ITERATORS

In computer science, an **iterator** is an object that allows a programmer to traverse through all the elements of a collection regardless of its specific implementation.

Iterators in Python are a fundamental part of the language and in many cases go unseen as they are implicitly used in forstatement, in **list comprehensions**, and in **generator expressions**

All of Python's standard built-in sequence types support iteration, as well as many classes which are part of the standard library

The **for** loop works on any **iterable object**. Actually, this is true of all iteration tools that scan objects from left to right in Python including **for** loops, the **list comprehensions**, and the **map** built-in function, etc.

An object is **iterable** if it is either a physically stored sequence or an object that produces one result at a time in the context of an iteration tool like **for** loop.

File Iterators

Keys of a Dictionary

os.popen ()

range ()

enumerate ()

CLOSURES

A function defined inside another function is called a nested function. Nested functions can access variables of the enclosing scope.

In Python, these non-local variables are read only by default and we must declare them explicitly as non-local (using nonlocal keyword) in order to modify them

def print\_msg(msg):

"""This is the outer enclosing function"""

def printer():

"""This is the nested function"""

print (msg)

printer()

We execute the function as follows.

>>> print\_msg("Hello")

Hello

We can see that the nested function **printer ()** was able to access the non-local variable msg of the enclosing function.

**Another Way:**

def print\_msg(msg):

"""This is the outer enclosing function"""

def printer():

"""This is the nested function"""

print(msg)

return printer # this got changed

Now let's try calling this function.

>>> another = print\_msg("Hello")

>>> another()

Hello

That's unusual. The print\_msg() function was called with the string "Hello" and the returned function was bound to the name another. On calling another(), the message was still remembered although we had already finished executing the print\_msg() function. This technique by which some data ("Hello") gets attached to the code is called closure in Python.

This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.

>>> del print\_msg

>>> another()

Hello

>>> print\_msg("Hello")

Traceback (most recent call last):

...

NameError: name 'print\_msg' is not defined

The criteria that must be met to create closure in Python are summarized in the following points.

* We must have a nested function (function inside a function).
* The nested function must refer to a value defined in the enclosing function.
* The enclosing function must return the nested function.

Closures can avoid the use of global values and provides some form of data hiding.

GENERATOR EXPRESSIONS AND GENERATORS

In computer science, a **generator** is a special routine that can be used to control the iteration behavior of a loop

A generator is very similar to a function that returns an array, in that a generator has parameters, can be called, and generates a sequence of values. However, instead of building an array containing all the values and returning them all at once, a generator **yields** the values one at a time, which requires less memory and allows the caller to get started processing the first few values immediately. In short, a generator **looks like a function** but **behaves like an iterator**.

Python provides tools that produce results only when needed:

•Generator functions

They are coded as normal def but use yield to return results one at a time, suspending and resuming.

•Generator expressions

These are similar to the list comprehensions. But they return an object that produces results on demand instead of building a result list.

Because neither of them constructs a result list all at once, they save memory space and allow computation time to be split by implementing the iteration protocol.

**GENERATOR FUNCTIONS:**

**Example:**

>>> def create\_counter(n):

print('create\_counter()')

while True:

yield n

print('increment n')

n += 1

>>> c = create\_counter(2)

>>> c

<generator object create\_counter at 0x03004B48>

>>> c.next()

create\_counter()

2

>>> c.next()

increment n

3

>>> c.next()

increment n

4

>>>

**Another Example:**

>>> def cubic\_generator(n):

for i in range(n):

yield i \*\* 3

>>>

**GENERATOR EXPRESSIONS:**

>>> # List comprehension makes a list

>>>a=[x \*\* 3 for x in range (5)]

>>> print a

[0, 1, 8, 27, 64]

>>>

>>> # Generator expression makes an iterable

>>> a=(x \*\* 3 for x in range (5))

>>> print a

<generator object <genexpr> at 0x000000000315F678>

>>> a.next ()

**Generator: A Single-Iterator Object**

Both generator functions and generator expressions are their own iterators. So, they support just one active iteration. We can't have multiple iterators

This is different from the behavior of some built-in types. Built-in types support multiple iterators and passes and reflect their in-place changes in active iterators:

>>>

>>> L = [1, 2, 3, 4]

>>> I1, I2 = iter(L), iter(L)

>>> next(I1)

1

>>> next(I1)

2

>>> # Lists support multiple iterators

>>> next(I2)

1

>>> # Changes reflected in iterators

>>> del L[2:]

>>> next(I1)

Traceback (most recent call last):

File "<pyshell#21>", line 1, in <module>

next(I1)

StopIteration

>>>

DECORATORS

**Assign functions to variables:**

def xyz(name):

return "hello "+name

r=xyz(“World”)

print r

# Outputs: hello World

**Define functions inside other functions:**

def xyz (name):

def abc ():

return "Hello "

r= abc()+name

return r

print xyz(“world”)

# Outputs: Hello World

**Functions can be passed as parameters to other functions:**

def abc(name):

return "Hello " + name

def xyz(func):

r=func(“Prasad”)

return r

print xyz(abc)

# Outputs: Hello Prasad

**Functions can return other functions:**

In other words, functions generating other functions.

def xyz():

def abc():

return "Hello there!"

return abc

r=xyz()

print r()

# Outputs: Hello there

**Composition of Decorators:**

Function decorators are simply wrappers to existing functions. In this example let's consider a function that wraps the string output of another function by p tags

def get\_text(name):

return ‘Your name is %s’ % (name)

def p\_decorate(func):

def func\_wrapper(name):

return "Hello!!! “ + (func(name))

return func\_wrapper

my\_get\_text = p\_decorate(get\_text)

print my\_get\_text("Prasad")

# Hello !!! Your name is Prasad

That was our first decorator. A function that takes another function as an argument, generates a new function, augmenting the work of the original function, and returning the generated function.

**Another way:**

def p\_decorate(func):

def func\_wrapper(name):

return "Hello!!! “ + (func(name))

return func\_wrapper

@p\_decorate

def get\_text(name):

return ‘Your name is %s’ % (name)

print get\_text("Prasad")

**Decorating Methods:**

In Python, methods are functions that expect their first parameter to be a reference to the current object. We can build decorators for methods the same way, while taking self into consideration in the wrapper function.

def p\_decorate(func):

def func\_wrapper(self):

return "<p>{0}</p>".format(func(self))

return func\_wrapper

class Person(object):

def \_\_init\_\_(self):

self.name = "John"

self.family = "Doe"

@p\_decorate

def get\_fullname(self):

return self.name+" "+self.family

my\_person = Person()

print my\_person.get\_fullname()

**Passing arguments to decorators:**

def tags(tag\_name):

def tags\_decorator(func):

def func\_wrapper(name):

return "<{0}>{1}</{0}>".format(tag\_name, func(name))

return func\_wrapper

return tags\_decorator

@tags("p")

def get\_text(name):

return "Hello "+name

print get\_text("John")

# Outputs <p>Hello John</p>