequeued before the element with low priority .
    2. If two elements are having the same priority, they are served according to their order in the queue.

4. Binary Tree : It is a hierarchical data structure that can have the utmost two children.

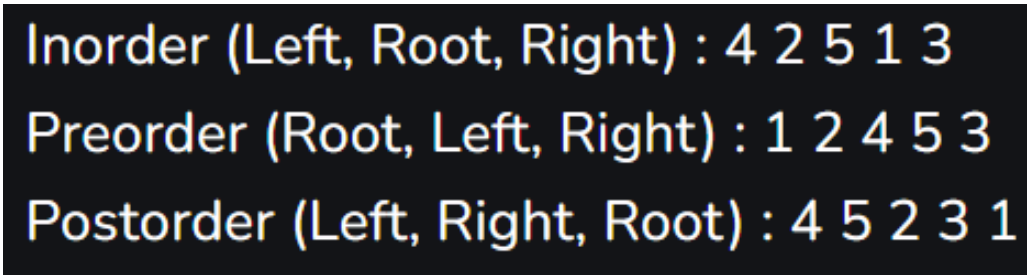


- The topmost node of the tree is called the root whereas the bottommost nodes or the nodes with no children are called the leaf nodes.
- The nodes that are directly under a node are called its children and the nodes that are directly above something are called its parent.
- The different types of tree traversal are as follows:

Eg:

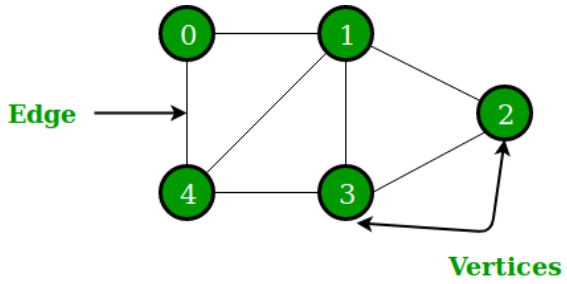


1. Depth First Traversal: As the name suggests, involves traversing a tree deeply before exploring siblings. There are three variants to traverse using depth first traversal.



2. Breadth-First or Level Order Traversal: Traversing a tree level by level, visiting all nodes at the present depth before moving to nodes at the next depth.

5. Graphs : It is a nonlinear data structure consisting of nodes and edges.
   - The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.



The different types of graph traversal are:

1. Breadth-First Search (BFS)
2. Depth First Search

2.4 Standard & Third-Party Language Packages

- A Python package is a directory of a collection of modules to perform specific tasks.
- Standard python library: The module which is automatically installed during the time of python installation but needs to be loaded to use it.

Eg: random module

- Third-Party Language Packages: Some useful modules that need to be installed. In python, packages can be installed with the help of pip.

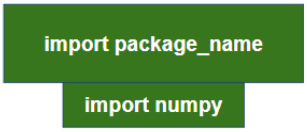
Eg: NumPy

Need to first install the package in terminal:

```
pip install <package_name>
```

```
pip install numpy
```

Then to use it in code, need to import it in a code.


import package_name
import numpy

References

(n.d.) Python tutorial. Available at: https://www.w3schools.com/python

Automate the boring stuff with python (n.d.) Automate the Boring Stuff with Python book cover thumbnail. Available at: https://automatetheboringstuff.com/

Data types - premetive and non premetive (2016) SlideShare. Available at: https://www.slideshare.net/RajNaik12/data-types-premetive-and-non-premetive

(n.d.) Codercuy.com. Available at: https://codercuy.com/primitive-types-and-abstract-data-types/

Lemonaki, D. (2023) Python array tutorial – define, index, methods, freeCodeCamp.org. Available at: https://www.freecodecamp.org/news/python-array-tutorial-define-index-methods/

OluseyeJeremiah (2023) Multi-dimensional arrays in python – matrices explained with examples, freeCodeCamp.org. Available at: https://www.freecodecamp.org/news/multi-dimensional-arrays-in-python/

Implement stack in python using queue.LifoQueue (2017) YouTube. Available at: https://www.youtube.com/watch?v=hlj9KyCYduc

An introduction to python linked list and how to create one (n.d.) Built In. Available at: https://builtin.com/data-science/python-linked-list

GfG (2023) Python Data Structures, GeeksforGeeks. Available at: https://www.geeksforgeeks.org/python-data-structures/

YouTube. (2020, August 7). What's the difference between Python's Standard Library and builtins? YouTube. https://www.youtube.com/watch?v=fO0dGvdMIos

The Python Standard Library. Python documentation. (n.d.). https://docs.python.org/3/library/index.html