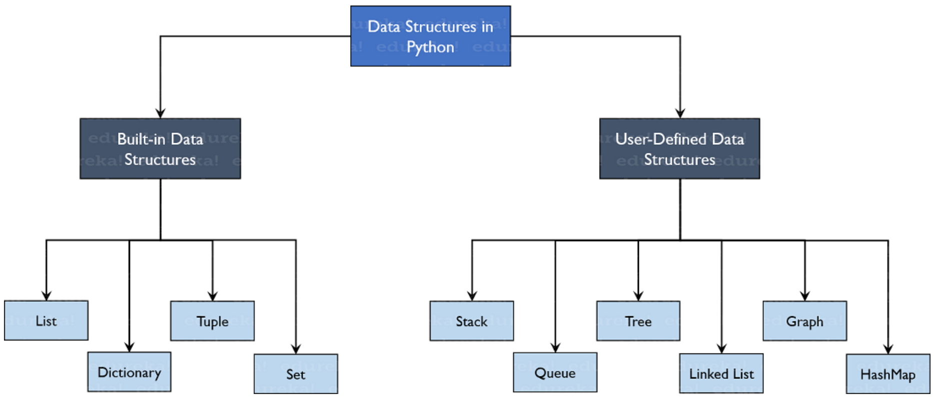
Data structure in Python

1 Definition

- **Data Structures**: organize your data in such a way that enables you to store collections of data, relate them and perform operations on them accordingly.

2 Types of Data Structures in Python



- **Built-in Data Structures**: list, dictionary, tuple, set
- **User-defined data structures**: stack, queue, Tree, linked list, graph, HashMap

3 List (built-in)

- **Definition**: Lists are used to store data of different data types in a sequential manner.
- **Index**: address assigned to every element of the list. The index value starts from 0 and goes on until the last element called the positive index. There is also negative indexing which starts from -1 enabling you to access elements from the last to the first.
- **Example**:

```python
'''
Data structure: list
'''
print('----------------------------List-------------------------')
a=[]
print('a=',a)
b=[1,'1',2,'2','Hellow world!']
print('b=',b)

print('Positive index: b[3]=',b[3])
print('Negative index: b[-1]=',b[-1])


b.append([555, 12]) #add as a single element
print('Append [555, 12], b=',b)
b.extend([234, 'more_example']) #add as different elements
print('add another two elements, b=',b)
print('Lenght of b is', len(b)) # Length of b


for element in b: #access elements one by one
    print(element)

print(b[0]==b[1])
```

```python
my_list = [1, 2, 3, 'example', 3.132, 10, 30]
del my_list[5] #delete element at index 5
print(my_list)
my_list.remove('example') #remove element with value
print(my_list)
a = my_list.pop(3) #pop element from index 1 of list
print('Popped Element: ', a, ' List remaining: ', my_list)
my_list.clear() #empty the list
print(my_list)


my_list = [1, 2, 3, 10, 30, 10]
print(len(my_list)) #find length of list
print(my_list.index(10)) #find index of element that occurs first
print(my_list.count(10)) #find count of the element
print(sorted(my_list)) #print sorted list but not change original
my_list.sort(reverse=True) #sort original list in descent order
print(my_list)
```

4 Dictionary(built-in)

- **Definition**: Dictionaries are used to store key-value pairs.
- **Possible operations**: create a dictionary, Changing and Adding key-value pairs, Deleting key-value pairs, Accessing Elements
- Examples:

```python
'''
Data structure: dictionary
'''
print('----------------------------dictionary-------------------------')
mydict={}
print(mydict)
mydict={1:'python', 2:'C'}
print(mydict)

mydict[3]='Java' # insert key-value pair
print('Insert a pair 3:Java, mydict=', mydict)

my_dict = {'First': 'Python', 'Second': 'Java'}
print(my_dict)
my_dict['Second'] = 'C++' #changing element
print(my_dict)


my_dict = {1: 'Python', 2: 'Java', 3: 'Ruby'}

print('my_dict[3]=', my_dict[3])
```

```python
print('keys=:',my_dict.keys()) #get keys
print('Values:',my_dict.values()) #get values
b=my_dict.items()
print('Key-value pairs:', b) #get key-value pairs
for e in b:
    print(e[0], e[1])

a=my_dict.pop(1)
print('Value:', a)
print('Dictionary:', my_dict)
b = my_dict.popitem() #pop the key-value pair
print('Key, value pair:', b)
print('Dictionary', my_dict)
my_dict.clear() #empty dictionary
print('n', my_dict)
```

5 Tuple (built-in)

- **Definition**: Tuples are the same as lists are with the exception that the data once entered into the tuple cannot be changed no matter what.
- **Exception**: when the data inside the tuple is mutable, only then the tuple data can be changed.
- **Examples:**

```python
'''
Data structure: tuple
'''
print('------------------------------tuple-------------------------')

mytuple=(1,2,3) #create tuple
print('mytuple=:',mytuple)

print('Show each element')
for e in mytuple:
    print(e)

print(mytuple[1])  # Get access one element

mytuple=mytuple+(5,6,7) # add elements
print('Add elements 5,6,7, mytuple=:',mytuple)
print(mytuple.count(2))# Frequency of 2
print(mytuple.index(5))   # Index of 2
```

6 Set (built-in)

- **Definition**: Sets are a collection of unordered elements that are unique.
- **Examples:**

```python
my_set = {1, 2, 3, 4, 5, 5, 5} #create set
print(my_set)
```

Output:

{1, 2, 3, 4, 5}

Operations in sets

The different operations on set such as union, intersection and so on are shown below.

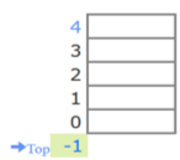
```python
my_set = {1, 2, 3, 4}
my_set_2 = {3, 4, 5, 6}
print(my_set.union(my_set_2), '----------', my_set | my_set_2)
print(my_set.intersection(my_set_2), '----------', my_set & my_set_2)
print(my_set.difference(my_set_2), '----------', my_set - my_set_2)
print(my_set.symmetric_difference(my_set_2), '----------', my_set ^ my_set_2)
my_set.clear()
print(my_set)
```

- The union() function combines the data present in both sets.
- The intersection() function finds the data present in both sets only.
- The difference() function deletes the data present in both and outputs data present only in the set passed.
- The symmetric_difference() does the same as the difference() function but outputs the data which is remaining in both sets.

7 Array (user-defined) vs list

- **List**: allow heterogeneous data element storage,
- **Array**: allow only homogenous elements to be stored within them.

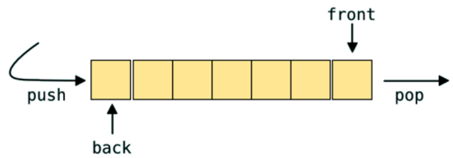
8 Stack (user-defined)



- **Definition**: linear Data Structures which are based on the principle of Last-In-First-Out (LIFO). It is built using the array structure and has operations namely, pushing (adding) elements, popping (deleting) elements and accessing elements only from one point in the stack called as the TOP.
- **Top**: the pointer to the current position of the stack.
- **Example**:

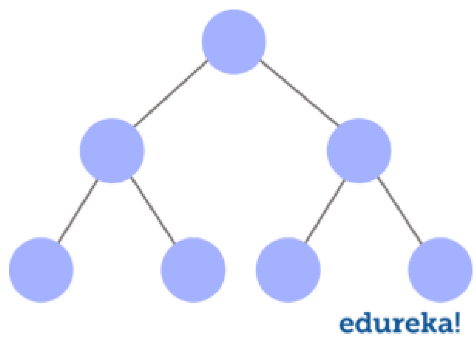
9 Queue (user-defined)

- **Definition**: a linear data structure that is based on the principle of First-In-First-Out (FIFO) where the data entered first will be accessed first. It is built using the array structure and has operations which can be performed from both ends of the Queue, that is, head-tail or front-back. Operations such as adding and deleting elements are called En-Queue and De-Queue and accessing the elements can be performed.



10 Tree (user-defined)

- **Definition**: Trees are non-linear Data Structures that have a root and nodes. The root is the node from where the data originates and the nodes are the other data points that are available to us.



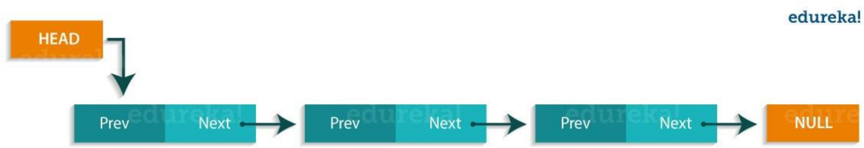
- **Tree**:
  In-order Traversal: left, root, right
  Pre-order Traversal: root, left, right
  Post-order Traversal: right, left, root
- **Example**:

```python
'''
Data structure:tree
'''
print('-----------------------------Tree-------------------------')

class node:
    def __init__(self, val=None):
        self.val=val
        self.left=None
        self.right=None


root=node(3)
print("root.val=", root.val)

List_tree=[3,2,4,5,6,7]

root=tree=node(List_tree[0])
root.left=node(List_tree[1])
root.right=node(List_tree[2])
print("tree")
print(tree)


def disp_tree(tree):
    if tree:
        print(tree.val)

    if tree.left:
        disp_tree(tree.left)

    if tree.right:
        disp_tree(tree.right)

print("print the tree")
disp_tree(root)
```

```
------------------------------Tree--------------------------
root.val= 3
tree
<__main__.node object at 0x000001D87834CFA0>
print the tree
3
2
4

In [38]:
```

11 Linked List (user-defined)

- **Definition**: Linked lists are linear Data Structures that are not stored consequently but are linked with each other using pointers. Each node consists of data and a pointer called next.



- **Example**:

```
'''
Data structure: LinkedList
'''
print('----------------------------Linkedlist-------------------------')


class Node:
   def __init__(self, val=None): #Pay attention to the
   #function __init__(self, val=None) with long "__"
       self.val=val
       self.next=None

#p=Node(None)
a=[1,2,3,4,5]

print("Input data a=:",a)

m=len(a)
header=S_link=Node(None)
#S_link.next=Node(1)
for i in range(m):
    S_link.next=Node(a[i])
    S_link=S_link.next

print("print the linkedlist")
header=header.next
while(header):
    print(header.val)
    header=header.next
```

```
-----------------------------Linkedlist--------------------------
Input data a=: [1, 2, 3, 4, 5]
print the linkedlist
1
2
3
4
5

In [35]:
```

12 Graph (user-defined)

- **Definition**: Graphs are used to store data collection of points called vertices (nodes) and edges (edges). Graphs can be called the most accurate representation of a real-world map. They are used to find the various cost-to-distance between the various data points called the nodes and hence find the least path.

References:

[1]. https://www.edureka.co/blog/data-structures-in-python/
[2]. https://www.geeksforgeeks.org/static-variables-in-c/
[3].