DSA IN PYTHON

1.Explain arrays in Pythons with example

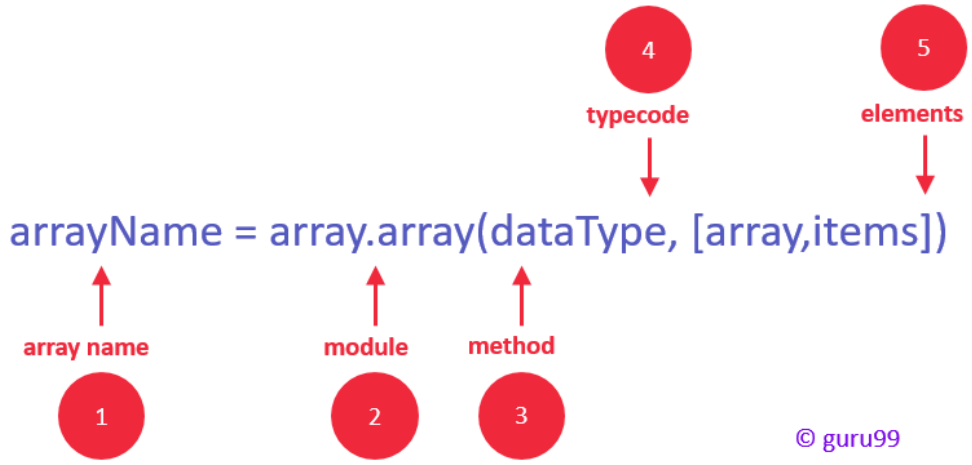
A Python Array is a collection of a common type of data structures having elements with the same data type. It is used to store collections of data. In Python programming, arrays are handled by the “array” module. If you create arrays using the array module, elements of the array must be of the same numeric type

Syntax to Create an Array in Python

You can declare an array in Python while initializing it using the following syntax.

```
arrayName = array.array(type code for data type, [array,items])
```

The following image explains the syntax.



1. **Identifier**: specify a name like usually, you do for variables
2. **Module**: Python has a special module for creating array in Python, called “array” – you must import it before using it
3. **Method**: the array module has a method for initializing the array. It takes two arguments, type code, and elements.
4. **Type Code**: specify the data type using the type codes available (see list below)

5. **Elements**: specify the array elements within the square brackets, for example [130,450,103]

```
import array as myarray
abc = myarray.array('d', [2.5, 4.9, 6.7])
```

2.How can you access array elements?

You can access any array item by using its index.

The syntax is

```
arrayName[indexNum]
```

Example

```
import array
balance = array.array('i', [300,200,100])
print(balance[1])
```

3.How can you insert elements in array?

Python array insert operation enables you to insert one or more items into an array at the beginning, end, or any given index of the array. This method expects two arguments index and value.

The syntax is

```
arrayName.insert(index, value)
```

Example

Let us add a new value right after the second item of the array. Currently, our balance array has three items: 300, 200, and 100. Consider the second array item with a value of 200 and index 1.

In order to insert the new value right “after” index 1, you need to reference index 2 in your insert method, as shown in the below Python array example:

```
import array
balance = array.array('i', [300,200,100])
balance.insert(2, 150)
print(balance)
```

4.How can you insert elements in array?

Python array insert operation enables you to insert one or more items into an array at the beginning, end, or any given index of the array. This method expects two arguments index and value.

The syntax is

```
arrayName.insert(index, value)
```

Example

Let us add a new value right after the second item of the array. Currently, our balance array has three items: 300, 200, and 100. Consider the second array item with a value of 200 and index 1.

In order to insert the new value right “after” index 1, you need to reference index 2 in your insert method, as shown in the below Python array example:

```
import array
balance = array.array('i', [300,200,100])
balance.insert(2, 150)
print(balance)
```

5.How can you delete elements in array?

With this operation, you can delete one item from an array by value. This method accepts only one argument, value. After running this method, the array items are re-arranged, and indices are re-assigned.

The syntax is

`arrayName.remove(value)`

Example

Let's remove the value of "3" from the array

```
import array as myarray
first = myarray.array('b', [2, 3, 4])
first.remove(3)
print(first)
```

6.How can you search and get the index of a value in an array?

With this operation, you can search for an item in an array based on its value. This method accepts only one argument, value. It is a non-destructive method, which means it does not affect the array values.

The syntax is

`arrayName.index(value)`

Example

Let's find the value of "3" in the array. This method returns the index of the searched value.

```
import array as myarray
number = myarray.array('b', [2, 3, 4, 5, 6])
print(number.index(3))
```

7.How can you reverse array in Python?

You can use reverse() to reverse array in Python.

Example:

```
import array as myarray
number = myarray.array('b', [1,2, 3])
number.reverse()
print(number)
```

8.Give example to convert array to Unicode

The Example to convert array to Unicode is:

```
from array import array
p =
array('u',[u'\u0050',u'\u0059',u'\u0054',u'\u0048',u'\
u004F',u'\u004E'])
print(p)
q = p.tounicode()
print(q)
```

Methods For Performing Operations on Arrays in Python

Arrays are mutable, which means they are changeable. You can change the value of the different items, add new ones, or remove any you don't want in your program anymore.

Let's see some of the most commonly used methods which are used for performing operations on arrays.

How to Change the Value of an Item in an Array

You can change the value of a specific element by speficying its position and assigning it a new value:

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
#change the first element
#change it from having a value of 10 to having a value of 40
numbers[0] = 40
print(numbers)
#output
#array('i', [40, 20, 30])
```

How to Add a New Value to an Array

To add one single value at the end of an array, use the append() method:

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
#add the integer 40 to the end of numbers
numbers.append(40)
print(numbers)
#output
#array('i', [10, 20, 30, 40])
```

Be aware that the new item you add needs to be the same data type as the rest of the items in the array.Look what happens when I try to add a float to an array of integers:

```
import array as arr

#original array
numbers = arr.array('i',[10,20,30])

#add the float 40.0 to the end of numbers
numbers.append(40.0)

print(numbers)

#output

#Traceback (most recent call last):
#  File "/Users/dionysialemonaki/python_articles/demo.py", line 19, in
<module>
#   numbers.append(40.0)
#TypeError: 'float' object cannot be interpreted as an integer
```


But what if you want to add more than one value to the end an array?

Use the `extend()` method, which takes an iterable (such as a list of items) as an argument. Again, make sure that the new items are all the same data type.

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
#add the integers 40,50,60 to the end of numbers
```

```
#The numbers need to be enclosed in square brackets
numbers.extend([40,50,60])
print(numbers)
#output
#array('i', [10, 20, 30, 40, 50, 60])
```

And what if you don't want to add an item to the end of an array? Use the `insert()` method, to add an item at a specific position.

The `insert()` function takes two arguments: the index number of the position the new element will be inserted, and the value of the new element.

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
#add the integer 40 in the first position
#remember indexing starts at 0
numbers.insert(0,40)
print(numbers)
#output
#array('i', [40, 10, 20, 30])
```

How to Remove a Value from an Array

To remove an element from an array, use the `remove()` method and include the value as an argument to the method.

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
numbers.remove(10)
print(numbers)
#output
#array('i', [20, 30])
```

With `remove()`, only the first instance of the value you pass as an argument will be removed.

See what happens when there are more than one identical values:

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30,10,20])
numbers.remove(10)
print(numbers)
#output
#array('i', [20, 30, 10, 20])
```

Only the first occurence of 10 is removed.
You can also use the pop() method, and specify the position of the element to be removed:

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30,10,20])
#remove the first instance of 10
numbers.pop(0)

print(numbers)
#output
#array('i', [20, 30, 10, 20])
```

Array operations
Some of the basic operations supported by an array are as follows:

- Traverse - It prints all the elements one by one.
- Insertion - It adds an element at the given index.
- Deletion - It deletes an element at the given index.
- Search - It searches an element using the given index or by the value.
- Update - It updates an element at the given index.

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

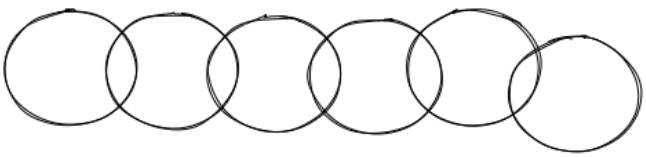
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Linked List

A linear data structure called a linked list has elements that are not kept in consecutive locations in memory. A linked list's elements are connected via pointers.

A pointer to the first node of the linked list serves as the representation of a linked list. The head refers to the top node. The head's value is NULL if the linked list is empty. In a list, each node has at least two components:

- **Data**
- **Pointer (Or Reference) to the next node**



A chain :)

Advantages of Linked Lists:

1. Because of the chain-like system of linked lists, you can add and remove elements quickly. This also doesn't require reorganizing the data structure unlike arrays or lists. Linear data structures are often easier to implement using linked lists.
2. Linked lists also don't require a fixed size or initial size due to their chainlike structure.

Disadvantages of a Linked Lists:

1. More memory is required when compared to an array. This is because you need a pointer (which takes up its own memory) to point you to the next element.
2. Search operations on a linked list are very slow. Unlike an array, you don't have the option of random access.

When Should You Use a Linked List?

You should use a linked list over an array when:

1. **You don't know how many items will be in the list (that is one of the advantages - ease of adding items).**
2. **You don't need random access to any elements (unlike an array, you cannot access an element at a particular index in a linked list).**
3. **You want to be able to insert items in the middle of the list.**
4. **You need constant time insertion/deletion from the list (unlike an array, you don't have to shift every other item in the list first).**

These are a few things you should consider before trying to implement a linked list.

Now with all the theory out of the way, it's time to implement one. We'll do this using Python, but most of what we learn here applies to any language you are using. The most important thing is to understand how it works.

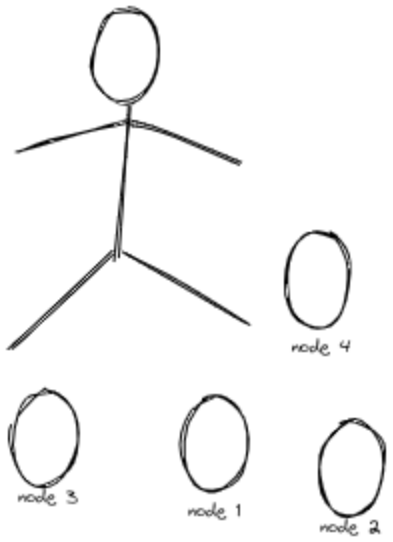
How to Use Linked Lists in Python

Here's a trick when creating a Linked List. It's something that helped me understand it much better.

You just have to realize that every item that you will be adding to the list is just a node (similar to a ring in a chain). What differentiates the head (which is the first node in the list) is that you gave it the title head, and then you started adding other nodes to it.

Remember that a Linked List is similar to how a chain is coupled together.

Joe is here with some rings, and he is going to help us.



I will be using this to illustrate as we go...so you can think along these lines (this is not an art class – I repeat, this is not an art class :)) .

So let's create the nodes first:

```
class Node:
    def __init__(self,value):
        self.value = value
        self.next = None
```


That is it. We add the value because for anything to be added to the linked list, it should at least have some value (for example, except in rare situations, you don't add an empty string to an array, right?).

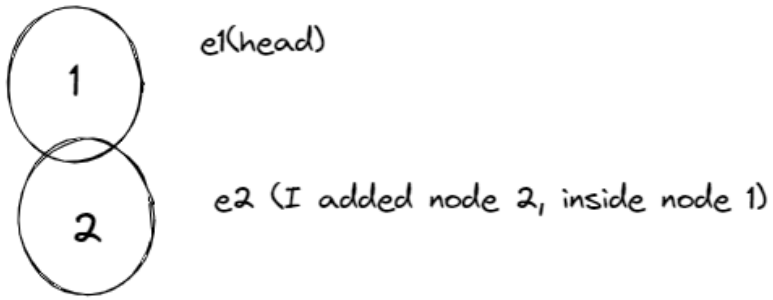
The next means that it is possible we want to chain other nodes – I mean, that is the major aim of a linked list.

Next we are going to define some basic functions:

```
class LinkedList:
    def __init__(self,head=None):
        self.head = head
```

```
def append(self, new_node):
    current = self.head
    if current:
        while current.next:
            current = current.next
        current.next = new_node
    else:
        self.head = new_node
```

The append() method lets you add a new node to the list. Let's explore how it works.



If I have two values – say 1 and 2 – and I want to add them to the list, the first thing is to define them as individual nodes (that is, as rings of a chain). I can do that like this:

```
e1 = Node(1)

e2 = Node(2)
```

I can now define a linked list since I have my nodes ready. A linked list (like the chains we see – always has a head, right?), so I can define my linked list with a head value which basically is just another node (ring):

```
ll = LinkedList(e1)
```

Now from the code above, e1 is the head of the linked list, which is just a fancy way of saying the starting point of my linked list. I can add more items to it, and since each chain has to be connected (that is, inside each other), I have to first set up the base case to check if the list has a head.

What makes a linked list is the fact that it has a starting point. If it doesn't, we simply need to set the new element to the head. But if it already has a head, I have to go through the whole list and keep checking to see if any of the nodes has a next that is empty (that is, None).

Again, a linked list is like a chain, right? So every node should point to another with the next pointer. Once a node has a next that is none, it simply means that it is the end of the list. So I can easily add the new node in that position.

Let's create a method to delete a node. But before we do, let's think about it for a second. Imagine you have a chain, and you find out a ring is weak. What do you do?

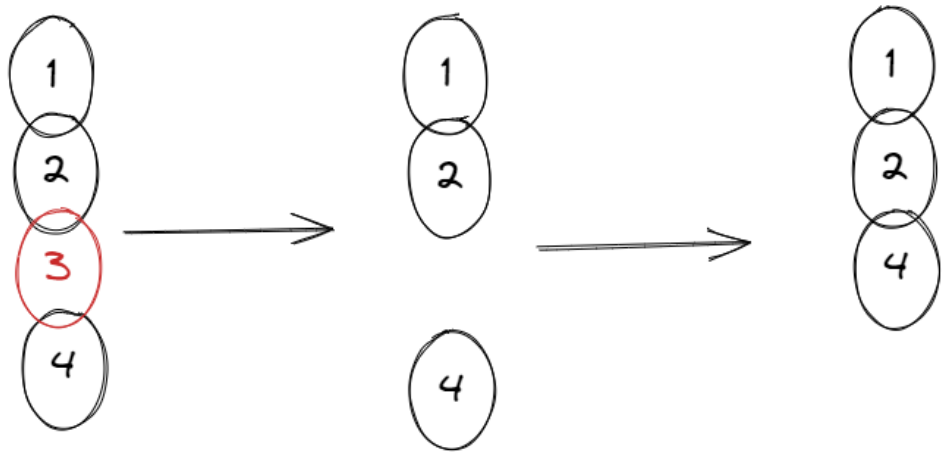
You first find the weak ring, then you remove it and connect the one before it and after it together. But if the weak ring is the first one, that is easy – you just remove it and you don't really have to join anything. The second ring automatically becomes the head of the chain. Try to visualize that.

We want to do the same thing here. So we first find the weak ring – in this case that will be the value we are looking for – and then we will take the one before and the one after and join them together:

```
class LinkedList:
    def __init(...)
    def append(...)
      def delete(self, value):
        """Delete the first node with a given value."""
        current = self.head
        if current.value == value:
            self.head = current.next
        else:
            while current:
                if current.value == value:
```

```
            break
        prev = current
        current = current.next
    if current == None:
        return
    prev.next = current.next
    current = None
```

So what we are doing here is simply going through each node to see if that is the value we want to remove. But as we move through the list, we have to keep track of the value before (we still have to join the list back together). We do this with prev = current as you can see above or below :).



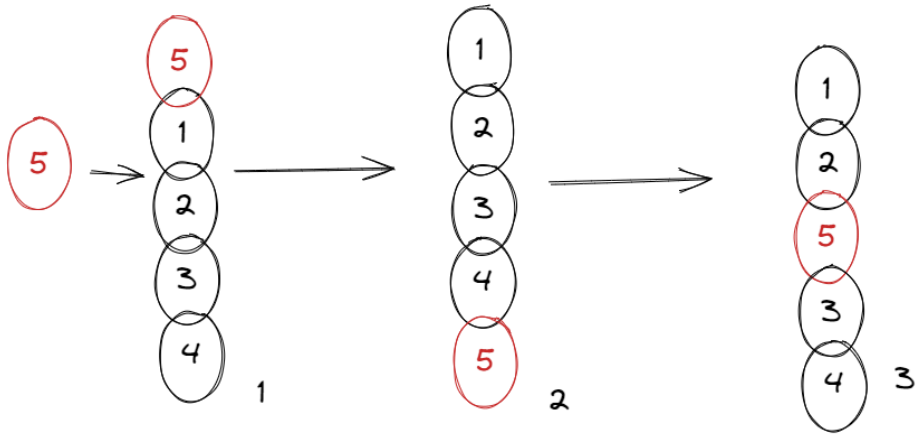
So when the node has been found, the prev which contains the node before it, can be easily switched (that is, the next value) to

point to another node – in this case the other nodes connected to the node we want to remove. I hope this makes sense :).

Let's work on inserting a node into a particular position. We will use our chain analogy to understand this better.

When you hold a chain, and you actually want to increase the length of the chain, you have three options. You can:

1. Add a link (element) to the beginning of the chain (this should be pretty simple, right?)
2. Add it to the end of the chain (kind of similar to 1)
3. Or you can add it at any point in the middle (a little trickier)



One thing you should have in mind is that wherever you decide to add it, you have to join the other nodes back to it. That's only possible if you keep track of the other nodes with a loop.

Let's see that in action:

```
class LinkedList:
 def __init(...)
 def append(...)
 def delete(...)
 def insert(self, new_element, position):
    """Insert a new node at the given position.
    Assume the first position is "1".
    Inserting at position 3 means between
    the 2nd and 3rd elements."""
    count=1
    current = self.head
    if position == 1:
       new_element.next = self.head
       self.head = new_element
    while current:
       if count+1 == position:
          new_element.next =current.next
          current.next = new_element
          return
```

```
        else:
            count+=1
            current = current.next
        # break

    pass
```

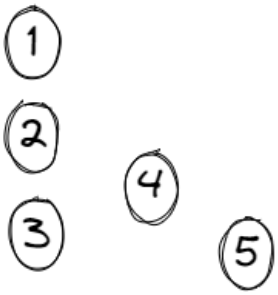
We are given a position to insert the node in the code above. If the position is one, it means it is going to be the root. Since we are not so sure, we can initialize a loop and a counter to keep track of the loop.

If the position we are to insert is one (that is, the root), simply store the current root in a dummy variable, create a new root, and then add the previous root (that is, the whole chain) to this new root.

If the position is not one, keep going through the chain until you find the position.

Finally for this article, let's work on displaying the values of our linked list in any format you want – for example, printing it out or adding it to a list collection. I will just be printing the values out.

This is pretty straightforward, similar to a physical chain: you just look through everywhere there is a node and get the value, then move to the next node:




```
class LinkedList:
    def __init(...)
    def append(...)
    def insert(...)
    def delete(...)
    def print(self):
        current = self.head
        while current:
            print(current.value)
            current = current.next
```

So that is all on linked lists for now! We will work on solving a few questions on linked lists later on.