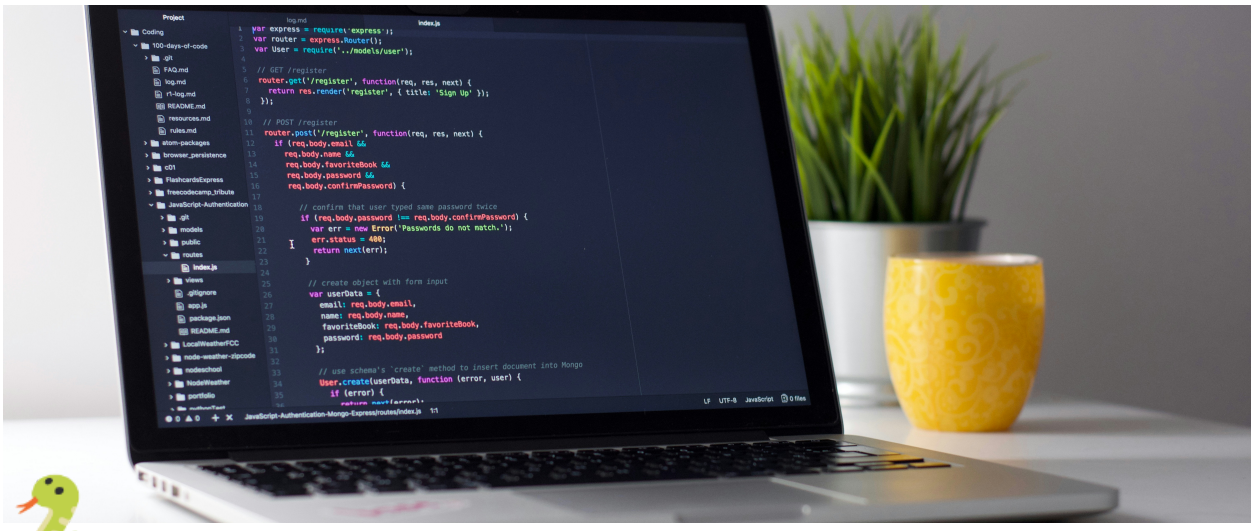


Python https://docs.python.org/3/

▼ LIST

List is a dynamically growing array.

After Deletion of list, it is still present in RAM, but the reference is removed.

> for i in range(len(l2)): #here i is acting as int DT
>
> print(l2[i])
> print("----------")

> for i in l2: #here i is acting as Auto(obj)
> print(i)

> We have Named references in python instead of pointers

💡 l1.pop(l1[1]) #it will take element in index value as count to pop
print(l1)

💡 **Errors:**

1.Value Error : when value x is not present

2.Type Error : '<' is not supported between instances of 'str' and 'float', but comparison is supported between in and float

3.Index Error : index out of range

l1.sort() #prints in ascending order
l1.sort(reverse = true) #prints in descending order

List object has no attribute "Size"

---

pointers in c:

A special variable that stores the address of another variable or points to another variable.

Pointer does not belong to any datatype, it is a void type.

Allocation of memory takes place from bottom to top.

📌 **size of the pointer is 4 bytes**

```
#include <stdio.h>
int main()
{
  int *ptr;
  int a =10;
  ptr = &a;
  printf("%d\n",*ptr);
  printf("address of a is=%u\n",&a);
```

```
   printf("address of ptr is=%u\n",&ptr);
   printf("%p\n",&*ptr);
   printf("contents of ptr=%u",ptr);

 }
```

💡 how name of an array itself is sufficient for Base-Address ?

converting Array to pointer:

Arr[] ———> *Arr———> Arr[]

💡 how do you represent an array in a pointer notation ?

—>arr[2]={1,2}
pf(“%d”,arr[0])-->pf(“%d”,*arr+0) 0-is index
pf(“%d”,arr[1])-->pf(“%d”,*arr+1) 1-is index- pointing to next address in the array

type of pointers:

1. **wild :** Initially it is not pointing to anything , when it is not assigned it can point to - anything—> int *ptr
2. **void :** if the pointer is made generic it is void pointer —> void *ptr =NULL;
3. **null :** int*ptr =NULL
4. **dangling ptr or situation ptr:** pointer is still pointing to the dereferenced address to avoid dangling pointer we need to declare the variable as static.

int *ptr;*
ptr = 10;
*print(”%d”,*ptr);*
#gives run time error as segmentation fault

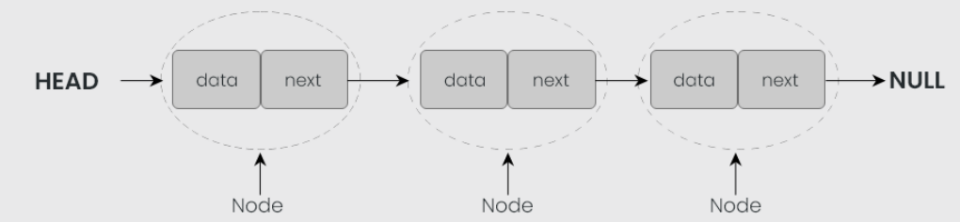
malloc: The `malloc` function is a standard library function in the C programming language used for dynamic memory allocation. It stands for "memory allocation," and its primary purpose is to allocate a specified number of bytes of memory during the program's execution. The `malloc` function is declared in the `stdlib.h` header file.

```
#include <stdio.h>
int main()
{
int
ptr =NULL;
ptr = (int* )malloc(3
*sizeof(datatype);
ptr = 9000;

*(ptr+0) =10;
*(ptr+1) =20;
*(ptr+2) =30;
}
```

Linked List:

A linked list is a data structure that consists of a sequence of nodes, each containing some data and a pointer to the next node in the list.

value	Address(self / Next node)



self referential pointer : a pointer ,pointing to its own-type

10	*(self)

```
struct node {
        int val;
        struct node *ptr; //self referenital pointer
}; //;--->it say that the whole set should be treated as one li
```

💡 for a self referential pointer the structure must be named (it should be declared)

💡 **“typedef”** used to define the name

📌 STEP 1 : create and assign values to the pointer.

pseudo code:

n1.val = 10;
n1.ptr= NULL;
n2.val = 20;
n2.ptr= NULL;
n3.val = 30;
n3.ptr= NULL;

STEP 2 : create the relationship.

pseudo code:

n1.ptr = &n2;
n2.ptr = &n3;

STEP 3 : Linked list is completed.

💡 **head** will be pointing to the base address

Arrow(→) is used to access the value in node

```
// linkedlist traversing
#include <stdio.h>
struct node {
    int val;
    struct node *ptr;
};
int main() {
    struct node n1, n2, n3;
    struct node *temp, *head;
     n1.val = 10;
     n1.ptr = NULL;
```

```
     n2.val = 20;
     n2.ptr = NULL;
     n3.val = 30;
     n3.ptr = NULL;

     n1.ptr = &n2;
     n2.ptr = &n3;

     head = &n1;
     temp = head;
     /*
     printf("values of  n1 is %d \n", temp->val);
     temp = temp->ptr;
     printf("values of  n2 is %d\n ", temp->val);
     temp = temp->ptr;
     printf("values of  n3 is %d ", temp->val);
     */
     while (temp){
         printf("%d->",temp->val);
         temp = temp->ptr;
     }
     printf("NULL");


    return 0;
}
```

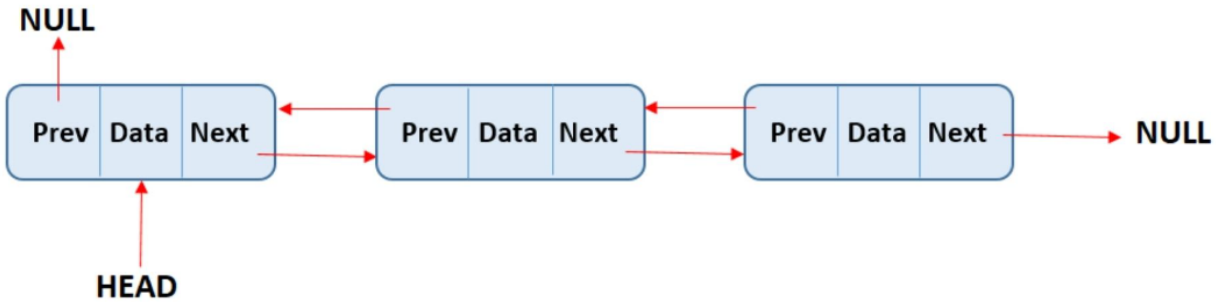
```
//LinkedList
#include <stdio.h>
#include <stdlib.h>
struct node
{
  int value;
  struct node *ptr;
};
```

```
typedef struct node NODE;
void dispList(NODE *);
int main()
{
  NODE* temp=NULL,*head = NULL;
  NODE *newNode;
  int choice = 1;
  while(choice){

  newNode = (NODE *)malloc(sizeof(NODE));
  //printf("\nEnter the New node value:");
  scanf("%d",&newNode->value);
  newNode->ptr = NULL;

  if(head == NULL){
    head = newNode;
    temp = head;
  }
  else
  {
    temp->ptr = newNode;
    temp = temp->ptr;
  }
 // printf("\nDo you want to create NN (0/1)");
  scanf("%d", &choice);
  }
  dispList(head);
  return 0;
}

void dispList(NODE *h){
    printf("\n list is \n");
    while (h){
        printf("->%d",h->value);
        h = h->ptr;
```

```
    }
}
```

Doubly-LinkedList

A doubly linked list is a type of linked list in which each node has two pointers: one to the next node and one to the previous node. This allows the list to be traversed in both directions and makes insertion and deletion operations more efficient. However, it also requires more memory and complexity than a singly linked list.




```
//doubly linked list
#include <stdio.h>
#include<stdlib.h>
typedef struct node {
        int val;
        struct node *right;
        struct node *left;
}NODE;


int main(){
    NODE n1, n2, n3,n4 ;
    NODE *head, *temp;

    n1.val = 10;
```

```
    n2.val = 20;
    n3.val = 30;
    n4.val = 40;


    n1.left = NULL;
    n1.right= &n2;
    n2.left= &n1;

    n2.right= &n4;
    n4.left =&n2;
    n4.right=&n3;
    n3.left =&n4;
    n3.right = NULL;
    head = &n1;
    temp = head;
     while (temp){
         printf("%d->",temp->val);
         temp = temp->right;

     }
     printf("NULL");
    return 0;

}
```

```
// node insertion
#include <stdio.h>
#include<stdlib.h>
typedef struct node{
    int val;
    struct node *prev;
    struct node *next;
}NODE;
NODE *createNode();
```

```
NODE*InsertNode();
void print(NODE*head);
void printreverse(NODE*temp);
int main() {
NODE *head=InsertNode();
print(head);
printreverse(head);
    return 0;
}
NODE *createNode(){
  return (NODE*)malloc(sizeof(NODE));
}

NODE *InsertNode(){
  NODE *temp=NULL,*head=NULL;
  int choice=1;
  while(choice){
    printf("enter data");
    NODE *n=createNode();
    scanf("%d",(&n->val));
    n->prev=NULL;
    n->next=NULL;
    if(head==NULL){
      head=n;
      temp=head;
    }
    else{
      n->prev=temp;
      temp->next=n;
      temp=temp->next;
    }
    printf("enter (0/1)");
    scanf("%d",&choice);

  }
  return head;
```

```
}
void print(NODE*head){
    while(head){
        printf("%d->",head->val);
        head=head->next;
    }
    printf("NULL");
}
void printreverse(NODE*temp){
    printf("\nNULL");
    while(temp->next!= NULL){
        temp=temp->next;
    }
    while(temp){
        printf("<-%d",temp->val);
        temp=temp->prev;
    }
}
```

Arrays:

In Python, arrays are not a built-in data type like in some other programming languages (e.g., C or Java). Instead, Python provides a powerful built-in data structure called lists, which can be used to implement arrays and offer additional features.

list can be included in array using→ .**fromlist()**

List Comprehensions

List comprehensions in python are concise, synatic constructs that can be used to generate a list by applying functions to each element in the list

Lambda (Inline/Anonymous)

lambda function creates an inline function that contains a single expression. the value of the expression is what when it is invoked.

```
greet_me = lambda: "hello"
print(greet_me())
str1 = "hello"
str3 = "Good After-noon"
greet_me = lambda str2, str3: str2+str3
print(greet_me(str1,str3))

powerMe = lambda x: x**3
for i in range(1,11):
  print(powerMe(i))
```

Zip

iterate over several iterables in parallel producing tuples with an item from each one.

```
l1 = [1,2,3,4]
str4 = ["Apple","ball","cat","dog"]
for item in zip(l1,str4):
  print(item)
```

Data Types in Python:

```
#using sys module

import sys
a =0
print(sys.getsizeof(a))
```

```
b =1
print(sys.getsizeof(b))
c = 12423456789023456 78
print(sys.getsizeof(c))
print(len(str(c)),type(c)) # converts int into str to give leng
```

```
#implement backspace in python
a= "paa#rr#ulll##"
b = []
for i in a :
    if i!='#':
        b.append(i)
    elif :
        b.pop()
print(''.join(b))
```

```
#reverse oky words
print(" ".join(input().split())[::-1]))
```

List	Tuple
Mutable	Immutable
occupy more space	occupy less space
slower	faster

```
#tuple operations
t = 10,20,30,10,20,30,40,50
print(t.count(20),t.index(50),sum(t), max(t), min(t), len(t))
```

Dictionary :

keys and values mapping

keys are immutable and values are mutable

dosen’t allow duplicate keys keys must be unique, the values can be duplicate

indexing is not possible with dictionaries, we can access the values using key

```
dic={1:"val1",2:"val2"}
dic.get(2)
dic.get(2,"default val")
dic.setdefault(4,"add value if key not found")
dic.update({key,val})#used to add values to the dictionary
```

```
#Dynamic input to the dictionary
dic={}
n=int(input("enter no of elements"))
for i in range(n):
    rollno=int(input("enter roll no"))
    name=input("enter name of the student")
    dic.update({rollno:name})
print(dic)
```

```
n=int(input("enter no of students"))
student={}
for i in range(n):
  rollno=int(input("enter roll no"))
  marks=list(map(int,input("enter marks of 3 subs").split())
)
  student.update({rollno:sum(marks)})
topper=max(zip(student.values(),student.keys()))
print("topper is ",topper[1],"marks are ",topper[0])
```

```
dic={}
n=input("enter string")
for i in n:
```

```
    dic.update({i:dic.setdefault(i,0)+1})
print(dic)
```

sorting the dictionary based on keys or values

```
import operator
d={10:100,20:200,30:300,40:75,50:13}
#sorted we can apply to any structure
#sorted will not modify the original list where as sort list
function which is for list
#sorted is better as compared to sort
import operator
d={10:100,20:200,30:300,40:75,50:13}
print(dict(sorted(d.items(),key=operator.itemgetter(0))))
print(dict(sorted(d.items(),key=operator.itemgetter(1))))
#0 represent keys
#1 reprsent values
```

```
import heapq
print(heapq.nlargest(2,d))#n largest keys
print(heapq.nsmallest(2,d))#n smallest keys
```

strings:

strings are immutable .directly we cannot modify the original string

Exception handling:

Exception handling in Python allows you to gracefully handle and manage errors that may occur during program execution. It helps prevent your program from crashing and allows you to handle different types of errors in different ways.

In Python, you can use `try`, `except`, `else`, and `finally` statements to handle exceptions. The `try` block contains the code that may raise an exception, and the `except` block is where you

handle the exception. The `else` block is executed if no exception occurs, and the `finally` block is always executed, regardless of whether an exception occurred or not.

Here is an example of exception handling in Python:

```
try:
    # Code that may raise an exception
    a = 10 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError:
    # Handling the ZeroDivisionError
    print("Cannot divide by zero!")
except:
    # Handling any other type of exception
    print("An error occurred!")
else:
    # Code to execute if no exception occurred
    print("No exception occurred!")
finally:
    # Code to always execute
    print("Finally block executed!")
```

You can handle specific types of exceptions by specifying the type in the `except` block, or you can handle multiple types of exceptions by using multiple `except` blocks.

Exception handling allows you to catch and handle errors, making your code more robust and resilient.

Types of Errors:

1. **Value Error:** This error occurs when a function receives an argument of the correct data type, but the value of the argument is invalid. For example, trying to convert a string that contains non-numeric characters to an integer.

```
# Example of Value Error
string_number = "abc"
try:
    number = int(string_number)
```

```
except ValueError:
    print("Invalid value!")
```

2. **Type Error:** This error occurs when an operation is performed on an object of an inappropriate type. For example, trying to compare a string and a float.

```
# Example of Type Error
string_value = "hello"
float_value = 3.14
try:
    result = string_value < float_value
except TypeError:
    print("Unsupported operation!")
```

3. **Index Error:** This error occurs when attempting to access an index that is outside the range of a sequence (e.g., a list or a string).

```
# Example of Index Error
my_list = [1, 2, 3]
try:
    value = my_list[5]
except IndexError:
    print("Index out of range!")
```

Handling Errors:

To handle errors, you can use try-except blocks. The code inside the try block is executed, and if an error occurs, it is caught and handled in the except block.

```
try:
    # Code that may raise an error
    # ...
except ErrorType:
    # Handling the error
    # ...
```

You can include multiple except blocks to handle different types of errors separately. Additionally, you can use the else block to specify code that should be executed if no exceptions occur, and the finally block to specify code that should always be executed, regardless of whether an exception occurred or not.

```
try:
    # Code that may raise an error
    # ...
except ErrorType1:
    # Handling ErrorType1
    # ...
except ErrorType2:
    # Handling ErrorType2
    # ...
else:
    # Code to execute if no exceptions occur
    # ...
finally:
    # Code to always execute
    # ...
```

Errors:

- NameError: Raised when a local or global name is not found.
- AttributeError: Raised when an attribute reference or assignment fails.
- KeyError: Raised when a key is not found in a dictionary.
- IndexError: Raised when an index is out of range.
- FileNotFoundError: Raised when a file or directory is not found.
- ZeroDivisionError: Raised when division or modulo by zero occurs.
- ImportError: Raised when an import statement fails.
- TypeError: Raised when an operation is performed on an object of an inappropriate type.
- SyntaxError: Raised when there is a syntax error in the code.

- ValueError: Raised when a function receives an argument of the correct data type, but the value of the argument is invalid.
- MemoryError: Raised when an operation runs out of memory.
- OverflowError: Raised when the result of an arithmetic operation is too large to be represented.
- IOError: Raised when an input/output operation fails.
- AssertionError: Raised when an assert statement fails.
- KeyboardInterrupt: Raised when the user interrupts the execution of a program (e.g., by pressing Ctrl+C).
- SystemError: Raised when the interpreter detects an internal error.
- FloatingPointError: Raised when a floating-point operation fails.

single try with multiple except:

```
try:
    # Code that may raise a
n error
    a = 10 / 0  # This will
raise a ZeroDivisionError
    b = "abc" + 123  # This
will raise a TypeError
except ZeroDivisionError:
    # Handling the ZeroDivi
sionError
    print("Cannot divide by
zero!")
except TypeError:
    # Handling the TypeErro
r
    print("Unsupported oper
ation!")
```

pointers.docx

```
try:
    # Code that may raise an error
    file = open("nonexistent_file.txt", "r")  # This will rai
se a FileNotFoundError
    my_list = [1, 2, 3]
    value = my_list[5]  # This will raise an IndexError
except FileNotFoundError:
    # Handling the FileNotFoundError
    print("File not found!")
except IndexError:
    # Handling the IndexError
    print("Index out of range!")
```

Set:

Pointers:

Lambda:

Map function:

Filter Function:

File-handling:

Class:

Encapsulation:

Abstraction:

Functions:

class rules:

user defined DS:

LinkedList(Python):

DoublyLL

sorting:

assignment

decorators

arithemetic operations:

TREES:

unix vs windows: