ITERATORS GENERATORS DECORATORS Cheat Sheet

by sahusourabh via cheatography.com/156407/cs/33300/

ITERATORS

Looping

Important functions to be implemented

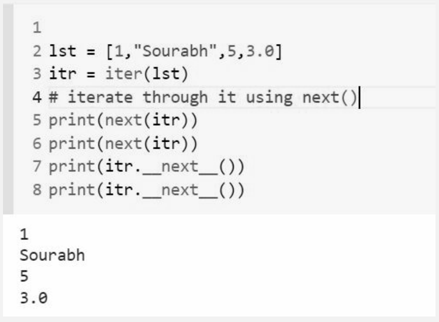
__iter__()

__next__()

__iter__() : takes iteratable object like list, tuples

__next__(): is used to return the next value in iteration

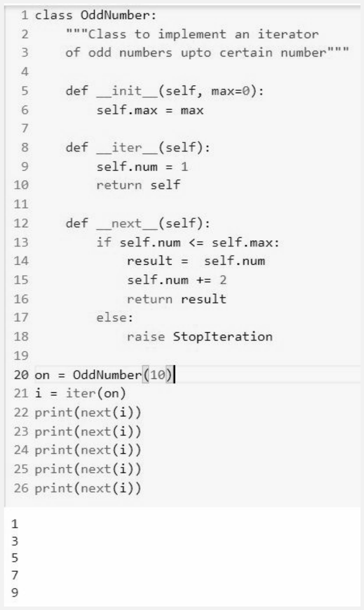
Use of iterators



```

lst = [1,"Sourabh",5,3.0]
itr = iter(lst)
# iterate through it using next()
print(next(itr))
print(next(itr))
print(itr.__next__())
print(itr.__next__())
```

```
1
Sourabh
5
3.0
```

Iterators with class



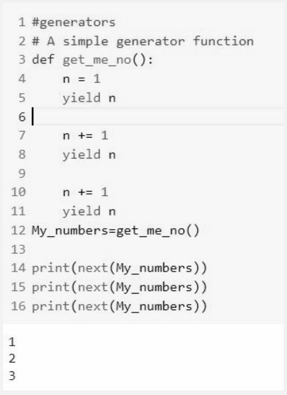
```python
class OddNumber:
    """Class to implement an iterator
    of odd numbers upto certain number"""

    def __init__(self, max=0):
        self.max = max

    def __iter__(self):
        self.num = 1
        return self

    def __next__(self):
        if self.num <= self.max:
            result =  self.num
            self.num += 2
            return result
        else:
            raise StopIteration

on = OddNumber(10)
i = iter(on)
print(next(i))
print(next(i))
print(next(i))
print(next(i))
print(next(i))
```

```
1
3
5
7
9
```

GENERATORS

1. Generator generate one element at a time from a sequence.
2. Yield is used to get the value
3 It saves the state not like function where once function is called state will be returned to new call

USE



```python
#generators
# A simple generator function
def get_me_no():
    n = 1
    yield n

    n += 1
    yield n

    n += 1
    yield n
My_numbers=get_me_no()

print(next(My_numbers))
print(next(My_numbers))
print(next(My_numbers))
```

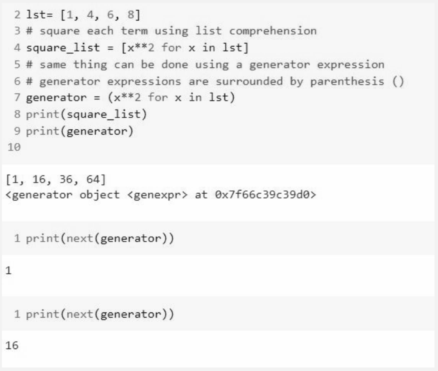
```
1
2
3
```

List Comprehension vs Generation

Comprehension: all in one go

Generation : one by one ...fast

EXAMPLE



```python
lst= [1, 4, 6, 8]
# square each term using list comprehension
square_list = [x**2 for x in lst]
# same thing can be done using a generator expression
# generator expressions are surrounded by parenthesis ()
generator = (x**2 for x in lst)
print(square_list)
print(generator)

```

```
[1, 16, 36, 64]
<generator object <genexpr> at 0x7f66c39c39d0>
```

```python
print(next(generator))
```

```
1
```

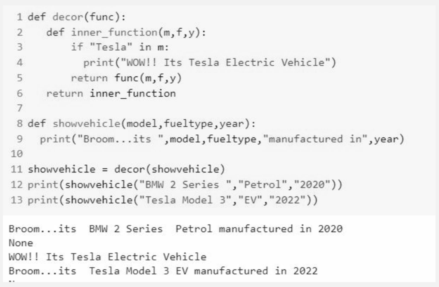
```python
print(next(generator))
```

```
16
```

DECORATORS

1. A decorator is a special function which adds some extra functionality to an existing function
2. A decorator is a function that accepts a function as a parameter and returns a function.
3. Decorators are useful to perform some additional processing required by a function.

Want to add addition functionality to function



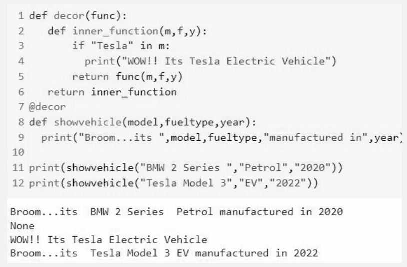
```python
def decor(func):
   def inner_function(m,f,y):
       if "Tesla" in m:
         print("WOW!! Its Tesla Electric Vehicle")
       return func(m,f,y)
   return inner_function

def showvehicle(model,fueltype,year):
  print("Broom...its ",model,fueltype,"manufactured in",year)

showvehicle = decor(showvehicle)
print(showvehicle("BMW 2 Series ","Petrol","2020"))
print(showvehicle("Tesla Model 3","EV","2022"))
```

```
Broom...its  BMW 2 Series  Petrol manufactured in 2020
None
WOW!! Its Tesla Electric Vehicle
Broom...its  Tesla Model 3 EV manufactured in 2022
```

@decor

showvehicle = decor(showvehicle) instead of this line, you can use @decor

@decor implementation



```python
def decor(func):
   def inner_function(m,f,y):
       if "Tesla" in m:
         print("WOW!! Its Tesla Electric Vehicle")
       return func(m,f,y)
   return inner_function
@decor
def showvehicle(model,fueltype,year):
  print("Broom...its ",model,fueltype,"manufactured in",year)

print(showvehicle("BMW 2 Series ","Petrol","2020"))
print(showvehicle("Tesla Model 3","EV","2022"))
```

```
Broom...its  BMW 2 Series  Petrol manufactured in 2020
None
WOW!! Its Tesla Electric Vehicle
Broom...its  Tesla Model 3 EV manufactured in 2022
```