STRING

In [2]:
```
txt = "The best things in life are free!"
if "free" in txt:
 print("Yes, 'free' is present.")
txt = "The best things in life are free!"
if "expensive" not in txt:
 print("Yes, 'expensive' is NOT present.")




b = "Hello, World!"
print(b[2:5])
# Negative Indexing
b = "Hello, World!"
print(b[-5:-2])


#Upper Case
a = "Hello, World!"
print(a.upper())
#Lower Case
a = "Hello, World!"
print(a.lower())
#Remove Whitespace
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
#Replace String
a = "Hello, World!"
print(a.replace("H", "J"))
#Split String
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

```
Yes, 'free' is present.
Yes, 'expensive' is NOT present.
llo
orl
HELLO, WORLD!
hello, world!
Hello, World!
Jello, World!
['Hello', ' World!']
```

LIST

In [3]:
```
thislist = ["apple", "banana", "cherry"]
print(thislist)

thislist = ["apple", "banana", "cherry"]
print(len(thislist))

thislist = ["apple", "banana", "cherry"]
print(thislist[1])

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])

thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
    print("Yes, 'apple' is in the fruits list")


#The insert() method inserts an item at the specified index
thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")
print(thislist)
#Using the append() method to append an item
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)


#The remove() method removes the specified item.
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
#The pop() method removes the specified index
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
#If you do not specify the index, the pop() method removes the last item
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
#The del keyword can also delete the list completely
thislist = ["apple", "banana", "cherry"]
del thislist
#The clear() method empties the list.
#The list still remains, but it has no content
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)



#You can loop through the list items by using a for loop
thislist = ["apple", "banana", "cherry"]
for x in thislist:
 print(x)
#Use the range() and len() functions to create a suitable iterable
#Print all items by referring to their index number
```

```
print("\n")
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
 print(thislist[i])
#Print all items, using a while loop to go through all the index numbers
print("\n")
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
 print(thislist[i])
 i = i + 1


#Looping Using List Comprehension
print("\n")
thelist = ["apple", "banana", "cherry"]
[print(x) for x in thelist]
```

```
['apple', 'banana', 'cherry']
3
banana
['cherry', 'orange', 'kiwi']
Yes, 'apple' is in the fruits list
['apple', 'banana', 'watermelon', 'cherry']
['apple', 'banana', 'cherry', 'orange']
['apple', 'cherry']
['apple', 'cherry']
['apple', 'banana']
[]
apple
banana
cherry


apple
banana
cherry


apple
banana
cherry


apple
banana
cherry
```

Out[3]:
```
[None, None, None]
```

LOOPS

In [4]:
```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
 print(x)
#Loop through the letters in the word "banana"
print("\n")
for x in "banana":
 print(x)
print("\n")
for x in range(6):
 print(x)
print("\n")
#Increment the sequence with a particualr value(default is 1)
for x in range(2, 30, 3):
 print(x)

print("..................WHILE LOOP.....................")
i = 1
while i < 6:
    print(i)
    i += 1
```

```
apple
banana
cherry


b
a
n
a
n
a


0
1
2
3
4
5


2
5
8
11
14
17
20
23
26
29
..................WHILE LOOP.....................
1
2
3
```

```
4
5
```

FUNCTIONS

In [5]:
```
#Eg1: type() -> method returns class type of the argument
print("Example for 'type' built-in function: ",type([]))
#Eg2: print() -> Prints to the standard output device
print("Example for 'print' built-in function")
#Eg3: callable() -> returns true if the object passed appears to be callable
def call(x):
 return x
print(callable(call))


#Eg:
def foo():
    pass
print(type(foo))
print("foo name attribute: ",foo.__name__)
```

```
Example for 'type' built-in function:  <class 'list'>
Example for 'print' built-in function
True
<class 'function'>
foo name attribute:  foo
```

LAMBDA

In [6]:
```
#Eg1:
y = lambda x:x*2
print(y(5))
#Eg2:
lst = [1,5,7,14]
newlst = list(filter(lambda a:a%7==0,lst))
print(newlst)
```

```
10
[7, 14]
```

METHODS

In [7]:
```
#Eg:
class Abc:
    def m1(self):
        pass
#Eg: __init__, __call__, __class__, __self__
def __init__(self):
    pass
```

CLASS

In [9]:
```
#Eg:
class sample:
    def __init__(self):
        self.a = 13
        self.b = 15
s = sample() # instance creation
print(s.a)
```

```
13
```

Class instances

In [10]:
```
class sample:
    def __call__(self):
        pass
s = sample()
print(callable(s))
```

```
True
```

Functions and Closures

In [12]:
```
def local_function():
    print("This is a local function")
local_function()

#Eg2:
#function inside another fn
def a():
    def b():
        print("inside b")
    print("inside a")
    b()
a()
```

```
This is a local function
inside a
inside b
```

In [13]:
```
def outer_func():
    def inner_func():
        print("This is a nested function")
    inner_func()
outer_func()

# Inner functions can access variables and objects from their enclosing function
#Eg:
def outer_func(a):
    def inner_func():
        print("This is a nested function",a)
    inner_func()
outer_func("'hi!'")

#referncing function as an argument
def abc(x):
    return x**2
def xyz(func): #name alone->reference
    num=10
    return func(num)
xyz(abc)
```

```
This is a nested function
This is a nested function 'hi!'
```

Out[13]:
```
100
```

In [15]:
```
#Eg:
def generate_power(exponent):
    def power(base):
        return base ** exponent
    return power
g = generate_power(3) # exponent->3
print(g(2)) # base->2
```

```
8
```

DECORATORS

In [21]:
```
#Example for normal function calling
def addexclamation(function):
    def add():
        func = function()
        return func +" !!!"
    return add
def sentence():
    return "hello all"
msg = addexclamation(sentence)
print(msg())


#Example of using decorators on same above example
def addexclamation(function):
    def add():
        func = function()
        return func +" !!!"
    return add
@addexclamation
def sentence():
    return "hello all"
print(sentence())
```

```
hello all !!!
hello all !!!
```

In [22]:
```
def addstar(func):
    def star():
        return "*"+func()+"*"
    return star
@addstar
@addexclamation
def sentence():
    return "hello all"
print(sentence())
```

```
*hello all !!!*
```

In [23]:
```
#Example for decorator with arguments
def args_function(func):
    def getargs(arg1,arg2):
        print(arg1,arg2)
        func(arg1,arg2)
    return getargs
@args_function
def decorator_with_args(num1,num2):
    print("arguments are {} and {}".format(num1,num2))
decorator_with_args(5,6)
```

```
5 6
arguments are 5 and 6
```

LIST COMPREHESSION

In [24]:
```
list1 = [x**2 for x in range(10)]
print(list1)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In [25]:
```
list2 = [5,2,7,8,14,22]
list3 = [x for x in list2 if x%2==0]
print(list3)
```

```
[2, 8, 14, 22]
```

In [26]:
```
#lambda - is a function without a name and lambda keyword is used to define this
#Syntax -> lambda arguments: expression
x = lambda a:a**2
print(x(5))
y = lambda a,b:a-b
print(y(10,6))
```

```
25
4
```

In [27]:
```
#filter() - filters the given sequence using the lambda function that tests each
#Syntax -> filter(function, sequence)
l = [1,2,3,4,5,6,7]
newl = list(filter(lambda x:x%2==0,l))
print(newl)
```

```
[2, 4, 6]
```

In [28]:
```
#map() - generates a new sequence by executing a specified function for each elem
#Syntax -> map(function, iterables)
num = []
for i in range(10):
 num.append(i)
newnum = list(map(lambda x:x+2,num))
print(newnum)
```

```
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

In [29]:
```
s='1 2 3 4'
l=s.split()
m=list(map(int,l))
n=list(map(lambda x:x**3,m))
o=list(filter(lambda x:(x%2==0),n))
print(o)
```

```
[8, 64]
```

DICT COMPREHESSION

In [30]:
```
#Eg1:
lst = [2,5,6,3,8]
dict = {x:x+5 for x in lst if x%2==0}
print(dict)
```

```
{2: 7, 6: 11, 8: 13}
```

In [31]:
```
#Eg2:
l1 = [1,2,3]
l2 = [5,8,7]
dict1 = {key:value for (key,value) in zip(l1, l2)}
print(dict1)
```

```
{1: 5, 2: 8, 3: 7}
```

In [32]:
```
#Eg3:
#Multiple if Conditional Dictionary Comprehension
dict = {'jack': 38, 'michael': 48, 'guido': 57, 'john': 33}
new_dict = {k: v for (k, v) in dict.items() if v%2!=0 if v<40}
print(new_dict)
```

```
{'john': 33}
```

In [33]:
```
#Eg4:
#if-else Conditional Dictionary Comprehension
dct = {'jack': 38, 'michael': 48, 'guido': 57, 'john': 33}
new_dct = {k: ('old' if v > 40 else 'young')
 for (k, v) in dct.items()}
print(new_dct)
```

```
{'jack': 'young', 'michael': 'old', 'guido': 'old', 'john': 'young'}
```

In [34]:
```
#Eg5:
#Nested Dictionary with Two Dictionary Comprehensions
dictionary = {k1: {k2: k1 * k2 for k2 in range(1, 6)} for k1 in range(2, 5)}
print(dictionary)
```

```
{2: {1: 2, 2: 4, 3: 6, 4: 8, 5: 10}, 3: {1: 3, 2: 6, 3: 9, 4: 12, 5: 15}, 4:
{1: 4, 2: 8, 3: 12, 4: 16, 5: 20}}
```

SET COMPREHENSSIONS

In [35]:
```
#Eg1:
lst = [1,2,3,2,5,3]
set1 = {x for x in lst}
print(set1)
```

```
{1, 2, 3, 5}
```

In [36]:
```
#Eg2:
#using for loop in set comprehensions
input_list = [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]
set2 = {var for var in input_list if var % 2 == 0}
print(set2)
```

```
{2, 4, 6}
```

In [37]:
```
#Eg1:
lst = [2,7,5,0,4,6]
gen = (x for x in lst if x%3==0)
for i in gen:
    print(i)
```

```
0
6
```

In [1]:
```
#Eg2:
gen2 = (i**2 for i in range(5))
for item in gen2:
    print(item)
```

```
0
1
4
9
16
```

Generator Comprehensions

lst = [2,7,5,0,4,6] gen = (x for x in lst if x%3==0) for i in gen: print(i)

In [3]:
```
#Eg2:
gen2 = (i**2 for i in range(5))
for item in gen2:
    print(item)
```

```
0
1
4
9
16
```

Classes in Python

Empty class

In [4]:
```
class Abc:
    pass
```

Simple class definition

In [5]:
```
class College:
    def __init__(self): #constructor to initialize attributes
        print("Welcome")

s1 = College() #instantiate an object
```

```
Welcome
```

In [6]:
```
class Values:
    def __init__(self):
        self.a = 13
        self.b = 15
s = Values()
print(s.a)
print(s.b)
s.b = 29 #change values
print(s.b)
```

```
13
15
29
```

Adding attributes to a class

In [7]:
```
class Values2:
    def __init__(hi,a,b):
        hi.a = a
        hi.b = b
s = Values2(int(input()),int(input()))
print(s.a,s.b)
```

```
11
1
11 1
```

Adding functions to a class

In [8]:
```
#Methods in objects are functions that belong to the object.
class Student:
    def __init__(self,name,age):
        self.name = name
        self.age = age

    def printname(hi):
        print("my name is :"+hi.name)
S1 = Student("arathi",22)
print(S1.name)
print(S1.age)
S2 = Student("surya",20)
S2.printname()
del S2 # deletes the class object
print(S2.name)
del S1.name #delete properties on objects
print(S1.name)
```

```
arathi
22
my name is :surya

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
C:\Users\ABBLEA~1\AppData\Local\Temp/ipykernel_10856/4152707788.py in <module>
     13 S2.printname()
     14 del S2 # deletes the class object
---> 15 print(S2.name)
     16 del S1.name #delete properties on objects
     17 print(S1.name)

NameError: name 'S2' is not defined
```

Inheritance

In [9]:
```
#parent class
class Campus():
    def __init__(self,code,name):
        self.code = code
        self.name = name

    def show(self):
        print(self.code,self.name)
c = Campus(123,'ASIET')
c.show()
```

```
123 ASIET
```

In [10]:
```
#Syntax for subclass -> class subclass_name (superclass_name):
class Student1(Campus):
    pass
s= Student1(456,'ABC') #accessing inherited props of parent class
s.show()
```

```
456 ABC
```

In [13]:
```
class Student2(Campus):
    def __init__(self,code,name,address): #using init in child overrides parents
        Campus.__init__(self,code,name) #calling parent init to keep inheritance
        self.address = address #additional prop of child class

    def all3(self):
        print(self.code,self.name,self.address)

st = Student2(156,'QWE','kerala')
st.show()
st.all3()
#If you forget to invoke the __init__() of the parent class inside child class th
# would not be available to the child class.
```

```
156 QWE
156 QWE kerala
```

Multiple inheritance

In [14]:
```
class A:
    def __init__(self):
        self.str1="hi"
        print("A")
class B:
    def __init__(self):
        self.str2 = "hello"
        print("B")

class C(A,B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        self.str3 = "bye"
        print("C")
    def printstr(self):
        print(self.str1,self.str2,self.str3)

cobj = C()
cobj.printstr()
```

```
A
B
C
hi hello bye
```

Multilevel Inheritance

In [15]:
```
class A:
    def __init__(self,num1):
        self.num1=num1
    def printnum1(self):
        print(self.num1)
class B(A):
    def __init__(self,num1,num2):
        A.__init__(self,num1)
        self.num2 = num2
    def printnum2(self):
        print(self.num2)

class C(B):
    def __init__(self,num1,num2,num3):
        B.__init__(self,num1,num2)
        self.num3 = num3
    def printnum3(self):
        print(self.num3)

cobj = C(1,2,3)
cobj.printnum1()
cobj.printnum2()
cobj.printnum3()
```

```
1
2
3
```

In [16]:
```
class Hide:
    def __init__(self):
        self.a=12
        self.__b=13 #making b private using __
class View(Hide):
    def __init__(self):
        super().__init__() #using super() with no self parameter instead of Hide.
        self.c=14

v = View()
print(v.a)
#print(v.b)
print(v.c)
```

```
12
14
```

Class Decorators

In [1]:
```
class MyDecorator:
    def __init__(self,function):
        self.function = function

    def __call__(self): #instance called in the form of function->call is execute
        var = self.function()
        print(var,"all")

@MyDecorator
def function():
    return "Hi"

function()
```

```
Hi all
```

In [2]:
```
class MyDecorator:
    def __init__(self, function):
        self.function = function

    def __call__(self, *args, **kwargs):
        self.function(*args, **kwargs)

@MyDecorator
def function(name, message ='Hello'):
    print("{} {}!".format(message, name))
function("arathi")
function("arathi","Hi")
```

```
Hello arathi!
Hi arathi!
```

In [3]:
```
class CubeDecorator:
    def __init__(self, function):
        self.function = function

    def __call__(self, *args, **kwargs):
        result = self.function(*args, **kwargs)
        return result
@CubeDecorator
def get_cube(n):
    print("given number is:", n)
    return n*n*n

print("Cube of number is:", get_cube(5))
```

```
given number is: 5
Cube of number is: 125
```

In [4]:
```
class Makeupper:
    def __init__(self,func):
        self.func = func

    def __call__(self):
        string = self.func()
        return string.upper()
@Makeupper
def enter_str():
    return "arathi"
enter_str()
```

Out[4]: 'ARATHI'

Ability to use Implement Iterators, Iterables and Collections

In [5]:
```
# define a list
my_list = [4, 7, 0, 3]
# get an iterator using iter()
my_iter = iter(my_list)
# iterate through it using next()
# Output: 4
print(next(my_iter))
# Output: 7
print(next(my_iter))
# next(obj) is same as obj.__next__()
# Output: 0
print(my_iter.__next__())
# Output: 3
print(my_iter.__next__())
# This will raise error, no items left
next(my_iter)
```

```
4
7
0
3

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
C:\Users\ABBLEA~1\AppData\Local\Temp/ipykernel_8596/1932000471.py in <module>
     14 print(my_iter.__next__())
     15 # This will raise error, no items left
---> 16 next(my_iter)

StopIteration:
```

In [6]:
```
print(dir(my_iter))
```

```
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__g
e__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass_
_', '__iter__', '__le__', '__length_hint__', '__lt__', '__ne__', '__new__', '__
next__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate_
_', '__sizeof__', '__str__', '__subclasshook__']
```

In [7]:
```
#Simple for loop use in python
for city in ["Berlin", "Vienna", "Zurich"]:
    print(city)
print("\n")
for city in ("Python", "Perl", "Ruby"):
    print(city)
for char in "Iteration is easy":
    print(char, end = " ")
```

```
Berlin
Vienna
Zurich


Python
Perl
Ruby
I t e r a t i o n   i s   e a s y
```

In [8]:
```
#Code 2 : Function 'iterable' will return True, if the object 'obj' is an iterabl
# list of cities
cities = ["Berlin", "Vienna", "Zurich"]
# initialize the object
iterator_obj = iter(cities)
print(next(iterator_obj))
print(next(iterator_obj))
print(next(iterator_obj))
#Note: If 'next(iterator_obj)' is called one more time, it would return 'StopIter
```

```
Berlin
Vienna
Zurich
```

In [9]:
```
# Function to check object
# is iterable or not
def iterable(obj):
    try:
        iter(obj)
        return True

    except TypeError:
        return False

#Driver Code
for element in [34, [4, 5], (4, 5),
 {"a":4}, "dfsdf", 4.5]:

 print(element, " is iterable : ", iterable(element))
```

```
34  is iterable :  False
[4, 5]  is iterable :  True
(4, 5)  is iterable :  True
{'a': 4}  is iterable :  True
dfsdf  is iterable :  True
4.5  is iterable :  False
```

Collections

In [10]:
```
from collections import Counter
lst=[1,3,2,5,7,9,3,2,6,9]
Counter(lst)
```

Out[10]:
```
Counter({1: 1, 3: 2, 2: 2, 5: 1, 7: 1, 9: 2, 6: 1})
```

In [11]:
```
cnt = Counter(lst)
print(cnt[1])
print(cnt.most_common())
```

```
1
[(3, 2), (2, 2), (9, 2), (1, 1), (5, 1), (7, 1), (6, 1)]
```

In [12]:
```
cnt = Counter({1:3,4:2,2:2})
print(list(cnt.elements()))
```

```
[1, 1, 1, 4, 4, 2, 2]
```

In [13]:
```
deduct={1:1,4:1,2:1}
cnt.subtract(deduct)
print(cnt)
```

```
Counter({1: 2, 4: 1, 2: 1})
```

In [14]:
```
from collections import namedtuple
a = namedtuple('details','name, age')
s = a('arathi',22)
print(s)
```

```
details(name='arathi', age=22)
```

In [15]:
```
#using list to get values
s = a._make(['abcd',21])
print(s)
```

```
details(name='abcd', age=21)
```

In [16]:
```
from collections import namedtuple
a=namedtuple('courses','name,technology')
s=a('data science','python')
print(s)
```

```
courses(name='data science', technology='python')
```

In [17]:
```
itr=iter(s)
while True:
    try:
        print(next(itr))
    except StopIteration:
        break
```

```
data science
python
```

In [18]:
```
from collections import deque
lst = ['a','e','i','o','u']
dequelst = deque(lst)
print(dequelst)
```

```
deque(['a', 'e', 'i', 'o', 'u'])
```

In [19]:
```
dequelst.append('b')
print(dequelst)
```

```
deque(['a', 'e', 'i', 'o', 'u', 'b'])
```

In [20]:
```
dequelst.appendleft('z')
print(dequelst)
```

```
deque(['z', 'a', 'e', 'i', 'o', 'u', 'b'])
```

In [21]:
```
dequelst.pop() #last element deleted
print(dequelst)
```

```
deque(['z', 'a', 'e', 'i', 'o', 'u'])
```

In [22]:
```
dequelst.popleft() #frst element deleted
print(dequelst)
```

```
deque(['a', 'e', 'i', 'o', 'u'])
```

In [23]:
```
from collections import deque
a=['a','m','i','t','h','a']
d=deque(a)
print(d)
```

```
deque(['a', 'm', 'i', 't', 'h', 'a'])
```

In [24]:
```
itr=iter(a)
while True:
    try:
        print(next(itr),end=' ')
    except StopIteration:
        break
```

```
a m i t h a
```

ChainMap

In [25]:
```
from collections import ChainMap
a={1:'m',2:'n'}
b={3:'x',4:'y'}
d=ChainMap(a,b)
print(d)
```

```
ChainMap({1: 'm', 2: 'n'}, {3: 'x', 4: 'y'})
```

In [26]:
```
for i in d.items():
 print(i)
```

```
(3, 'x')
(4, 'y')
(1, 'm')
(2, 'n')
```

In [27]:
```
itr=iter(d.items())
while True:
    try:
        print(next(itr))
    except StopIteration:
        break
```

```
(3, 'x')
(4, 'y')
(1, 'm')
(2, 'n')
```

In [28]:
```
from collections import OrderedDict
d = OrderedDict()
d[1] = 'a'
d[2] = 'r'
d[3] = 'a'
d[4] = 't'
d[5] = 'h'
d[6] = 'i'
print(d)
```

```
OrderedDict([(1, 'a'), (2, 'r'), (3, 'a'), (4, 't'), (5, 'h'), (6, 'i')])
```

In [29]:
```
d[2]='a'
print(d)
```

```
OrderedDict([(1, 'a'), (2, 'a'), (3, 'a'), (4, 't'), (5, 'h'), (6, 'i')])
```

DefaultDict

In [30]:
```
from collections import defaultdict
d = defaultdict(int)
d[1]='abc'
d[2]='efg'
print(d)
print(d[3]) #returns 0 if no value for that key, whereas normal dictionary gives
```

```
defaultdict(<class 'int'>, {1: 'abc', 2: 'efg'})
0
```

In [31]:
```
from collections import defaultdict
d=defaultdict(int)
d[1]='AI'
d[2]='ML'
print(d)
```

```
defaultdict(<class 'int'>, {1: 'AI', 2: 'ML'})
```

In [32]:
```
itr=iter(d.items())
while True:
    try:
        print(next(itr))
    except StopIteration:
        break
```

```
(1, 'AI')
(2, 'ML')
```

In [ ]:
```

```