**DICTIONARY**

mutable

no multiple keys

unordered

del

clear

pop

*METHODS:*

-------

-clear()

empty dictionary

-copy()

copy dict to another variable

-fromkeys()

create same value for multiple keys

-get()

return value if available by key else

-items()

it will return both keys and values

-keys()

it will return only keys

-pop

delete item using key

-popitem

random delete items

-setdefault()

used to add a key to a dictionary

-update()

used to update dictionaries

-values()

it will return only values

*NESTED DICT:*

-------------------

dict1={1:{'a':'A'},2:{'b':'B'}}

dict[1]['a']

=>'A'

**USER DEFINED FUNCTIONS**

*required argument*

-------------------------

Require same no of parameters in fn call and fn def

def function(a,b):

#code

function(1,2)

keyword argument

--------------------------

Pass value assigning to a variable

def function(a,b):

#code

function(b=1,a=3)

default argument

--------------------------

Assigns avalue if not given

def function(a,b=3):  
 #code

function(a=1,b)

variable length

----------------------

def function(\*a):  
 #code

function(a,b,c,d)

**ANONYMOUS OR NAMELESS FNTN**

*LAMBDA()*

------------

Lambda is a higher order function

f=(lambda arguments(1,2,3...):exp)

f(<value>)

*MAP()*

------

map(fn obj,iteratable)

a=map(lambda x:x\*2,[1,2,3,4])

tuple(a)

>>> (2, 4, 6, 8)

*FILTER()*

---------

filter(fn obj, iteratable value)

filter fn can have only one iteratable as input

i=[1,2,3,4,5,6,7,8,9]

f=filter(lambda x:2>x>5 ==0,i)

list(f)

>>> [2, 4, 6, 8]

*REDUCE()*

------------

It is a fn whch performs sme computation on a list

reduce(fn obj,iteratable)

l=[1,2,3,4]

functools.reduce(lambda a,b:a+b,l)

=>10

*GENERATOR()*

-----------------

A simple way of creatiing itertetors & it returns obj whch can be iterated ovr anfd ovr

**COLLECTIONS**

Collections in python are containers that r used to store collections of data

Secialized collections data types

\*Counter

\*deque

\*named\_tuple

\*chain\_map

\*Ordered\_Dict

\*defaultdict

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

l1=[]

for i in l:

c=0

for j in l:

if(i==j):

c+=1

l1.append(c)

*COUNTER*

------------

It returns values as dictionary

So it is unordered

from collections import Counter as c

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

d=c(l)

d=>counter({1:2,2:1,3:2})

Functions in counter:

\*most\_common()

\*elements()

\*subtracts()

*most\_common()*

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

d=counter(l).most\_common()

=>[(1,3),(2,2),(3,1),(4,1)]

*elements()*

a='aacbabd'

d=Counter(d).elements()

=>['a','a','a','a','c','b','b','d']

sorted(d)=>returns values sorted based on ASCII values

*subtract()*

Subtract fn subtracts two counter dictionaries and stores in one of its dict

x=Counter({'a'=5,'b'=2})

y=Counter({'a'=2,'b'=3})

x.subtract(y)

{'a'=3,'b'=-1}

*DEFAULTDICT*

------------

from collections import defaultdict

d=defaultdict(int)

d['apple']=3

d=>defaultdict(<class 'int'>, {'a': 1})

d['c']

=>0

d=>defaultdict(<class 'int'>, {'a': 1, 'c': 0})

*ORDEREDDICT*

------------

maintains order of dict as initialized

from collections import OrderedDict

d={'a':1,'b':2,'c':0}

OrderedDict(d)

OrderedDict([('a', 1), ('b', 2), ('c', 0)])

*NAMED TUPLES*

-------------

In tipical tuples access only the value through index

we cannot give names to individual elements to a tuple but can by named tuple

from collections import namedtuple

*Data*=namedtuple('*Data*','Name Age Ph')

Raghul=Data('Raghul','20','9876543210')

Raghu l=> Data(Name='Raghul', Age='20', Ph='9876543210')

Raghul.Age => '20'

*CHAIN MAP*

---------

Used to combine several dictionaries

It returns a list of dictiionaries

from collections import ChainMap

d1={'a':1,'b':2}

d2={'c':3,'d':4}

ChainMap(d1,d2)

>>> ChainMap({'a': 1, 'b': 2}, {'c': 3, 'd': 4})

ChainMap(d1,d2).maps

>>> [{'a': 1, 'b': 2}, {'c': 3, 'd': 4}]

ChainMap(d1,d2)['a']

>>> 1

list(d.values())

>>> [3, 4, 1, 2]

*Adding new dict to chainMap*

To add new dict to an existing chainmap we use new\_child()

adds at the beginning of chainmap

d3={'e':5}

d.new\_child(d3)

>>> ChainMap({'e': 5}, {'a': 1, 'b': 2}, {'c': 3, 'd': 4})

*DEQUE*

Deque keyword is used to make a list as queue

It is used to add and remove elements on boths ends

\*insert elements in deque

append()  
\*add one value from right side

appendleft()

\*add one value from left side

appendright()

\*add more than one value from left

extendleft()

\*add more than one value from right

Extendright()

OOPs(Object Oriented Programming)

4 PILLARS OF OOPS

\*Encapsulation

-Bundling data and methods

\*Polymorphism

-poly(many) morph(shape)

-single function works for different inputs

\*Inheritance

-Eliminate redundancy

-'is a relation' btwn classes

\*Abstraction

-hiding data

OOPS VS PROCEDURAL

OOPs

\*Bottom-top approach

\*Divided into objects

\*Access Specifiers

\*Overloading and overriding

\*More secure

\*Communication btwn functions

Procedural

\*Top down approach

\*No access specifuers

\*No communication btwn funcitions

\*Less secure

\*No overloading and overriding

Object

\*Real world entity

\*Instance of class

\*Object Representation

\*attributes(appearance and properties)

\*methods(behaviour and characteristics)

Classes

\*Describes object

\*Blueprint of object

EXAMPLES:

Class

-Chocolate

Object

-Dairy milk

Attributes

-color

-size

-price

Methods

-sweet

Class

-Machine

Object

-Laptop

Attributes

-weight

-color

-storage

-price

Methods

-on

-off

-calculate

SYNTAX

class classname:  
 methods&attributes

class student:  
 pass

obj=student() #instantiate a class

# Creating an object of a class

* Memory will be created only whn object is created to a class

#CODE

class student:  
 pass

obj=student()

print(obj)

#OUTPUT

=>address of object of that class

UML diagram is a graphical/pictorial representaion of a class





Can be accessed with object name

class student:

name='Raghul'

age=21

def study(self):

print('studying')

obj=student()

obj.study()

print(obj.age)

#OUTPUT

studying

21

class Student:

def \_\_init\_\_(s,name,rollno,dept,cgpa):

s.name=name

s.rollname=rollno

s.dept=dept

s.cgpa=cgpa

def hobby(s):  
 print(s.name,'is playing')

rag=Student('Raghul','17cse68','cse',7.5)

kis=Student('Kishore','17cse03','cse',6.0)

rag.hobby()

kis.hobby()

CONSTRUCTOR

It is a spl type of method which gets invoked automatically whn object is created to a class

classs variable

it is a variable that is shared by all the data members of a class

it is defined within a class but outside any of the class methods

instance variable

a variable that is defined inside a method and belong only to the current instance of a class

Code to change the class variable outside the class

class Dog:  
 breed='lab'

obj=Dog()

obj1=Dog()

obj.breed='pitbull'

obj1.breed='Syberian Husky'

print(obj.breed())

we can call the instance variable only with the help of obj name and not with class name

Empty parameter constructor- Non parameterized constructor

class Student:

def \_\_init\_\_(s):

s.name=input()

s.rollname='17cse68'

s.dept='cse'

s.cgpa=7.8

def hobby(s):

print(s.name,'is playing')

rag=Student()

rag.hobby()

getattr()

Get attribute

returns value of an attribute

syntax

getattr(obj,attribute\_name)

setattr()

It is used to set a value to an value

syntax

setattr(obj,attribute\_name,modifying value)

hasattr()

It checks whether the attribute exist or not

syntax

hasattr(obj,attribute\_name)

delattr()

It is used to delete an attribute  
 syntax

delattr(obj,attribute\_name)

\_\_dict\_\_

It gives all the information about your class

syntax

obj.\_\_dict\_\_

\_\_name\_\_

It gives name of your class

\_\_doc\_\_

It is used to give your document string

\_\_module\_\_

It will display the current module

\_\_bases\_\_