**PYTHON3 NOTES**

What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

Why Python?

* Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
* Python has a simple syntax similar to the English language.
* Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
* Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
* Python can be treated in a procedural way, an object-oriented way or a functional way.

Python Syntax compared to other programming languages

* Python was designed for readability, and has some similarities to the English language with influence from mathematics.
* Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
* Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Python Variables: In Python, variables are created when you assign a value to it. Ex: x = 5

y = "Hello, World!" , Python has no command for declaring a variable.

Print function :The print() function prints the specified message to the screen, or other standard output device. The message can be a string, or any other object, the object will be converted into a string before written to the screen.

Syntax:

print(object(s), sep=separator, end=end, file=file, flush=flush)

|  |  |
| --- | --- |
| object(s) | Any object, and as many as you like. Will be converted to string before printed |
| sep='separator' | Optional. Specify how to separate the objects, if there is more than one. Default is ' ' |
| end='end' | Optional. Specify what to print at the end. Default is '\n' (line feed) |

Examples:

print("hello world")

print(2 and 3)

print(2 or 3)

a=input()

print(a)

print(type(a))

print("abc","cdf","pdf",sep="")

print("abc","cdf","pdf",sep="\*")

print("abc","cdf","pdf")

print("abc","cdf","pdf",end="\n\n\n")

print("abc","cdf","pdf")

Output:

hello world

3

2

carrot

carrot

<class 'str'>

abccdfpdf

abc\*cdf\*pdf

abc cdf pdf

abc cdf pdf

abc cdf pdf

Comments lines:

* Single line :starts with #explain Python code.
* Multi line: starts and ends with """ """ or ''' ''',makes code more readable.

Python Operators : Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

* Arithmetic operators
* Assignment operators
* Comparison operators

Comparison operators are used to compare two values:

* Logical operators
* Identity operators
* Membership operators

|  |  |  |
| --- | --- | --- |
| **Operator** | **Name** | **Example** |
| == | Equal | x == y |
| != | Not equal | x != y |
| > | Greater than | x > y |
| < | Less than | x < y |
| >= | Greater than or equal to | x >= y |
| <= | Less than or equal to | x <= y |

* Bitwise operators

Arithmetic operators are used with numeric

values to perform common mathematical

operations:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Name** | **Example** |
| + | Addition | x + y |
| - | Subtraction | x - y |
| \* | Multiplication | x \* y |
| / | Division | x / y |
| % | Modulus | x % y |
| \*\* | Exponentiation | x \*\* y |
| // | Floor division | x // y |

The del keyword is used to delete objects. In Python everything is an object, so the del keyword can also be used to delete variables, lists, or parts of a list etc.

Example:

y = ["apple", "banana", "cherry"]

del y[0]

print(y)

x = "hello"

del x

print(x)

Assignment operators are used to assign values to variables:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Example** | **Same As** |
| = | x = 5 | x = 5 |
| += | x += 3 | x = x + 3 |
| -= | x -= 3 | x = x - 3 |
| \*= | x \*= 3 | x = x \* 3 |
| /= | x /= 3 | x = x / 3 |
| %= | x %= 3 | x = x % 3 |
| //= | x //= 3 | x = x // 3 |
| \*\*= | x \*\*= 3 | x = x \*\* 3 |
| &= | x &= 3 | x = x & 3 |
| |= | x |= 3 | x = x | 3 |
| ^= | x ^= 3 | x = x ^ 3 |
| >>= | x >>= 3 | x = x >> 3 |
| <<= | x <<= 3 | x = x << 3 |
| := | print(x := 3) | x = 3  print(x) |

Logical operators are used to combine conditional statements:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| and | Returns True if both statements are true | x < 5 and  x < 10 |
| or | Returns True if one of the statements is true | x < 5 or x < 4 |
| not | Reverse the result, returns False if the result is true | not(x < 5 and x < 10) |

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| is | Returns True if both variables are the same object | x is y |
| is not | Returns True if both variables are not the same object | x is not y |

Membership operators are used to test if a sequence is presented in an object:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| in | Returns True if a sequence with the specified value is present in the object | x in y |
| not in | Returns True if a sequence with the specified value is not present in the object | x not in y |

 Bitwise operators are used to compare (binary) numbers:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name** | **Description** | **Example** |
| & | AND | Sets each bit to 1 if both bits are 1 | x & y |
| | | OR | Sets each bit to 1 if one of two bits is 1 | x | y |
| ^ | XOR | Sets each bit to 1 if only one of two bits is 1 | x ^ y |
| ~ | NOT | Inverts all the bits | ~x |
| << | Zero fill left shift | Shift left by pushing zeros in from the right and let the leftmost bits fall off | x << 2 |
| >> | Signed right shift | Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off | x >> 2 |

 Operator precedence describes the order in which operations are performed.

If two operators have the same precedence, the expression is evaluated from left to right.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| () | Parentheses |
| \*\* | Exponentiation |
| +x  -x  ~x | Unary plus, unary minus, and bitwise NOT |
| \*  /  //  % | Multiplication, division, floor division, and modulus |
| +  - | Addition and subtraction |
| <<  >> | Bitwise left and right shifts |
| & | Bitwise AND |
| ^ | Bitwise XOR |
| | | Bitwise OR |
| ==  !=  >  >=  <  <=  is  is not  in  not in | Comparisons, identity, and membership operators |
| not | Logical NOT |
| and | AND |
| or | OR |

Built-in Data Types

In programming, data type is an important concept. Variables can store data of different types, and different types can do different things. Python has the following data types built-in by default, in these categories:

Return the type of these objects:

 a = ('apple', 'banana', 'cherry')

b = "Hello World"

c = 33

x = type(a)

y = type(b)

z = type(c)

print(x, y, z)

Output:

<class 'tuple'> <class 'str'> <class 'int'>

|  |  |
| --- | --- |
| Text Type: | str |
| Numeric Types: | int, float, complex |
| Sequence Types: | list, tuple, range |
| Mapping Type: | dict |
| Set Types: | set, frozenset |
| Boolean Type: | Bool(True or False) |
| Binary Types: | bytes, bytearray, memoryview |
| None Type: | NoneType |

Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total\_volume). Rules for Python variables:

* A variable name must start with a letter or the underscore character
* A variable name cannot start with a number
* A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and \_ )
* Variable names are case-sensitive (age, Age and AGE are three different variables)
* A variable name cannot be any of the python keywords.

Variable names with more than one word can be difficult to read.

There are several techniques you can use to make them more readable:

Camel Case : Each word, except the first, starts with a capital letter:

myVariableName = "John"

Pascal Case : Each word starts with a capital letter:

range(start,step,stop):returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

MyVariableName = "John"

Snake Case

Each word is separated by an underscore character:

my\_variable\_name = "John"

Assign to multiple variable :

x, y, z = "Orange", "Banana", "Cherry"

x = y = z = "Orange"

fruits = ["apple", "banana", "cherry"]

x, y, z = fruits

Global Variables

Variables that are created outside of a function (as in all of the examples above) are known as global variables. Global variables can be used by everyone, both inside of functions and outside.

Local Variables:

If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function.

INPUT: The input() function allows user input. Syntax:

input(prompt)

|  |  |
| --- | --- |
| prompt | A String, representing a default message before the input. |

Ex:

*Example:*

x = 1 # int

y = 2.8 # float

z = 1j # complex

🡪 convert from int to float:

a = float(x)

🡪 convert from float to int:

b = int(y)

🡪 convert from int to complex:

c = complex(x)

print(a)

print(b)

print(c)

print(type(a))

print(type(b))

print(type(c))

x = input('Enter your name:')

print('Hello, ' + x)

o/p:

Enter your name: kamala

Hello, kamala

 a=int(input()) #integer input

a=input() #always accepts a string

Indentation : Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Example:

def fun(n):

    print(n)

1.0

2

(1+0j)

<class 'float'>

<class 'int'>

<class 'complex'>

Conditional statements:

If statement:

Syntax:

if(cond:1):

#if cond is true then execute this part of code

If-else statement:

Syntax:

if(cond:1):

#if cond is true then execute this part of code

else:

#if cond is true then execute this part of code

If-else Ladder statement:

Syntax:

if(cond:1):

#if cond is true then execute this part of code

elif(cond:2):

#if cond is true then execute this part of code

elif(cond:3):

#if cond is true then execute this part of code

else:

#if cond is true then execute this part of code

The elif keyword is Python's way of saying "if the previous conditions were not true, then try this condition".

Nested if-else statement:

You can have if statements inside if statements, this is called nested if statements.

Syntax:

if(cond:1):

if(cond:2):

#if cond is true then execute this part of code

else :

#stmt2

else :

#stmt3

The pass Statement

if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

Looping statements:

FOR LOOP:

Python For loop is used for iterating over an iterable like a String, Tuple, List, Set, or Dictionary.

Syntax:

for var in iterable:  
 # statements

WHILE LOOP:

Based on the condition loops will iterate, if condition is false loop terminates.

Syntax:

Switch statement:

match var\_name:

Case 1:

#cond1

Case 2:

#cond2

Case 3:

#cond3

Case 4:

#cond4

.

.

Case \_:

#default case

while(condition):

#stmts

CONTROL STATEMENTS:

Continue:

it returns the control to the beginning of the loop.

 🡪 Prints all letters except 'e' and 's'

for letter in 'gold':

    if letter == 'e' or letter == 'd':

        continue

    print('Current Letter :', letter)

 Break:

it brings control out of the loop.

 for letter in 'letters':

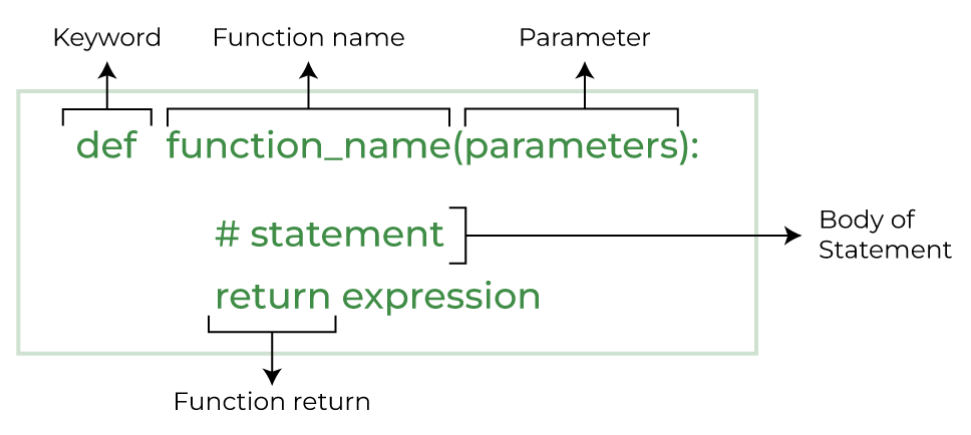
    # break the loop as soon it sees 'e' or 's'if letter == 'e' or letter == 's':

Import random

random.radint() and random.choice()

        Break

print('Current Letter :', letter)

FUNCTIONS: is a block of statements that return the specific task.

Types of Python Function Arguments

The main types of functions in Python are:

* Built-in function
* User-defined function
* Lambda functions
* Recursive functions
* Default argument
* Keyword arguments (named arguments)
* Positional arguments
* Arbitrary arguments (variable-length arguments
* \*args and \*\*kwargs)

Example:

def fun(a,b,c=0,d,e,\*f)->None:

print(a,b,c,d,e,f)

return

fun(2,3,d=5,2,3,5,4,6) #function call -> function\_name()

output: 2 3 0 5 2 (3,4,5,6)

Memory allocation:

Memory allocation in python , for variables is by call by reference .when we pass variable , a new reference object is created.

IMPORTANT:

To create virtual environment in python :🡪python -m venv any\_name

To activate the environment 🡪any\_name\Scripts\activate

If we get any errors , open windows powershell and type🡪Set-ExecutionPolicy Remotesigned

To install any module🡪pip install module\_name

To create a package🡪\_\_init\_\_.py

Ways to import a module and its specified functions:

🡪import module

🡪from module import function\_name

🡪from module import \*

🡪import from module\_name as another\_name #for ease of use

STRING : immutable , collections of characters . In python , by default input() takes string type input.

Indexing is used to access the elements of the string. Slicing is used to just accessing certain range of elements.

🡪For ex : string\_name[start : end index : step] to print entire string str\_name[::] , to reverse the string str\_name[::-1].

🡪To add element to a string , we use concatenation ( + ) , i.e. str\_1+str\_2=new\_str

String methods:

|  |  |
| --- | --- |
| capitalize() | Converts the first character of the string to a capital (uppercase) letter |
| count() | Returns the number of occurrences of a substring in the string. |
| endswith() | Returns “True” if a string ends with the given suffix |
| find() | Returns the lowest index of the substring if it is found |
| index() | Returns the position of the first occurrence of a substring in a string |
| isalnum() | Checks whether all the characters in a given string is alphanumeric or not |
| isalpha() | Returns “True” if all characters in the string are alphabets |
| isdecimal() | Returns true if all characters in a string are decimal |
| isdigit() | Returns “True” if all characters in the string are digits |
| isidentifier() | Check whether a string is a valid identifier or not |
| islower() | Checks if all characters in the string are lowercase |
| isnumeric() | Returns “True” if all characters in the string are numeric characters |
| isprintable() | Returns “True” if all characters in the string are printable or the string is empty |
| isspace() | Returns “True” if all characters in the string are whitespace characters |
| istitle() | Returns “True” if the string is a title cased string |
| isupper() | Checks if all characters in the string are uppercase |
| join() | Returns a concatenated String |
| lower() | Converts all uppercase characters in a string into lowercase |
| partition() | Splits the string at the first occurrence of the separator (tuple) |
| replace() | Replaces all occurrences of a substring with another substring |
| rfind() | Returns the highest index of the substring |
| rindex() | Returns the highest index of the substring inside the string |
| startswith() | Returns “True” if a string starts with the given prefix |
| split() | Returns the list of string at the occurrence of the separator |
| swapcase() | Converts all uppercase characters to lowercase and vice versa |
| title() | Convert string to title case |
| upper() | Converts all lowercase characters in a string into uppercase |

LISTS : mutable, collection of unordered data. Square braces are used to store elements in list . ex: l=[1,2,3]

How to take input for lists:

Method 1:

L=input().split()

for i in range(len(L)):

l[i]=int(l[i])

method 2: list comprehension

var\_name=[expression(element) for element in old\_list if condition ]

method 3:

var\_name=list(map(fun, input().split()))

ex: l=list(map(int, input().split()))

LAMBDA Function: lambda keyword is used .Also known as one line function. No need of fun calls.

Ex: lambda a : a\*\*2

Lists methods:

🡪to add element in the list ==l.append(ele)

🡪to mearge a list ==l.extend([list])

🡪to insert at specific index==l.insert(index,ele)

🡪To convert list to string : “”.join(list)

🡪to print list without braces ==print(\*list\_name)

🡪to delete list element : del l[i] , l.pop(i) returns deleted element. L.remove(value) returns none but deletes the value in the list .

🡪l.clear() : clears all elements but still empty list is present .

🡪del means it deletes entire list .

🡪sorting a list : l.sort() it returns None but sorts the list . sorted(list\_name) it returns a sorted list , does not change anything in the original list . l.sort(key= condition to sort , reverse if false :ascending order)

🡪to copy a list , b=list\_name.copy().

🡪to reverse a list : it returns None , but reverses the list l.reverse().

🡪l.count(val) : returns number of times the value has occurred .

🡪l.index(value) : returns the first index of value. If element not present raise ValueError.

TUPLE: immutable , collection of data . round braces are used to store elements in a tuple. Ex : a = (1,2,3,4)

T=tuple(map(int,input().split())) #to take input for tuples

🡪 tuples concatenation : t3=t2+t1

🡪nested tuples : t3=(t2,t1) , t3=((t2),(t1))

🡪repetition tuple: t3=(1,2)\*4, t3=(1,2,1,2,1,2,1,2)

🡪slicing tuples : tuples[start: stop : step] .

🡪deleting a tuple : del tuple\_name , deleting single element is not possible .

🡪length of tuples, lists, string : len(obj\_name).

🡪heterogenous , can store any datatype value.

🡪convert list to tuple : tuple(list\_name)

Set: A set is a collection of distinct elements where each element is unique within the set. Sets do not allow duplicate elements .Sets are unordered collections. Elements of a set must be immutable (cannot be changed after they are added to the set). Creating a set : my\_set = {1, 2, 3, 4, 5}. Adding elements to a set

my\_set.add(6) , Removing elements from a set : my\_set.remove(3) or set.pop(index) , Iterating through a set : for element in my\_set:

print(element)

Displaying the set : print(my\_set) , Output: {1, 2, 3, 4, 5, 6} , s=set(map(int,input().split()))) .

Set Intersection : Intersection of sets returns a new set containing only the elements that are common to both sets. Syntax: set\_intersection = set1.intersection(set2) , Example: Define two sets set1 = {1, 2, 3, 4, 5} set2 = {4, 5, 6, 7, 8}, Find the intersection , intersection\_set = set1.intersection(set2)

print(intersection\_set) Output: {4, 5} , a.intersection\_update(b): changes in a , return None . print(a&b)

Set Union

Union of sets returns a new set containing all unique elements from both sets.

Syntax: set\_union = set1.union(set2) , Define two sets : set1 = {1, 2, 3, 4, 5} , set2 = {4, 5, 6, 7, 8}

Find the union , union\_set = set1.union(set2) , print(union\_set) Output: {1, 2, 3, 4, 5, 6, 7, 8}

set.update(set2) : set is updated union of set . print(a/b) : union of sets.

Set Difference

Difference of sets returns a new set containing the elements that are present in the first set but not in the second set.Syntax : set\_difference = set1.difference(set2) , Define two sets : set1 = {1, 2, 3, 4, 5}

set2 = {4, 5, 6, 7, 8}

Find the difference (elements in set1 but not in set2) , difference\_set = set1.difference(set2)

print(difference\_set) Output: {1, 2, 3} , print(a-b)

Symmetric difference :  method returns a set that contains all items from both set, but not the items that are present in both sets. Ex : a.symmetric\_difference(b) , a.symmetric\_difference\_update(b) , print(a^b)

sub set and super set : print(a>=b) 🡪 a is super set of b , b is sub set of a . print(a>b) 🡪 a is proper superset of b , b is proper subset of a . a.issuperset(b) : returns bool , a.subset(b) : returns bool .

Disjoint function : a.isdisjoint(b) : returns bool

Dictionary : dictionary is a collection of key-value pairs, where each key is unique and associated with a value. Keys in a dictionary must be immutable (e.g., strings, numbers, tuples).Values in a dictionary can be of any data type. Dictionaries are unordered collections.

Creating a dictionary : my\_dict = {'key1': 'value1', 'key2': 'value2'}

Accessing elements of a dictionary : print(my\_dict['key1'])

Adding new key-value pairs ; my\_dict['key3'] = 'value3'

Iterating through a dictionary

for key, value in my\_dict.items():

print(key, value)

duplicate keys means modified value of key . a.get(key) : returns particular key , a.keys() returns lists of keys , a.values() : returns list of values . a.items() : returns list of tuples of key-value pair .

Example:

 class cse:

    def \_\_init\_\_(self,name,roll) -> None:

        self.n=name

        self.rn=roll

    def fun(self):

        print(self.n,s1.n)

s1=cse("aishu",7)

s2=cse("abc",9)

s1.fun()

print(s1.n,s2.n)

class circle:

    def printing(self,ra):

        self.R=r

        print(3.14\*self.R\*\*2)

class rectangle:

    def printing(self,le,br):

        self.L=le

        self.B=br

        print(self.B\*self.L)

l=int(input())

b=int(input())

r=int(input())

o=circle()

o1=rectangle()

o.printing(r)

o1.printing(l,b)

Oops: Class and Object

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields, and code, in the form of procedures. A class is a blueprint for creating objects (instances).An object is an instance of a class. Attributes are data stored inside a class or instance, and methods are functions associated with a class or instance.

Constructors

Definition: Constructors are special methods in Python classes that are automatically called when a new instance of the class is created.

Syntax: Defining a class

class MyClass:

def \_\_init\_\_(self, parameter1, parameter2):

self.parameter1 = parameter1

self.parameter2 = parameter2

def my\_method(self):

print("Hello from my\_method!")

# Creating an object of the class 🡪my\_object = MyClass(value1, value2)

# Accessing class attributes🡪print(my\_object.parameter1)

# Calling class methods🡪my\_object.my\_method()

Inheritance

Inheritance is a mechanism in object-oriented programming where a new class (subclass) is created by inheriting the properties and behaviors of an existing class (superclass). The subclass can then extend or modify the functionality of the superclass.

Syntax:

class SuperClass:

# Superclass attributes and methods

 class SubClass(SuperClass):

# Subclass inherits attributes and methods

Instance Variables

Definition: Instance variables are variables that are unique to each instance of a class. They are defined within methods of a class and are accessed using the `self` keyword.

Syntax:

class MyClass:

def \_\_init\_\_(self, parameter1, parameter2):

self.instance\_var1 = parameter1

self.instance\_var2 = parameter2

Example:

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

🡪Creating objects of the class

person1 = Person("Alice", 30)

person2 = Person("Bob", 25)

print(person1.name) 🡪 Output: Alice

print(person2.age) 🡪 Output: 25

Static Variables

Definition: Static variables (class variables) are shared among all instances of a class. They are defined within the class but outside any method using the class name.

Syntax:

class MyClass:

static\_var = value

Local Variables

Definition: Local variables are variables that are defined within a function or method and can only be accessed within that function or method.Example:

def my\_function():

local\_var = 10

print(local\_var)

my\_function() 🡪 Output: 10

Trying to access local\_var outside the function will result in an error

print(local\_var) 🡪NameError: name 'local\_var' is not defined

Example:

class a:

    def \_\_init\_\_(self) -> None:

        pass

    def fun1(self):

        print("fun1")

        #self.fun1() recurrsion

    def fun2(self):

        print("fun2")

class b(a):

    def fun3(self):

        print("fun3")

        self.fun2()

    def fun4(self):

        print("fun4")

class c(b,a):

    def fun5(self):

        print("fun5")

    def fun6(self):

        print("fun6")

class d(b):

    def fun7(self):

        print("fun7")

    def fun8(self):

        print("fun8")

p=a()

q=b()

r=c()

s=d()

p.fun1()

p.fun2()

q.fun1()

q.fun2()

q.fun3()

q.fun4()

Example: static variable

class Circle:

pi = 3.14

def \_\_init\_\_(self, radius):

self.radius = radius

def area(self):

return Circle.pi \* self.radius \* self.radius

🡪 Creating objects of the class

circle1 = Circle(5)

circle2 = Circle(7)

print(circle1.area()) 🡪 Output: 78.5

print(circle2.area()) 🡪 Output: 153.86

LINKED LISTS: linear data structure , nodes and links fields .

Types:

Singly linked list.

Doubly linked list.

Circular singly ll.

Circular doubly ll

Example:

def check\_age(age):

    if age < 0:

        raise ValueError("Age cannot be negative")

    elif age < 18:

        raise ValueError("You must be at least 18 years old")

    else:

        print("Welcome! You are eligible.")

try:

    user\_age = int(input("Enter your age: "))

    check\_age(user\_age)

except ValueError :

    print("hello")

def fun(a:int,b:int):

    try:

        s=a/b

    except ValueError:

        print("value mistake")

a=input()

b=input()

try:

    fun(a,b)

except TypeError:

    print("only int can be accepted")

finally:

    print("done!")

Exception Handling : Exception handling is the process of responding to and managing exceptions (errors) that occur during the execution of a program. Syntax:

try:

# Code that may raise an exception

pass

except ExceptionType:

# Code to handle the exception

pass

else:

# Code to execute if no exception occurs

pass

finally:

# Code that always executes, regardless of whether an exception occurred

pass

How to create a node?:

class node:

    def \_\_init\_\_(self,data) -> None: data link == NODE

        self.data=data

        self.next=None

SINGLY LINKED LISTS

class Node:

    def \_\_init\_\_(self, data=None):

        self.data = data

        self.next = None

class LinkedList:

    def \_\_init\_\_(self):

        self.head = None

def insert\_at\_end(self, data):

        new\_node = Node(data)

        if not self.head:

            self.head = new\_node

        else:

            current = self.head

            while current.next:

                current = current.next

            current.next = new\_node

   def insert\_at\_front(self, data):

        new\_node = Node(data)

        new\_node.next = self.head

        self.head = new\_node

def print\_list(self):

        count = 0

        if not self.head:

            print("List is empty")

        else:

            current = self.head

            while current:

                print(current.data , end=””)

                count += 1

                current = current.next

            print("Number of nodes:", count)

def delete\_end(self):

        if not self.head:

            return

        elif not self.head.next:

            self.head = None

        else:

            current = self.head

            prev = None

            while current.next:

                prev = current

                current = current.next

            prev.next = None

def delete\_front(self):

        if not self.head:

            return

        elif not self.head.next:

            self.head = None

        else:

            self.head = self.head.next

def reverse(self):

        prev = None

        current = self.head

        while current:

            next\_node = current.next

            current.next = prev

            prev = current

            current = next\_node

        self.head = prev

l = [1, 2, 3, 4, 5]

linked\_list = LinkedList()

for item in l:

    linked\_list.insert\_at\_end(item)

    linked\_list.insert\_at\_front(item)

print("Elements inserted at end and front:")

linked\_list.print\_list()

linked\_list.delete\_end()

print("After deleting end:")

linked\_list.print\_list()

linked\_list.delete\_front()

print("After deleting front:")

linked\_list.print\_list()

print("After reversing:")

linked\_list.reverse()

linked\_list.print\_list()

output:

Elements inserted at end and front:

5 4 3 2 1 1 2 3 4 5

Number of nodes: 10

After deleting end:

5 4 3 2 1 1 2 3 4

Number of nodes: 9

After deleting front:

4 3 2 1 1 2 3 4

Number of nodes: 8

After reversing:

4 3 2 1 1 2 3 4

Number of nodes: 8

def delete\_end(self):

        if self.head is None:

            print("List is empty")

            return

        if self.head == self.tail:

            self.head = None

            self.tail = None

        else:

            self.tail = self.tail.prev

            self.tail.next = None

    def reverse(self):

        if self.head is None:

            print("List is empty")

            return

        current = self.head

        while current:

            current.prev, current.next = current.next, current.prev

            current = current.prev

        self.head, self.tail = self.tail, self.head

data = [1, 2, 3, 4, 5]

dll = DoublyLinkedList()

for item in data:

    dll.insert\_end(item)

print("Initial list:")

dll.print\_list()

dll.delete\_front()

print("List after deleting from the front:")

dll.print\_list()

dll.delete\_end()

print("List after deleting from the end:")

dll.print\_list()

dll.reverse()

print("Reversed list:")

dll.print\_list()

Initial list:

1 2 3 4 5

List after deleting from the front:

2 3 4 5

List after deleting from the end:

2 3 4

Reversed list:

4 3 2

#swap a,b=b,a

#curr.next,curr.prev=curr.prev,curr.next

#curr=curr.prev

class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

        self.prev = None

class DoublyLinkedList:

    def \_\_init\_\_(self):

        self.head = None

        self.tail = None

    def insert\_front(self, data):

        new\_node = Node(data)

        if self.head is None:

            self.head = new\_node

            self.tail = new\_node

        else:

            new\_node.next = self.head

            self.head.prev = new\_node

            self.head = new\_node

    def insert\_end(self, data):

        new\_node = Node(data)

        if self.head is None:

            self.head = new\_node

            self.tail = new\_node

        else:

            new\_node.prev = self.tail

            self.tail.next = new\_node

            self.tail = new\_node

    def print\_list(self):

        if self.head is None:

            print("List is empty")

            return

        current = self.head

        while current:

            print(current.data)

            current = current.next

 def delete\_front(self):

        if self.head is None:

            print("List is empty")

            return

        if self.head.next is None:

            self.head = None

            self.tail = None

        else:

            self.head = self.head.next

            self.head.prev = None

    def delete\_end(self):

        if self.head is None:

            print("List is empty")

            return

        if self.head == self.tail:

            self.head = None

            self.tail = None

        else:

            self.tail = self.tail.prev

            self.tail.next = None

 def DFrnt(self):

        if self.head == None:

            return

        elif self.head.next == None:

            self.head = None

        else:

            self.head = self.head.next

    def rev(self):

        if self.head == None:

            print("list is empty")

            return

        p = None

        curr = self.head

        while curr:

            nxt = curr.next

            curr.next = p

            p = curr

            curr = nxt

            if curr == self.head:

                break

        self.head = p

l = [1, 2, 3, 4, 5]

p = a()

for i in range(len(l)):

    p.IEnd(l[i])

    p.IFrnt(l[i])

print("Original list:")

p.printing()

p.DEnd()

print("List after deleting from end:")

p.printing()

p.DFrnt()

print("List after deleting from front:")

p.printing()

print("Reversed list:")

p.rev()

p.printing()

Original list:

5 4 3 2 1 1 2 3 4 5

List after deleting from end:

5 4 3 2 1 1 2 3 4

List after deleting from front:

4 3 2 1 1 2 3 4

Reversed list:

4 3 2 1 1 2 3 4

Original list:

5 4 3 2 1 1 2 3 4 5 List after deleting from end:

5 4 3 2 1 1 2 3 4 List after deleting from front:

4 3 2 1 1 2 3 4 Reversed list:

4 3 2 1 1 2 3 4

CIRCULAR SINGLY LINKED LISTS:

class node:

    def \_\_init\_\_(self, data) -> None:

        self.data = data

        self.next = None

class a:

    def \_\_init\_\_(self) -> None:

        self.head = None

    def IEnd(self, data):

        new = node(data)

        if self.head == None:

            self.head = new

        else:

            i = self.head

            while i.next:

                i = i.next

            i.next = new

    def IFrnt(self, data):

        new = node(data)

        if self.head == None:

            self.head = new

        else:

            new.next = self.head

            self.head = new

  def printing(self):

        if self.head == None:

            print("list is empty")

            return

        i = self.head

        while i:

            print(i.data,end=" ")

            i = i.next

            if i == self.head:

                break

 def DEnd(self):

        if self.head == None:

            return

        elif self.head.next == None:

            self.head = None

        else:

            i = self.head

            prev = None

            while i.next:

                prev = i

                i = i.next

            prev.next = None

CIRCULAR DOUBLY LINKED LISTS:

class node:

    def \_\_init\_\_(self,data) -> None:

        self.data=data

        self.next=None

        self.prev=None

class dll:

    def \_\_init\_\_(self) -> None:

        self.head=None

        self.tail=None

    def IFrnt(self, data):

        new = node(data)

        if self.head==None:

            self.head=new

            self.tail=self.head

        else:

            new.next=self.head

            self.head.prev=new

            self.head=new

            self.head.prev=self.tail

            self.tail.next=self.head

    def IEnd(self,data):

        new=node(data)

        if self.head==None:

            self.head=new

            self.tail=self.head

        else:

            self.tail.next=new

            new.prev=self.tail

            self.tail=new

            self.head.prev=self.tail

            self.tail.next=self.head

    def Print(self):

        c=0

        if self.head==None:

            print("list is empty")

            return

        i=self.head

        while i!=self.tail:

            print(i.data)

            c=c+1

            i=i.next

        print(self.tail.data)

        print("number of nodes:",c+1)

def DEnd(self):

if self.head == None:

print("list is empty")

elif self.head == self.tail:

self.head = None

self.tail = None

else:

self.tail = self.tail.prev

self.tail.next = None

    def DFrnt(self):

        if self.head==None:

            print("list is empty")

            return

        elif self.head.next==self.tail:

            self.head=None

            self.tail=None

        else:

            self.head=self.head.next

            self.head.prev=self.tail

def rev(self):

        if self.head!=None:

            p=None

            curr=self.head

            n=self.head.next

            while curr.next:

                curr.next=p

                p=curr

                curr.prev=p

                curr=n

                if n:

                    n=n.next

            self.head=p

        else:

            print("list is empty")

l=[1,2,3,4,5]

p=dll()

for i in range(len(l)):

    p.IEnd(l[i])

    #p.IFrnt(l[i])

p.Print()

p.DFrnt()

p.Print()

p.DEnd()

p.Print()

"""p.rev()

p.Print()"""

#swap a,b=b,a

#curr.next,curr.prev=curr.prev,curr.next

#curr=curr.prev

#create random first obj if , we want to delete first element only

"""c.next=c.next.next

    to delete element in between"""

"""new.next=c.next

   c.next=new

   to insert any element in between"""

RECURRSION:

#binary search

l=[1,2,3,4,5,6]

se=4

si=0

li=len(l)-1

while si<=li:

    mid=(si+li)//2

    if l[mid]==se:

        print("ele found",l[mid])

        break

    elif l[mid]<se:

        si=mid+1

    else:

        li=mid-1

else:

    print("ele not found")

ele found 4

#stack implementation

l=[1,2,3,4,5]

l.append(9)

print(l)

l.pop()

print(l)

[1, 2, 3, 4, 5, 9]

[1, 2, 3, 4, 5]

a function calls itself again and again .

#FIBONACCI SERIES

def fun(n):

    if n==0 or n==1:

        return n

    return fun(n-1)+fun(n-2)

print(fun(5))

#FACTORIAL

def fun(n):

    if n==0:

        return 1

    return n\*fun(n-1)

print(fun(5))

output:

5

120

Trees: Hirachical

#binary tree representation

class node:

    def \_\_init\_\_(self,data) -> None:

        self.data=data

        self.r=None

        self.l=None

root=node(1)

root.l=node(2)

root.r=node(3)

    print(root.data)

print("preorder\_traversal")

dfs\_preorder(root)

print("inorder\_traversal")

dfs\_inorder(root)

print("postorder\_traversal")

dfs\_postorder(root)

Output:

preorder\_traversal

1

2

4

5

3

inorder\_traversal

4

2

5

1

3

postorder\_traversal

4

5

2

3

1

root.l.l=node(4)

root.l.r=node(5)

"""print(root.data)

print(root.l.data)

print(root.r.data)

print(root.l.l.data)

print(root.l.r.data)"""

#dfs(deapth first search) traversal

def dfs\_preorder(root):

    if root==None:

        return

    print(root.data)

    dfs\_preorder(root.l)

    dfs\_preorder(root.r)

def dfs\_inorder(root):

    if root==None:

        return

    dfs\_inorder(root.l)

    print(root.data)

    dfs\_inorder(root.r)

def dfs\_postorder(root):

    if root==None:

        return

    dfs\_postorder(root.l)

    dfs\_postorder(root.r)

Implementation of Queue:

class Queue:

def \_\_init\_\_(self):

self.items = []

def enqueue(self, item):

self.items.append(item)

def dequeue(self):

if self.is\_empty():

raise IndexError("Cannot dequeue from an empty queue")

return self.items.pop(0)

def is\_empty(self):

return len(self.items) == 0

def size(self):

return len(self.items)

def peek(self):

if self.is\_empty():

raise IndexError("Cannot peek into an empty queue")

return self.items[0]

# Example usage:

q = Queue()

q.enqueue(1)

q.enqueue(2)

q.enqueue(3)

print("Queue size:", q.size())

print("Front of the queue:", q.peek())

print("Dequeue:", q.dequeue())

print("Queue size after dequeue:", q.size())