Function Decorator

Decorators

- Functions and methods are called callable as they can be called.
- In fact, any object which implements the special method __call__() is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable.
- Basically, a decorator takes in a function, adds some functionality and returns it.

Simple Nested Function

```
def is_called():
    def is_returned():
        print("Hello")
    return is_returned

new = is_called()
new()
```

Example 2:

- In the example shown above, make_pretty() is a decorator. In the assignment step.
- pretty = make_pretty(ordinary)
- The function ordinary() got decorated and the returned function was given the name pretty.
- We can see that the decorator function added some new functionality to the original function. This is similar to packing a gift. The decorator acts as a wrapper. The nature of the object that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).

Example 2:

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner

def ordinary():
    print("I am ordinary")

#ordinary()
pretty=make_pretty(ordinary)
pretty()
```

Example 2:

- Generally, we decorate a function and reassign it as,
- *ordinary = make_pretty(ordinary).*
- This is a common construct and for this reason, Python has a syntax to simplify this.
- We can use the @ symbol along with the name of the decorator function and place it above the definition of the function to be decorated. For example,

```
@make_pretty
def ordinary():
    print("I am ordinary")
is equivalent to
def ordinary():
    print("I am ordinary")
ordinary = make_pretty(ordinary)
```

Decorator Ex

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner
@make_pretty
def ordinary():
    print("I am ordinary")
ordinary()
#pretty=make_pretty(ordinary)
#pretty()
```

Generators in Python

- Generators are a special kind of function, which enable us to implement or generate iterators.
- Iterators are a fundamental concept of Python.Mostly, iterators are implicitly used, like in the for loop of Python. A generator is a function that produces a sequence of results instead of a single value.
- Generators are a simple and powerful possibility to create or to generate iterators. These iterators are called generator objects. On the surface they look like functions, but there is both a syntactic and a semantic difference. Instead of return statements you will find inside of the body of a generator only yield statements, i.e. one or more yield statements.
- A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. If the body of a def contains yield, the function automatically becomes a generator function.
- A more practical type of stream processing is handling large data files such as log files. Generators provide a space efficient method for such data processing as only parts of the file are handled at one given point in time.

Iterator

Iterator in python is any python type that can be used with a ‘for in loop’. Python lists, tuples, dicts and sets are all examples of inbuilt iterators. These types are iterators because they implement following methods. In fact, any object that wants to be an iterator must implement following methods.

__iter__ method that is called on initialization of an iterator. This should return an object that has a next or __next__ (in Python 3) method.

next (__next__ in Python 3) The iterator next method should return the next value for the iterable. When an iterator is used with a ‘for in’ loop, the for loop implicitly calls next() on the iterator object. This method should raise a StopIteration to signal the end of the iteration.

Iterator or Generator

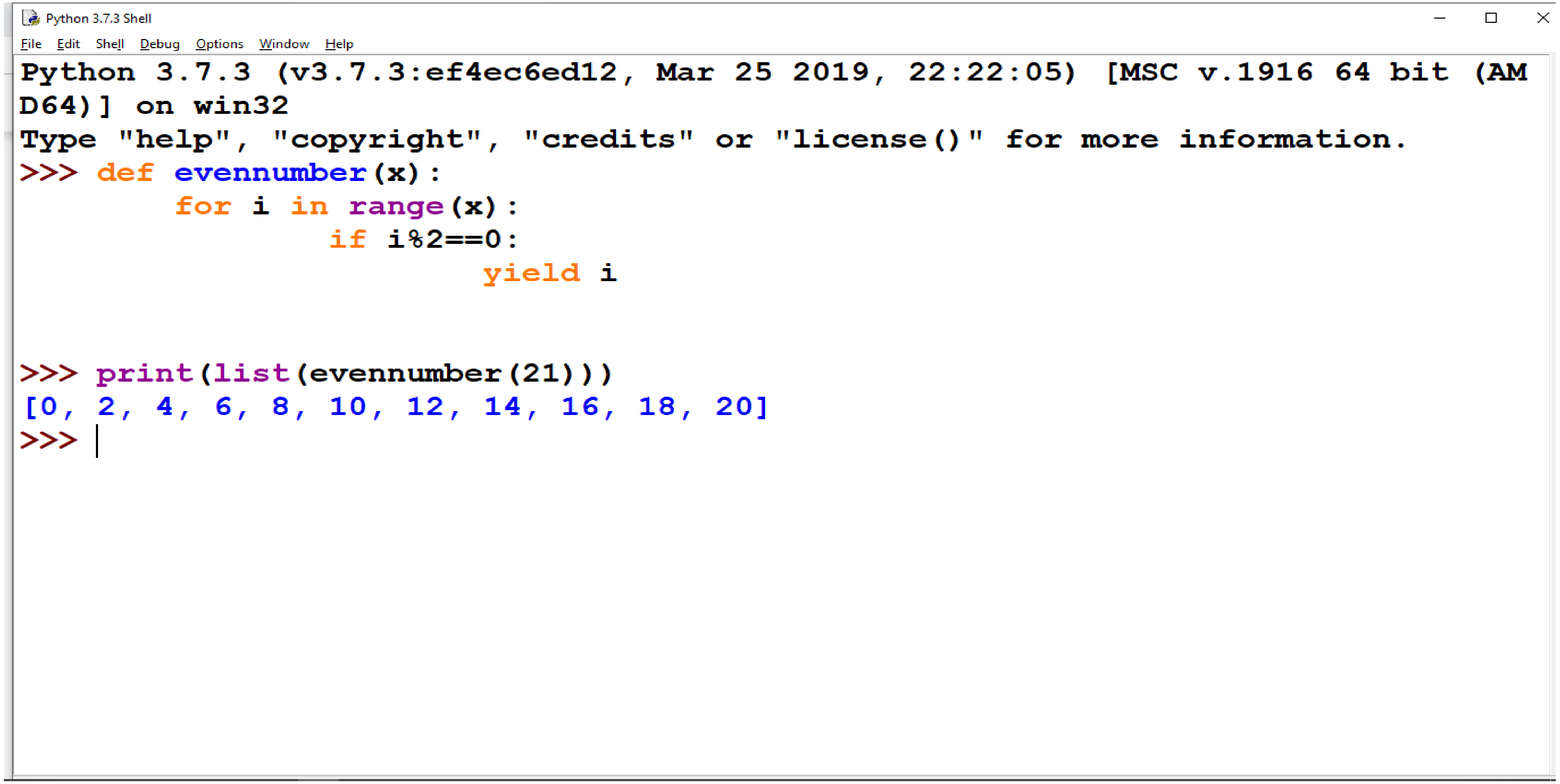
```
list_string = ['Boy 1', 'Boy 2', 'Boy 3', 'Boy 4',
'Boy 5']

iterator_you = iter(list_string)

# point the first boy

output = next(iterator_you)

# This will print 'Boy 1'

print(output)

# point the first boy

output = next(iterator_you)

# This will print 'Boy 1'

print(output)

# point the next boy, the third boy

output = next(iterator_you)

# This will print 'Boy 3'

print(output)

# point the next boy, the fifth boy

output = next(iterator_you)

# This will print 'Boy 5'

print(output)

# point the next boy, the fifth boy

output = next(iterator_you)

# This will print 'Boy 5'

print(output)

# point the next boy, but there is no boy left

# so raise 'StopIteration' exception

output = next(iterator_you)

# This print will not execute

#print(output)
```

Yield

The yield statement suspends function’s execution and sends a value back to caller, but retains enough state to enable function to resume where it is left off. When resumed, the function continues execution immediately after the last yield run. This allows its code to produce a series of values over time, rather them computing them at once and sending them back like a list.

Return sends a specified value back to its caller whereas Yield can produce a sequence of values. We should use yield when we want to iterate over a sequence, but don’t want to store the entire sequence in memory.

Example 1 for Generator



```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> def evennumber(x):
        for i in range(x):
                if i%2==0:
                        yield i


>>> print(list(evennumber(21)))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>>> 
```

Yield EX:

```
# A Python program to demonstrate use of
# generator object with next()
 # A generator function
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3
# x is a generator object
x = simpleGeneratorFun()
# Iterating over the generator object using next
print(x.__next__());
print(x.__next__());
print(x.__next__());
```