Iterators and Generators

- Iterator in python is an object that is used to iterate over iterable objects like list,tuples,dicts and sets
- Iterator object is initialised using the iter() method .It uses the next() method for iteration
- **iter**(iterable):-method that is called for initialization of an iterator.This returns an iterator obj.
- **next**() :- method returns the next value for the iterable

In [14]:

```
s=("Welcome","to","Python","programming")
for i in s:
    print(i)
```

```
Welcome
to
Python
programming
```

In [ ]:

```
# How for loop internally iterating the list of iterable objects
```

In [15]:

```
obj=iter(s)
obj
```

Out[15]:

```
<list_iterator at 0x1f199ed5c10>
```

In [16]:

```
next(obj)
```

Out[16]:

```
'Welcome'
```

In [17]:

```
next(obj)
```

Out[17]:

```
'to'
```

In [18]:

```
next(obj)
```

Out[18]:

```
'Python'
```

In [19]:

```
next(obj)
```

Out[19]:

```
'programming'
```

In [20]:

```
next(obj)
```

```
---------------------------------------------------------------------------
-
StopIteration                             Traceback (most recent call las
t)
<ipython-input-20-2e36627a780e> in <module>
----> 1 next(obj)

StopIteration:
```

In [26]:

```
p=["red","blue","green","yellow"]
obj1=reversed(p)
next(obj1)
```

Out[26]:

```
'yellow'
```

In [27]:

```
next(obj1)
```

Out[27]:

```
'green'
```

In [28]:

```
next(obj1)
```

Out[28]:

```
'blue'
```

In [29]:

```
next(obj1)
```

Out[29]:

```
'red'
```

In [31]:

```
def print_each(i):
    it=iter(i)
    while True:
        try:
            item=next(it)
        except StopIteration:
            break
        else:
            print(item)
col=['red','blue','green',"yellow"]
print_each(col)
```

```
red
blue
green
yellow
```

Generator Function

- generator fun ia a function which returns generator-iterator with the help of yield keyword

In [36]:

```
def fib(Mymax):
    a,b=0,1
    while True:
        c=a+b
        if c < Mymax:
            print("Before yield")
            yield c # program exection will be stopped
            print("After yield")
            a=b
            b=c
        else:
            break
gen=fib(4)
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
```

```
Before yield
1
After yield
Before yield
2
After yield
Before yield
3
After yield

---------------------------------------------------------------------------
-
StopIteration                             Traceback (most recent call las
t)
<ipython-input-36-64a104a5c1da> in <module>
     15 print(next(gen))
     16 print(next(gen))
---> 17 print(next(gen))

StopIteration:
```

In [42]:

```
# Example program for Iterator
class Nums():
    MAX=10
    def __init__(self):
        self.current=0
    def __iter__(self):
        return self
    def __next__(self):
        next_val=self.current
        if next_val >= self.MAX:
            raise StopIteration
        self.current+=1
        return next_val
obj=Nums()
print(next(obj))
print(next(obj))
```

```
0
1
```

In [41]:

```
# Example program for Generator
def Nums(m):
    n=0
    while n<=m:
        yield n
        n+=1
gen=Nums(5)
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
```

```
0
1
2
3
4
5
```

```
---------------------------------------------------------------------------
-
StopIteration                             Traceback (most recent call las
t)
<ipython-input-41-992a1f846dc7> in <module>
     11 print(next(gen))
     12 print(next(gen))
---> 13 print(next(gen))

StopIteration: 
```

In short Summery

- Generators allow you to create iterators
- Iterators allow Lazy evaluation only generating the next element of an iterable object when requested
- Iterators and generators can only be iterated over once.
- Generator Functions are better than iterators

In [ ]: