```
Iteration :

- Iteration is a general term for taking each item of something, one
after another.

- Anytime we use loop explicit or implicit, to go over a group of items,
that is called iteration
```

In [1]:
```
num = [1,2,3,4]
for i in num:
    print(i,end=' ')
```

```
1 2 3 4
```

In [ ]:
```

```

```
Iterator :
- Is an object that allows a programmer to traverse over a -- sequence
of data -- without having to store the entire data in the memory.
```

```
Note :
We can use getsizeof method of sys module to get he size the var taking
inside the memory
```

In [9]:
```
import sys
l = [i for i in range(1,500)]
for i in l:
    i**2

print('Size occupied ',sys.getsizeof(l)/1024)
```

```
Size occupied  4.1640625
```

In [ ]:
```

```

```
performing the same operation using an iterator to see the difference in
size of  memory occupied into ram
```

In [10]:
```
import sys

x  = range(1,500)
for i in x:
    i**2

print('Size occupied ',sys.getsizeof(x)/1024)
```

```
Size occupied  0.046875
```

In [ ]:
```

```

```
Iterable
- Iterable is an obj. which one can iterate over

- It generated an iterator when passed to iter() method
```

In [14]:
```
l = [1,2,3]
type(l)
# l --> is an iterable

type(iter(l))
# iter(l) --> is an list_iterator
```

Out[14]:
```
list_iterator
```

In [ ]:
```

```

```
Points to remember ::
- Every iterable is not an iterator
    [Not all iterables are iterators]

- Every iterator is also an iterable
```

In [ ]:
```

```

```
Trick :
To find out if an obj is ITERABLE or not

+ loop on the object
- Loop successful that means it's an iterable object

+ use dir(object)
- If we found and __iter__ method that means it's iterable otherwise
it's not
```

Ex 1.

In [16]:
```
# method 1

a = 2
for i in a:
    print(i)

# Loop successful that means it's an iterable object
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call l
ast)
Input In [16], in <module>
      1 a = 2
----> 2 for i in a:
      3     print(i)

TypeError: 'int' object is not iterable
```

In [17]:
```
# method 2
a = 2
print(dir(a))

# If we found and __iter__ method that means it's iterable otherwise it's
```

```
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float_
_', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattribut
e__', '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__',
'__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or_
_', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__red
uce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '_
_rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift_
_', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr_
_', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv_
_', '__trunc__', '__xor__', 'as_integer_ratio', 'bit_length', 'conjugat
e', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_byte
s']
```

In [ ]:
```

```

Ex 2.

In [18]:
```
t = (1,2,3)

for i in t:
    print(i,end=' ')

# its an iterable
```

```
1 2 3
```

In [19]:
```
t = (1,2,3)
print(dir(t))

# it consisit of __iter__ method,
# it's an iterator
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__d
oc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem
__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subcla
ss__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__', '__se
tattr__', '__sizeof__', '__str__', '__subclasshook__', 'count', 'inde
x']
```

In [ ]:
```

```

Ex 3.

In [21]:
```
dic = {'one':1,2:2}
print(dic,type(dic))
```

```
{'one': 1, 2: 2} <class 'dict'>
```

In [23]:
```
print(dir(dic))

# it consisit of __iter__ method,
# it's an iterator
```

```
['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__get
item__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter
__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__setattr__', '__setitem_
_', '__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy', 'from
keys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'updat
e', 'values']
```

In [25]:
```
for i in dic:
    print(i)
```

```
one
2
```

In [ ]:

In [ ]:

```
Trick :
To find out if an obj is ITERATOR or not

+ use dir(object)
- If we found __next__ and __iter__ method that means it's ITERATOR
otherwise it's not
```

Ex 1.

In [26]:
```
l = [1,2,3]

print(dir(l))

# only __iter__ methods is found that means
# Its an iterable
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute
__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__in
it__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr_
_', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof
__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

In [30]:
```
x = iter(l)
print(dir(x))

#  __iter__ & __next__ methods is found that means
# Its an iterator
```

```
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format_
_', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__
init_subclass__', '__iter__', '__le__', '__length_hint__', '__lt__', '_
_ne__', '__new__', '__next__', '__reduce__', '__reduce_ex__', '__repr_
_', '__setattr__', '__setstate__', '__sizeof__', '__str__', '__subclass
hook__']
```

In [ ]:

In [ ]:

Understanding how for-loop works

In [31]:
```
no = [1,2,3,4,5,6]
for i in no:
    print(i,end=' ')
```

```
1 2 3 4 5 6
```

how for loop works

In [42]:
```
no = [1,2,3,4,5,6]

# fetch the iterator
iter_no = iter(no)

print(next(iter_no))
print(next(iter_no))
print(next(iter_no))
print(next(iter_no))
print(next(iter_no))
print(next(iter_no))
print(next(iter_no))

# till it get's the error : stopIteration
```

```
1
2
3
4
5
6

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call l
ast)
Input In [42], in <module>
     10 print(next(iter_no))
     11 print(next(iter_no))
---> 12 print(next(iter_no))

StopIteration:
```

In [ ]:
```

```

Custom For Loop

In [46]:
```
def mera_for_loop(iterable):
    iter_no = iter(iterable)
    while True:
        try:
            print(next(iter_no),end=' ')
        except StopIteration:
            break
```

Now using our own for loop

In [47]:
```
a = [1,2,3,4]
b = range(1,5)
c = (6,7,8)
d = {'hello':1,2:2}
```

In [48]:
```
mera_for_loop(a)
```

```
1 2 3 4
```

In [50]:
```
mera_for_loop(b)
```

```
1 2 3 4
```

In [51]:
```
mera_for_loop(c)
```

```
6 7 8
```

In [55]:
```
mera_for_loop(d)
```

```
hello 2
```

In [54]:
```
mera_for_loop(d.items())
```

```
('hello', 1) (2, 2)
```

In [57]:
```
mera_for_loop(d.values())
```

```
1 2
```

In [58]:
```
mera_for_loop(d.keys())
```

```
hello 2
```

In [ ]:
```

```

```
Note :

When we run __iter__ over a iterable we get an iterator.
but When we again run __iter__ over a iterator we get an iterator that's
the same iterator (himself)
```

In [ ]:
```

```

Creating our own custom range fucntion

In [61]:
```
# iterable class
class MeraRange:
    def __init__(self,start,end):
        self.start = start
        self.end = end

    def __iter__(self):
        return MeraRangekaIteratoraObj(self)
```

In [62]:
```
# iterator class
class MeraRangekaIteratoraObj:
    def __init__(self,iterable_obj):
        self.iterable = iterable_obj

    def __iter__(self):
        return self

    def __next__(self):
        if self.iterable.start >= self.iterable.end:
            raise StopIteration

        current = self.iterable.start
        self.iterable.start+=1
        return current
```

In [63]:
```
for i in MeraRange(1,10):
    print(i,end=' ')
```

```
1 2 3 4 5 6 7 8 9
```

In [64]:
```
x = MeraRange(11,20)
print(type(x))
```

```
<class '__main__.MeraRange'>
```

In [65]:
```
print(iter(x))
```

```
<__main__.MeraRangekaIteratoraObj object at 0x7f93f9be4c70>
```

In [ ]:

Generators is an simple/efficient way of creating an iterators

In [ ]: