13-3-23. # Set Functions

add :- It is a pre-defined set built-in function. By using this function we can add the value inside the set.

-> we can assign only one value in this function.

-> It will add the value.

-> It returns none value.

Syntax : variable.add(object)

```
# s = {'a', 'b', 10, 20}
# s.add(50)
```

Program : user defined set function

```
def uadd(ele, coll):
    coll = list(coll)
    coll[-1:-1] = [ele]
    return set(coll)
uadd(50, s)
```

O/P :-

{10, 20, 50, 'a', 'b'}.

Clear: It is a pre-defined set built in function.

-> used for remove all the data items in set.

-> It doesn't accepts any arguments.

-> It clear all / removes all values.

-> It returns None value.

Program:

```
s={'a', 'b', 'c', 'd'}
# print(s.clear())
# s

def uclear(s):
    return set()
uclear(s).
```

O/p:

```
set()
```

Difference b/w arithematic and assignment operators.

addition operation

```
a = 10
b = 20
print (a+b)
print (a)
print (b)
```

O/P :-

```
30
10
20
```

assignment addition operation

```
a = 10
b = 20
print (a)
print (b)
a += b
print (a)
```

O/P :-

```
10
20
30
```

-> In arithematic operation, the operation is done b/w '2' operands.

-> In assignment operation, the '2' variables are given, are performed addition b/w 2 variables and re-assigned that value to any one of the variable.

Union : To perform the concatination operation in between two sets and it will eliminate the common elements in the second set.

-> union operation we can perform two ways.

1. By using pipe symbol (|)
2. By using built-in function.

By using pipe symbol (|):

```
S = {10, 20, 30}
t = {10, 40, 50}
res = S | t
print(res)
```

O/P :

{50, 20, 40, 10, 30}

union :

By using built in function :

-> It is a pre-defined set built in function, it is used for concatening the two sets and eliminating the common elements.

Syntax :

varname1.union(varname2)

where varname1, varname2 are sets.

this function it returns final set.

```
s = {1, 2, 3}
t = {1, 4, 5}
print(s.union(t))
```

O/P :

{1, 2, 3, 4, 5}.

update : update is a pre-defined set built in function, it is used for concating the two sets and eliminating the common elements and final set is reassigned to left hand side variable.

→ It returns None value.

program :

```
s = {10, 20, 30}
t = {10, 40, 50}
print(s)
s.update(t)
print(s)
```

O/P

{10, 20, 30}

{50, 20, 40, 10, 30}

~~{40, 10, 50}~~

difference : It will perform difference between two sets, set 1 values are compared to set 2 values what are the values are uncommon values to set 2. those values will consider as difference value we will consider diff value.

-> Difference operation we can perform two ways by using

① minus symbol and

② built in function.

by using minus symbol.

```
s = {10, 20, 30}
t = {10, 40, 50}
res = s - t
print (res)
```

O/P

{20, 30}

by using built in function

```
s = {10, 20, 30}
t = {10, 40, 50}
print (s.difference(t))
```

→ This function will accepts 'n' no. of arguments all the values should be in set.

→ It will find out the difference b/w all the sets and it return final set.

→ final set consist of difference values for left hand side set.

```
s = {10, 20, 30}
t = {10, 20, 40, 50}
print (s.difference(t))
```

O/P: {30}

```
Print (s.difference(t, {10, 20, 30, 50}))
```

O/P: set()

Syntax :-

variable1. difference (variable2, ...)

Difference_update : It is a pre-defined set built in function it is used for point out the difference values for set1 and final set value is reassigned to left hand side set.

Program :

```
s = {10, 20, 30}
t = {10, 40, 50}
print (s)
s. difference_update (t)
print (s)
```

o/p :-

{10, 20, 30}

{20, 30}

Intersection : To find out the common elements between the sets. we can perform intersection operation in two ways.

1. By using '&' symbol.
2. built in function.

By using '&' symbol.

s = {10, 20, 30}

t = {10, 40, 50}

res = s & t

print(res)

{10, }

By using intersection ~~Symbol~~.

s = {10, 20, 30}

t = {10, 20, 40, 50}

print(s.intersection(t))

{10, 20}

-> It is a set pre-defined built in function
-> used to find out common elements b/w two sets.
-> This function will accept one or more than one value and it return the final set.
-> final set is consists of all common elements between two sets.

```
s = {10, 20, 30}
t = {10, 20, 40, 50}
print (s. intersection (t))
print (s. intersection (t, {10, 20, 50}))
print (s. intersection (t, {10, 20, 30, 50}))
```

O/P :-

{10, 20}

{10, 20}

{10, 20}.

intersection_update : It is a pre defined set built in function. It is used to find out common elements b/w sets

-> The final set is updated to left hand side set.

```
s = {10, 20, 30}
t = {10, 40, 50}
print(s)
s.intersection_update(t)
print(s)
```

```
s = {10, 20, 30}
t = {10, 40, 50}
s.intersection_update(t, {10, 20})
print(s)
```

o/p :

{10}

{10}

Symmetric difference :

-> It performs and find out the uncommon elements in between sets (or) find out difference values b/w sets,

① By using symbol caret (^)

② built-in function.

S = {10, 20, 30}

t = {10, 40, 50}

res = S ∧ t

print (res)

14-3-23.

program

S = {10, 20, 30}

t = {10, 20, 40, 50}

Print (s. Symmetric - difference(t))

O/P :

{30, 40, 50}

Symmetric - difference - update :

It will find out the uncommon elements in b/w both the sets and it return final set.

-> The final set is reassigned to left hand side set variable.

program :

S = {10, 20, 30}

t = {10, 40, 50}

print (S)

S. symmetric - difference - update(t)

print(S)

O/P :-

{10, 20, 30}

{40, 20, 30, 50}

pop : It is a pre-defined set built-in function. It is used for eliminating the random element inside the set.

syntax :

variablename.pop()

-> This function will not accept any arguement and it returns deletion element.

-> S = {10, 30, 40, 60, 70, 'str'}

>>> S.pop()

70

>>> S.pop()

'str'

>>> S.pop()

60

>>> t = S.pop()

>>> t

10

remove : It is a pre-defined set built-in function it is used for eliminating the specific element inside the set. This function will accept only one value.

-> It will delete the user specified value inside the set.

-> It returns none.

Syntax :

variablename . remove ('specificelement)

```
s = {10, 30, 40, 60, 70, 'str'}
print (s.remove ('str'))
print s
```

O/P :

{10, 40, 30, 70, 60}

Discard : It is a pre-defined set built-in function it is used for eliminating the specific element inside the set. This function will accept only one value.

-> It will delete the user specified value inside the set. It returns none.

-> Difference between remove and discard

remove function :- If element is not present inside the set, it try to delete it returns key error.

```
S= {10, 30, 40, 'str'}
print (s.remove('st'))
```

o/p :-

keyerror : 'st'

discard function :- If element is not present inside the set, if try to delete the element then it return none.

```
S= {10, 30, 40, 50, 60, 'str'}
print (s.discard('st'))
```

->'st' is not present in 's

o/p :-

None.

Dict. Functions :-

Clear :- It is a pre-defined dict built in function it is used for remove the all the key and value pairs inside the dict.

-> It will not accept any values.

-> It will return None value.

Syntax :-

```
variable.clear()
```

```
d = {'a':150, 'str':50}
d.clear()
print(d)
```

O/P :

```
{}
```

Program :- user defined function.

```
d = {'a':150, 'str':50, 'Pys':'qsp'}
def uclear(d):
    l = list(d)
    start = 0
    while d:
        del d[l[start]]
        start+=1
```

```
print(d)
print(d.clear(d))
print(d)
```

O/P:

{'a':150, 'str':50, 'pys':'qsp'}

None

{ }

get: It is a pre-defined dict built in function it is used for extracting the value from the dict by the help of key.

-> This function will accept only one arguement.

-> It return the value based on key.

-> If key is not available then it will not raise the exception, instead of it returns None.

example:

```
d = {'a':150, 'str':50, 'pys':'qsp'}
print(d.get('a'))
```

150

```
Print(d.get('b'))  --> None
print(d['b'])  --> keyerror: 'b'
```

-> 'b' is not available in dict.

Program: user defined get function

```
d = {'a':150,'str':50,'pys':'qsp'}
def uget(key, coll):
    if key not in coll:
        return
    else:
        return coll[key]
print(uget('a',d))
```

~~None~~. O/P:

150.

Pop: It is a pre-defined dict built in function it is used for deleting the key value pair inside the dict.

Note:

It will delete the specified key and value pair.

-> This function will accept one arguement, that arguement we consider as key.

-> if key is present inside the dict. it will delete key and value.

→ It returns value.
→ If key is not available then it return key error.
→ If we don't want key error then defaultly we have to pass the one more argument inside the pop function.

Syntax:

varname.pop(key, defaultvalue)

key → Specified key (Mandatory)

defaultvalue → Any value (optional)

example:

```
d = {'a':150, 'str':50, 'pys':'sp'}
print(d.pop('str'))
```

O/P:
50.

```
print(d.pop('c'))
```

O/P:
KeyError: 'c'

```
print(d.pop('c',45))
```

O/P:
45

```
print(d.pop('c',None))
```

O/P:
None

popitem :- It is a pre-defined dict built in function, it is used for deleting the random key value pair.

Note:

Mostly it will delete end key value pair.

-> It will not accept any arguments.

-> It will delete random key value pair.

-> It will return deleted key value pair.

-> If elements (key value pair) is not available inside the dict, still if we perform popitem function then it return 'typeerror'.

Syntax :-

```
varname.popitem()
```

-> The return value should be in the form tuple format. In tuple 1st value is key, 2nd value is value.

```
d = {'a': 'pys', 'b': 19}
print(d.popitem())
```

O/P :-

('b', 19)

```
d = { }
print(d.popitem())
```

O/P :-

typeerror.

setdefault :- It is a pre-defined dict built in function and it is used for get the value from dict and create the new value pair inside the dict.

-> if given key is present then it returns corresponding value.

-> if key is not present it return None

-> if key is not present and defaultly we assign any value inside the function then it create new key and value

inside the dict.

```
>>d = {'a': 10, 'b': 20}
print (d.setdefault('a', 150))
```

o/p:

10.

```
>>d = {'a': 10, 'b': 20}
print (d.setdefault('c', 150))
```

o/p:

{'a': 10, 'b': 20, 'c': 150}.

```
>>d = {'a': 10, 'b': 20}
print (d.setdefault('c'))
```

o/p:

None.

15-3-23.

user defined set default function :-

```
def usetdefault (coll, key, defu = None):
    if key not in coll:
        coll[key] = defu
    return coll[key]
```

```
else :
    return cold[key]

d = {'a': 150, 'str': 50, 'pys': 'qsp'}
print (usetdefault (d, 'py', 1500))
print (d)
```

```
1500
{'a': 150, 'str': 50, 'pys': 'qsp',
 'py': 1500}
```

(doubt)

update :- It is a pre-defined dict built in function, it is used for concating the two dictionaries.

-> This function will accept only dict type of value, the dict value should be key as a string and value has any type of value.

-> It returns None value, but it will concate the two dictionaries.

Syntax :-

```
var. update ({key : value})
```

Any value

key should be string

example :-

```
d = {'a': 150, 'str': 50, 'pys': 'qsp
     'py': 1500}
print(d.update({'pys': (150, 15)}))
print(d)
```

o/p :-

None

```
{'a': 150, 'str': 50, 'pys': (150, 15),
 'py': 1500}.
```

program: ~~user defined update function~~

wap to concate the given dicts.

```
def ~~update~~ udconcat (coll1, coll2):
    for i in coll2:
        coll1[i] = coll2[i]

d = {'a': 150, 'str': 50, 'pys': 'qsp
     'py': 1500}
print(d)
udconcat (d, {'abc': 150, 'efg': 500,
              'a': 750})
Print(d)
```

O/P :-

{'a' : 150, 'str' : 50, 'pys' : 'qsp'
'py' : 1500}

{'a' : 750, 'str' : 50, 'pys' : 'qsp'
'py' : 1500, 'abc' : 150, 'efg' : 500}.

program : user defined update function. (doubt)

```
def update(coll1, coll2):
    for j in coll2:
        if type(j) != str:
            return 'key's should be
                    string'
    for i in coll2:
        coll1[i] = coll2[i]

d = {'a' : 150, 'str' : 50, 'pys' : 'qsp'
     'py' : 1500}
print(d)
print(update(d, {'abc' : 150, 'b' : 500,
                 'a' : 750})
print(d)
```

{'a':150, 'str':50, 'pys':'qsp'
'py':1500}

None

{'a':750, 'str':50, 'pys':'qsp'
'py':1500, 'abc':450, 'lo':500}

String built-in functions :-

Some examples:- startswith,
center, endswith, find, index, isidentifier,
join, lstrip, replace, rfind, rindex,
rsplit, rstrip, split, splitlines, strip.

Center : It is a string pre-defined built-in function. used to center the string based on the given number.

```
St = 'hello'
print (st.center (9,'*'))
```

O/P :-

hello

Program :

```
def ucenter (st, num, sym = "_"):
    if len (st) > num:
        return st
```

else:

diff = num - len(st)

left = diff // 2

right = diff - left

print (sym * left + st + sym * right)

center (st, 20, '*')

O/P :-

----------hello----------.

startswith :- It is a pre-defined string built-in function, it is used to compare the elements starting of the string (or) it compare the elements specified positions.

-> If elements is match starting of the string then it returns true.

(specific position)

example :

S = 'PYSPIDER'

S. startswith ('S', 2, 6)

O/P :- True.

16-3-23

Program : user defined startswith function: case 1 :-

```
def ustartswith(coll, sub, start=0
                end = 100000):
    if len(sub) > len(coll):
        return False
    else:
        i = 0
        s = ""
        while i<len(sub) and start<end:
            if coll[start] == sub[i]:
                s += coll[start]
                start += 1
                i += 1
            else:
                return False
    if s == sub:
        return True
```

else :

return False

startswith ('PYSPIDERS', 'DERS', 5, 7)

0 5 7

1 6 7

DE DERS

o/p :-

False.

Endswith :- It will compare elements or ending of the string specific position of the given string.

st = 'PYSPIDERS'

st.endswith ('SPID', 2, 6)

Case 1 :-

def endswith (coll, sub, start=0, end=100000):

coll = coll [start : end] [: : -1]

sub = sub [: : -1]

if len(sub) > len(coll):

return False

else :

i = 0

s = ""

```
while i < len(sub) and start<end:
    print(coll[i], sub[i])
    if coll[i] == sub[i]:
        s+ = coll[i]
        it =1
    else
        return False
if s== sub and len(sub)!=0:
    return True
else:
    return False
```

st = 'PySPIDERS'

endswith(st, 'DERS').

o/p :-

True

Startswith :-

Case 2 :

s= 'PySPIDERS'

```
def ustartswith(coll, sub, start=0, end=10000):
    if len(sub) > len(coll):
        return False
```

```
else:
    if start + len(sub) <= end and
        Coll [start : Start + len(sub)] == sub:
        return True
    else:
        return False
```

```
startswith (s, 'pys', 0, 3)
```

o/p :-
True

endswith :-

Case 2 :

```
s = 'pyspiders'
```

```
endswith (coll, sub, start=0,
          end = 10000):
    coll = coll [start:end][::-1]
    sub = sub [::-1]
    print (coll, sub)
    if len(sub) > len(coll):
        return False
```

```
else :
    start = 0
    if start + len(sub) <= end
        and call[start : start + len(sub)] == sub:
        return True
    else :
        return False
```

```
u endswith(s, 'spi', 2, 5)
```

O/P :-

False.

17-3-23.

Split function :-

```
def u split(call, sub):
    l = [ ]
    s = ""
    for i in call:
        if i != sub:
            s += i
        else :
            if len(s) != 0:
                s = ""
                l += [s]
```

```
if len(s)!=0:
    l+=[s]
return l
print(usplit("welcome,to,pyspiders",','))
```

sort function :-

```
def usort(coll):
    for i in range(0,len(coll)):
        for j in range(0,len(coll)-1):
            if coll[j] > coll[j+1]:
                coll[j], coll[j+1]
                = coll[j+1], coll[j]
    print(coll)
usort(['apple','dog','cat','bat','watch',
       'ice','egg','gun','fox','ant'])
```

join function :-

```
def ujoin(coll,sub):
    s=""
    for i in coll:
        s+=str(i)+sub
    return s[:-(len(sub))]
ujoin(['ant','apple','bat','fox'],',')
```

O/P :- 'ant,apple,bat,fox'.

Program :-

```
l = [`google.com', `yahoo.com', `fb.in',
     `insta.edu', `python.edu']
d = { }
for i in l:
    r = i.split(`.')
    if len(r[0]) not in d:
        d[len(r[0])] = [r[0]]
    else:
        d[len(r[0])] += [r[0]]
print(d)
```

O/P :-

```
{6: [`google', `python'],
 5: [`yahoo', `insta'], 2: [`fb']}
```

program :-

```
l = [`google.com', `python.edu',
     `google.in', `python.in',
     `insta.in', `python.com']
d = { }
for i in l:
    r = i.split(`.')
    if r[0] not in d:
        d[r[0]] = 1
    else:
        d[r[0]] += 1
print(d)
```

O/P :-

```
{`google': 2, `pytho': 3,
 `insta': 1}
```

Program :-

```
l = [`google.com', `yahoo.com',
     `fb.in', `insta.edu',
     `python.edu']
```

```
d = { }
for i in l:
    r = i.split(':')
    if len(r[-1]) not in d:
        d[len(r[-1])] = r[-1]
    else:
        d[len(r[-1])] += r[-1]
print(d).
```

Program :-

```
St = input("enter the value:")
st = st.split(',')
St.sort()
res = ','.join(st)
print(res)
```

18-3-23.

split :- It is a pre-defined string built in function it is used for splitting the main strings into sub strings based on seperator and maximum number of splits.

-> It accepts two arguments,

1. argument is seperator
2. Number of splits.

-> It will perform splitting operation left to right.

-> default seperator argument is space.

-> This function it returns list of sub strings.

Syntax :-

varname . split (seperator, no. of splits)

-> The first argument should be in string format.

-> second argument should be number.

Example :-

```
s = 'python is a programming
    language'
print(s.split())
['python', 'is', 'a', 'programming', 'language']
print(s.split('g'))
['python is a pro', 'rammin', 'lan',
 'ua', 'e']
print(s.split(' ', 2))
['python', 'is', 'a programming language']
print(s.split(maxsplit=2))
['python', 'is', 'a programming language']
```

program :- user defined split function.

```
def usplit(coll, sep):
    l=[]
    st=''
    for i in coll:
        if i!=sep:
            st+=i
```

```
    elif len(st) != 0:
        l += [st]
        st = ''
    if len(st) != 0:
        l += [st]
    return l
split(s, ' ')

def usplit(coll, sep, maxh = -1):
    count = 0
    if maxh == 0:
        return [coll]
    elif maxh < 0:
        maxh = len(coll)
    l = []
    st = ''
    for i in range(0, len(coll)):
        if coll[i] != sep:
            st += coll[i]
        else:
            count += 1
```

```
        if count < maxh:
            lt = [st]
            st = ''
        else:
            lt+=[st, coll[i+1:]]
            st = ''
            break
    lt =[st]
    return lt
s = 'python_is_a_programming_language'
usplit(s, '_')
['python', 'is', 'a', 'programming',
 'language'].
```

program: user defined rsplit function.

```
def rsplit(coll, sep, maxh = -1):
    count = 0
    if maxh == 0:
        return [coll]
    elif maxh < 0:
        maxh = len(coll)
```

```
l = [ ]
st = " "
for i in range(-1, -(len(colu)+1), -1):
    if colu[i] != sep:
        st = colu[i] + st
    else:
        count += 1
        if count < maxh:
            l = [st] + l
            st = " "
        else:
            l = [colu[i::-1][::-1]
                 st] + l
            st = " "
            break
if len(st) != 0:
    l = [st] + l
return l
```

s = "python is a programming language"

print(s.rsplit(' ', 0))

rsplit(s, " ", 0)

20-3-23

Arguments :- arguments is nothing but values or function inputs or requirements or properties or parameters.

-> To perform any task inside the function block, for that task it require some inputs, so those inputs we call it as arguments.

-> For these arguments first we have to declare and intialize the inputs in function signature.

-> Arguments are classified into two types;

1. formal arguments.
2. actual arguments.

Formal arguments :- These arguments are present in function definition as a user we have to declare the variables inside the function definition.

-> In function definition what are the variables we have or we write those variables we call as formal arguments.

Note :-

-> directly we can't assign the values inside the function definition.

Actual arguments :- These arguments are present inside the function call.

-> As a user we have to pass/ assign the values directly (or) indirectly inside the function call.

Note :-

-> In function call we can't declare the variables.

Types of formal arguments and Actual arguments

1. Positional arguments
2. keyword arguments
3. default arguments (default and non-default)
4. variable length arguments.

Positional arguments :- These are the arguments passed to function definition or function call in correct Positional order.

-> In function definition we should declare the variables.

-> In function call directly or indirectly we should declare the values.

-> The no. of actual and formal arguments should be same.

-> If we change the order of arguments then result may be change.

-> If we change or miss the no. of arguments then we will get error.

```
def sample (a,b,C):
    print ('haii', a, b, C)

sample (1,2)
```

Typeerror : sample( ) missing 1 required positional argument 'c'

```
def sample (a,b,C):
    print ('haii', a,b, C)

Sample (1, 2, 3, 4)
```

Typeerror : sample( ) takes 3 arguments but 4 were given.

Keyword arguments :- these are the arguments passed to function call by using variable name with value in order (or) (not in order) unordered.

→ We can pass the arguments (values) by using keyword names. — Those names with matched to the function declaration variable name.

→ order of the arguments is not important but no. of arguments must be match.

```
def sample(a,b):
    c=a-b
    print(c)
sample(a=60, b=20)
```

O/P :-

40

```
def sample(a,b)
    c= a-b
    print(c)
sample(b=20, a=60)
```

O/P :-

40

```
def sample(a,b,c)
    c=a-b
    print(c)
sample(a=60, b=20, d=30)
```

O/P :-

TypeError: sample() got an unexpected keyword argument 'd'

Combination of positional and keyword arguments :-

-> We can pass the Combination of positional and keyword arguments in function call.

-> Whenever we are using both args. we should follow one syntactical rule. -> positional arguments followed keyword arguments.

```
def Sample1(a, b):
    print(a, b)
Sample1(10, b=20)
```

o/p :-

10 20

```
def Sample2(a, b):
    print(a, b)
Sample2(20, a=10)
```

o/p :-

TypeError : Sample2() got multiple values for argument 'a'

```
def Sample2(a, b)
    print(a, b)
Sample2(a=20, a=10)
```

o/p :-

SyntaxError : keyword argument repeated : a

default arguments :- these are the arguments we are passing inside function definition with values.

→ In function definition we can pass variable name with values. this process we call as variable initialization in function definition.

→ In function definition we can provide the default values for the formal positional arguments.

→ Some times we don't want to assign the values in function call because in function definition we define variable intialization for each variable it have default value.

```
def sample(a=10, b=20):
    print(a,b)

sample( )
```

default arguments another name called optional arguments.

→ For non - default arguments or positional arguments another name called mandatory arguments.

```
def sample (a=10, b=20):
    print (a, b)
sample ()
```

O/P:

10 20

```
def sample (a=10, b=20):
    print (a, b)
sample (100, 200)
```

O/P:

100 200

-> user define any values in func. call then function definition consider user defined values. (it will not consider defaait values).

```
def sample (a=10, b=20):
    print (a, b)
sample (b=200, a=100)
```

O/P:

100 200

```
def sample (a=10, b=20):
    print (a, b)
sample (100, b=200)
```

O/P:

100 200

4-3-23.

Combination of non-default and default:

Whenever we are declaring the variables in function definition so that time we have to follow one rule that is "non-default argument follows default argument".

```
def sample(a=10,b):
    print(a,b)
sample(100,200)
```

O/P :-

Syntaxerror:

non-default argument follows default argument.

```
def sample(a,b,c=0,d=0):
    return a+b+c+d
sample(1,2,3,4)
```

O/P :-

10.

```
def sample(a,b=0,c,d=0):
    return a+b+c+d
sample(1,2,3,4)
```

O/P :-

Syntaxerror : non default argument follows default argument.

Variable length argument :- The Variable length arguments are present in function definition.

-> The variable length argument will accept zero or more than zero values.

-> The variable length arguments are declare in function definition.

-> The variable length arguments are classified into two types.

1. tuple variable length argument (* args)
2. dictionary variable length argument (** kwargs)

tuple variable length argument :-

This argument will accept only positional arguments from function call.

-> It will store all the values in the form of tuple format.

-> This process we call it as tuple packing.

dictionary variable length argument :-

- It will accept only keyword arguments from function call.
- Each and every value will store in form of dictionary.
- keyword names we consider as keys and values are consider as values.
- this process we call it is dictionary packing.

```
def sample(*args):
    print(args)
sample(1,2)
```

O/P :-
(1, 2)

```
def sample(**kwargs)
    print(kwargs)
sample(a=1, b=2, c=4, d=5)
```

O/P :-
{'a': 1, 'b': 2, 'c': 4, 'd': 5}

```
def demo1(n, n, o=10, p=20, *args):
    print(n,n,o,p,args)
demo1(1,2)
```

O/P :-
1 2 10 20 ()

```
def demo2(m, n, o=10, p=20, *args):
    print(m, n, o, p, args)

demo2(1, 2, 100, 200, 3, 4, 5, 6)
```

O/P :-

1 2 100 200 (3, 4, 5, 6)

```
def demo3(*args, *A):
    print(args)

demo3(1, 2, 3)
```

O/P :-

* argument may appear only once.

```
def demo3(*args, a, b):
    print(args, a, b)

demo3(1, 2, 3, b=20, a=10)
```

O/P :-

(1, 2, 3) 10 20

```
def demo3(*args, a, b):
    print(args, )

demo3(1, 2, 3, b=20, a=10, c=10)
```

Typeerror : demo3() got an unexpected keyword argument 'c'.

```
def demo3(*args, **kwargs):
    print(args, kwargs)

demo3(1, 2, 3, b=20, a=10, c=30)
```

(1, 2, 3) {'b': 20, 'a': 10, 'c': 30}

```
def demo(a, *args, h, **kwargs)
    print(a, args)
    print(h, kwargs)
demo(10, 150)
```

O/P :-

demo() missing 1 required keyword-only argument: 'h'

```
def demo(a, *args, h, **kwargs)
    print(a, args)
    print(h, kwargs)
demo(10, h=150)
```

O/P :-

(10 ()
150 {}

```
def demo(a, *args, h=150, **kwargs)
    print(a, args)
    print(h, kwargs)
demo(10, 1, 2, 3, 8, 9, 300, n=15, o=100, p=150)
```

O/P :-

10 (1, 2, 3, 8, 9, 300)
150 {'n': 15, 'o': 100, 'p': 150}

Packing and unpacking :-

Packing :-

-> Packing is a phenomenon of grouping collection of data items into a single data collection.

-> Packing can be done between the pair of single quotes (' '), double quotes (" "), square brace [ ], parentheses ( ), flower brace { }.

-> Packing will help us to keep the data to store, secured, save and easy to maintain.

-> Whenever the data transition happens there should not be data loss. The data packing should happen only in the form of immutable type.

-> If we do packing in the form of different packing techniques and storing, then the data seperation and data utilization will be a difficult task for us. That is the reason we are packing in the form of tuples data type with help of variable length arguments.

-> As programmers we can pack our data between any of the symbols, for example, single quotes (' '), double quotes (" "), square brace [ ], parentheses ( ), flower brace { }.

-> But the python application will be packed in the form of tuple and sometime dict.

-> It is classified into two types.

1. pre-defined packing.
2. user-defined packing.

packing: The sender side (function calling) will be a positional arguments or keyword arguments at the same time receiver side (function definition) will be the *args or **kwargs.

Syntax:-

```
def funcname(*args) -> tuple packing
    |
    | statement/instruction
    |
```

func_name(val1, val2, val3, ...)

note: we should pass only positional args.

def func_name(**kwargs): ← dict packing

|

|statement/instruction

|

func_name(~~val1~~, ~~val2~~, key=val1, key=val2 . . . .)

Note: ~~we should pass first positional followed by keyword args~~

def func_name(*args, **kwargs): ← tuple and dict packing.

|

| statement/instruction

|

func_name(val1, val2, key=val1, key=val2 . . .)

note:- we should pass first positional followed by keword arguments.

Packing the data items are in two ways: (variable length argument)

1. Tuple packing (*args)
2. Dictionary packing (key and value pair (**kwargs))

tuple Variable length packing:-

-> If we wanted to pack the data while doing the data transition from function call to function definition, then we can use the special operator called '*' with arg name.

we are performing variable length argument concepts at the receiving side (function definition) to pack the data.

-> In this packing we can pass the only positional arguments in the sender side (function calling).

-> So that it helps to do the packing in the form of tuples.

-> In industrial standards, we should follow *args.

dictionary variable length packing :-

-> If we wanted to pack the key value pair data while doing the data transition from function call to function definition, then we can use the special operator called '**' with kwargs name. we are performing variable length argument concepts at the receiving end (function definition) to pack the data.

-> In this packing we can pass the only keyword arguments in actual arguments (function calling).

-> So that it helps to do the packing in the form of a dictionary.
-> In industrial standards, they define a keyword called ** kwargs.

Unpacking:

-> Unpacking is a phenohenal way of extracting the data from the collection and storing it in the different memories (different arguments).
-> In the sender side (function calling) will be a *args or **kwargs at the same time receiver side (function definition) will be the number of arguments created based on length of the packed collection.

Syntax:-

tuple packed collection

```
def func_name(arg1, arg2, arg3...)
        |
        |Statement/instruction
        |
func_name(*args)
```

dictionary packed collection :

```
def func-name(arg1, arg2, ...):
    |
    |Statement/instruction
    |
func-name(**kwargs)
```

note :- dict packed collection key names and formal arguments names should be the same.

combination of tuple and dict packed collection :-

```
def func-name(arg1, arg2, arg3...):
    |
    |Statement/instruction
    |
func-name(*args, **kwargs).
```

22-3-23.

Unpacking example :-

```
def abc(a,b,c):
    print(a,b,c)
abc(*'xuv')
```

O/P :-
x u v

```
def abc(a,b,c):
    print(a,b,c)
abc(*[1,2,3])
```

O/P :-
1 2 3

```
def abc(a,b,c):
    print(a,b,c)
abc(*{1,2,3})
```

O/P :-
1 2 3

```
def abc(a,b,c):
    print(a,b,c)
abc(*(1,2,3))
```

O/P :-
1 2 3

```
def abc1(a,b,c):
    print(a,b,c)
abc1(*'xu')
```

O/P :-
error :
positional args required - 3
but given - 2

```
def abc(a,b,c):
    print(a,b,c)

abc(*{'a':1,'b':2,'c':3})
```

O/P :-

a b c

```
def abc(a,b,c):
    print(a,b,c)

abc(**{'a':1,'b':2,'c':3})
```

O/P :-

1 2 3.

WAP to sum of all the numbers in given collection by using packing and unpacking.

```
def add(*args):
    s=0
    for i in args:
        s+=i
    return s

res=add(1,2,3,4,5,6,7,8,9,0)
print(res)
```

O/P :-

37.

```
def add(*args):
    return sum(args)

res=add(1,2,3,4,5,6,7,8,9,0)
Print(res)
```

O/P :-

~~37~~

45.

23-3-23.

global :-

```
a = 169
def sample():
    print(a)
sample
```

>>> 169.

The global variable we can access inside local space.

```
a = 169
def sample():
    print(a)
    a = 100
sample()
```

o/p :-

Unbound local error: Cannot access local variable 'a' where it is not associated with a value.

Note :-

-> without permission global variable we cannot modify inside the local space.

-> If we want to modify we have to get the permission from global space.

-> By the help of global keyword we can get the permission from the global space.

```
a = 169
print(globals())
print('-' * 150)
```

```
def sample():
    print(locals())
    global a
    print(a)
    a=100
    print(a)
    print(locals())
sample()
print(globals())
```

O/P:

{ }

169

100

{ }

{. . . . .

'a':100, 'sample':

<fetch sample

at 0x. . . . >}

global variables:-

	Global space	local space	non local space
initialization	✓	✗ (global) ✓	✗ (global) ✓
Fetch/Access	✓	✓	✓
reinitialization	✓	✗ (global) ✓	✗ (global) ✓
delete	✓	✗ (global) ✓	✗ (global) ✓

```
print (globals())
a=169
print (globals())
print (a)
a=177
print (globals())
print (a)
del a
print (globals())
```

O/P :-

{ }

{ }

24-3-23

```
a=150
print (globals())
print ('_' * 75)
def sample():
    print ("locals", locals())
    def demo():
        global b
        print ("non local", locals())
        #Print(a)      #fetch global variable
        # b=500       # intialization global variable
        # a=222       # reinitialization global variable
        # del a       # delete global variable
        print ("non local", locals())
    demo()
```

```
    print("local", locals())
sample()
print('-'*75)
print(globals())
```

local variables :-

```
print(globals())
print('-'*75)
def sample():
    print("locals", locals())
    a=777  # local variable intialization
    print(a)  # local variable fetch
    print("locals", locals())
    a=555  # local variable
           reinitialization
    print(a)
    print("local", locals())
    del a  # local variable delete.
    print("local", locals())
sample()
print('-'*75)
print globals()
```

local variable in global :-

```
print (globals())
print ('-'*75)
def sample():
    print ("locals", locals())
    a = 777
    print ("locals", locals())
sample()
print (a)
print ('-'*75)
print (globals())
```

O/P :-

locals { }

locals {'a': 777}

NameError : name 'a' is not defined.

local variables :-

	global space	local space	non local space
Initialization	✗	✓	✗
fetch	✗	✓	✓
reinitialization	✗	✓	✗ (nonlocal) ✓
delete	✗	✓	✗ (nonlocal) ✓

local variable in non-local :-

```
print(globals())
print('-'*75)
def sample():
    print('sample locals', locals())
    a = 777
    b = 'hello'
    print('sample locals', locals())
    def demo():
        nonlocal a
        print('non locals', locals())
        print(a) # fetch the local variable
        # a = 999 # reinitialize the local variable
        # del a # delete the local variable
        print('non locals', locals())
    demo()
    print('sample locals', locals())
sample()
print('-'*75)
print(globals()).
```

27-3-23.

non-local variable :-

```
print(globals())
print('-'*75)
def sample():
    print("sample locals", locals())
    def demo():
        print("non locals", locals())
        x = 100  #non local variable initialization
        print("non locals", locals())
        print(x)  #non local variable fetch
        x = 500  #non local variable reinitialization
        print("non locals", locals())
        print(x)
        print("non locals", locals())
        del x #non local variable delete
        print("non locals", locals())
    demo()
    print("sample locals", locals())
sample()
print(globals())
```

Global variables:

-> These are variables which are created in the main space, we can access from any part of the program.

-> If we wanted to access, modify, delete and create these variables anywhere inside the main space, function area and nested function area, we can modify the global variables inside the main space, but we cannot modify, delete the global variable inside the function area/nested function area. If we want to do this task, then with the help of global keyword we can modify the global variables inside the function area/nested function area.

Syntax:

```
def fun_name():
    global var1, var2, var3.....varn
    modification statements
func_name()
```

-> The global keyword should be the first instruction in the function.

-> The modification of the global variables will have no impact inside the function, it will impact the main space.

Syntax:

```
global var1, var2 ------ varn
```

-> global variables because these are created outside the functions (inside the main space) the actual name of the global variables should be the same as the names of the variables in function that we use in the below functions example.

```
A=10
def sample():
    global A
    print(A)
    A=10
    print(A)
Sample()
```

Local variables :-

-> These are the variables defined or created inside the function area/nested function are known as local variables.

-> we can access, modify, delete, initialize the local variables with in the function.

-> when the control enters the function area/method area then it executes those statements. These variables will be created once the function call is done. (variable life scope will start the function execution is done, variable scope is destroying. these variables will be deleted once the control comes out of the function area/method area)

-> we can access local variables inside child functions (nested functions) but it is not possible to manipulate local variables inside nested functions.

-> To overcome the above problem, we make sure use of nonlocal keyword.

Syntax :-

```
def hello():
    var1, var2, var3...varn = val1, val2...valn
        # all these variables are called
          as local variable.
hello()
```

→ When the control reaches the function area. all statements are executed one by one. if ~~it~~ in case any variables are there, all things will be created inside the function area memory.

```
def hello():
    a=10
    b=20  # all these variables are called as local variable.
    def sample():
        x=10
        y=20  # all these variables are called as nonlocal variable.
    sample()
hello()
```

non local keyword :-

→ If we want to access the local variables of nested functions directly we can do so and the nested function is allowed. But if we want to modify, delete then it will not have allowed inside the nested function. If ever we still wanted to modify, delete the local variables in nested functions then we are using a non local keyword.

```
def hello():
a=10
b=20  # all these variables are called as local variable,
def sample():
    non local a
    x = 10
    a = x      -> # modify the local variable
    y = 20     -> # all these variables are called as non_local variables.
    sample()
hello().
```

global variable	global space	local space	non-local space
initilization	possible	possible (global kw)	possible (global kw)
fetch	possible	possible	possible
re-initilization	possible	possible (global kw)	possible (global kw)
delete	possible	possible (global kw)	possible (global kw)

Local variable	global space	local space	non local space
initilization	impossible	possible	impossible
fetch	impossible	possible	possible
re-initilization	impossible	possible	possible (nonlocal kw)
delete	impossible	possible	possible (nonlocal kw)

nonlocal variable	global space	local space	non local space
initilization	impossible	impossible	possible
fetch	impossible	impossible	possible
reinitilization	impossible	impossible	possible
delete	impossible	impossible	possible

```
a = 10
def Parent(x, y):
    print(x, y)
    def child():
        return 10
    return child
res = Parent(50, 60)
print(res)
```

O/P :-

50 60

<function Parent.<locals>.child at 0x0000>