***Iterators, Generators and Decorators***

These are all advanced topic and I totally struggled to explain the topics in wordings. It might be much easier to explain directly so please bear with me. I tried best here and feel free reach me out back if any questions

It is totally about implementing new patterns and programming designs to make better framework and so please feel free to ask me anytime if you have any questions

Before going to this topic, we need to visit one of the basic concept in python called **sequence**

**Sequence**

Sequence is the generic term used for ordered list in python. There are many and following 3 are important

1. List
2. Tuple
3. Strings

List is mutable one and generally it is represented using square brackets [ ]

Tuple is an immutable list and generally it is represented using curve brackets ( )

Strings in an immutable list of characters

There some operators works on all 3 of them and gives same/similar effect,

1. Plus (+) sign combines 2 sequences and return combined sequence
2. Asterisk (\*) repeats the sequence for given positive number of time
3. “in” operator validates whether given element is in the sequence. For example, Ram in Ramkumar. It should return true.
4. All the sequence’s character can be read using square brackets with zero-based index. For example, mylist[2]
5. Reverse indexing also supported in all type of sequence
6. Slice works on all the sequence [if you need to understand slice, refer the notes document from github page]
7. len() function returns length of the sequence
8. min() and max() functions return min and max element from the sequence
9. index(character) will return the index of first occurrence of the character in sequence
10. count(character) will return how many times that particular character comes in the sequence

Let’s see one example program with results for the same.

varstring = "this is my string"

varlist = ["a", "b", "c", "d", "e", "f", "g", "h"]

vartuple = ("a", "b", "c", "d", "e", "f", "g", "h")

# 1)    Plus (+) sign combines 2 sequences and return combined sequence

print "-" \* 100

print "\n String concatenation : "

print varstring + varstring

print "\n List concatenation : "

print varlist + varlist

print "\n Tuple concatenation : "

print vartuple + vartuple

print "-" \* 100

print "\n"

# 1.1) can we combine different type of sequences? No. If you would like to try, uncomment below code and try

# print "mixed sequence concatenation : "

# print vartuple + varlist

# 2)    Asterisk (\*) repeats the sequence for given positive number of time

print "-" \* 100

print "\n repeat the string for 5 times and print"

print varstring \* 5

print "\n repeat the list for 5 times and print"

print varlist \* 5

print "\n repeat the tuple for 5 times and print"

print vartuple \* 5

print "-" \* 100

print "\n"

# 3)    "in" operator validates whether given element is in the sequence. For example, Ram in Ramkumar. It should return true.

print "-" \* 100

print "\n string 'in validation'"

print "string" in varstring

print "Ram" in varstring

print "\n list 'in validation'"

print "a" in varlist

print "m" in varlist

print "\n tuple 'in validation'"

print "g" in vartuple

print "m" in vartuple

print "-" \* 100

print "\n"

# 4)    All the sequence's character can be read using square brackets with zero-based index. For example, mylist[2]

print "-" \* 100

print "\n read 3rd character in string"

print varstring[2]

print "\n read 3rd character in list"

print varlist[2]

print "\n read 3rd character in tuple"

print vartuple[2]

print "-" \* 100

print "\n"

# 5)    Reverse indexing also supported in all type of sequence

print "-" \* 100

print "\n read 2nd character from last in string"

print varstring[-2]

print "\n read 2nd character from last in list"

print varlist[-2]

print "\n read 2nd character from last in tuple"

print vartuple[-2]

print "-" \* 100

print "\n"

# 6)    Slice works on all the sequence [if you need to understand slice, refer the notes document from github page]

print "-" \* 100

print "\n slice the string"

print varstring[5:11]

print "\n slice the list"

print varlist[3:5]

print "\n slice the tuple"

print vartuple[2:4]

print "-" \* 100

print "\n"

# 7)    len() function returns length of the sequence

print "-" \* 100

print "\n Length of the string"

print len(varstring)

print "\n Length of the list"

print len(varlist)

print "\n Length of the tuple"

print len(vartuple)

print "-" \* 100

print "\n"

# 8)    min() and max() functions return min and max element from the sequence

print "-" \* 100

print "\n min and max of the string"

print min(varstring)

print max(varstring)

print "\n min and max of the list"

print min(varlist)

print max(varlist)

print "\n min and max of the tuple"

print min(vartuple)

print max(vartuple)

print "-" \* 100

print "\n"

# 9)    index(character) will return the index of first occurrence of the character in sequence

print "-" \* 100

print "\n index of specific character in the string"

print varstring.index("m")

print "\n index of specific character in the list"

print varlist.index("d")

print "\n index of specific character in the tuple"

print vartuple.index("e")

print "-" \* 100

print "\n"

# 10)   count(character) will return how many times that particular character comes in the sequence

print "-" \* 100

print "\n count of specific character in the string"

print varstring.count("s")

print "\n count of specific character in the list"

print varlist.count("e")

print "\n count of specific character in the tuple"

print vartuple.count("z")

print "-" \* 100

print "\n"

and output will be,

----------------------------------------------------------------------------------------------------

String concatenation :

this is my stringthis is my string

List concatenation :

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

Tuple concatenation :

('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h')

----------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------

repeat the string for 5 times and print

this is my stringthis is my stringthis is my stringthis is my stringthis is my string

repeat the list for 5 times and print

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

repeat the tuple for 5 times and print

('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h')

----------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------

string 'in validation'

True

False

list 'in validation'

True

False

tuple 'in validation'

True

False

----------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------

read 3rd character in string

i

read 3rd character in list

c

read 3rd character in tuple

c

----------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------

read 2nd character from last in string

n

read 2nd character from last in list

g

read 2nd character from last in tuple

g

----------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------

slice the string

is my

slice the list

['d', 'e']

slice the tuple

('c', 'd')

----------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------

Length of the string

17

Length of the list

8

Length of the tuple

8

----------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------

min and max of the string

y

min and max of the list

a

h

min and max of the tuple

a

h

----------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------

index of specific character in the string

8

index of specific character in the list

3

index of specific character in the tuple

4

----------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------

count of specific character in the string

3

count of specific character in the list

1

count of specific character in the tuple

0

----------------------------------------------------------------------------------------------------

Now we are good with sequence and will proceed to Iterators

In any programming language, at advanced level, we generally create new design patterns to make some of the data structure process easy or customized to the project need. Generally people who creates frameworks may end up in the need which is cannot be (or not easily) handled by basic functions available in language.

In other words implementing the data structure like list, queue, stack etc

***Iterators***

Before going to iterator’s design, let’s try to understand what it is?

*We use for statement for looping over a list,*

for i in [1, 2, 3, 4]:

print i

*If we use it with a string, it loops over its characters.*

for c in "python":

print c

*If we use it with a dictionary, it loops over its keys.*

for k in {"x": 1, "y": 2}:

print k

*If we use it with a file, it loops over lines of the file.*

for line in open("a.txt"):

print line

So there are many types of objects which can be used with for loop. These are called ***iterable objects.***

There are many functions which consume these iterables. For example,

",".join(["a", "b", "c"])

***The Iteration Protocol***

There is a built-in function **iter** takes an iterable object and returns an iterator.

x = iter([1, 2, 3])

print x.next()

print x.next()

Output is,

1

2

Let’s say we are going to re-implement similar iterator design. As part of iterator protocol, we should have 2 functions called \_\_iter\_\_ and next function.

\_\_iter\_\_ - which returns the iterator object itself

next - returns next element in sequence, if there is no more items to return then it should raise *StopIteration* exception.

class my\_iterator:

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

self.i = 0

self.n = n

def \_\_iter\_\_(self):

return self

def next(self):

if self.i < self.n:

i = self.i

self.i += 1

return i

else:

raise StopIteration()

myobj = my\_iterator(5)

print myobj.next()

print myobj.next()

print myobj.next()

Output is,

0

1

2

***Generators***

Generators simplifies creation of iterators. A generator is a function that produces a sequence of results instead of a single value.

Let’s see the example of below function,

def generator\_function(n):

i = 0

while i < n:

yield i

i += 1

y = generator\_function(3)

print y

print y.next()

print y.next()

print y.next()

Each time the ***yield*** statement is executed the function generates a new value and output will be like,

<generator object generator\_function at 0x027FD490>

0

1

2

How exactly this function works? As it is advanced topics, I would like you to do a little research here and explain me how?

There is an another example here, which would give you some thinking over it,

def foo():

print "begin"

for i in range(3):

print "before yield", i

yield i

print "after yield", i

print "end"

f = foo()

print f.next()

print f.next()

print f.next()

print f.next()

and output will be like,

begin

before yield 0

0

after yield 0

before yield 1

1

after yield 1

before yield 2

2

after yield 2

end

Traceback (most recent call last):

File "c:\Users\rbangaru\Documents\BitBucketRepos\Draft\python\_draft\iterator.py", line 13, in <module>

print f.next()

StopIteration

OK one another example here, in the below function, every time you call, it returns next value in the integer etc

def integers():

"""Infinite sequence of integers."""

i = 1

while True:

yield i

i = i + 1

def squares():

for i in integers():

yield i \* i

y = integers()

print y.next()

print y.next()

z = squares()

print z.next()

print z.next()

Output is,

1

2

1

4

Ok. Here I have given enough example, please think about how these functions working and we will discuss once you have went through it.

***Generator Expressions***

Generator Expressions are generator version of list comprehensions. They look like list comprehensions, but returns a generator back instead of a list.

print sum(x\*x for x in range(10))

Output is,

285

***Itertools***

This is another standard library which was implemented with patterns and it might help you to understand why we need it.

The itertools module in the standard library provides lot of interesting tools to work with iterators.

Let’s look at some of the interesting functions.

***chain*** – chains multiple iterators together.

import itertools

it1 = iter([1, 2, 3])

it2 = iter([4, 5, 6])

print list(itertools.chain(it1, it2))

Output is,

[1, 2, 3, 4, 5, 6]

***Product -*** list of product of input lists

import itertools

listvar1 = [1, 2, 3, 4, 5, 6]

listvar2 = [11, 12, 13, 14, 15, 16]

print list(itertools.product(listvar1, listvar2))

Output is,

[(1, 11), (1, 12), (1, 13), (1, 14), (1, 15), (1, 16), (2, 11), (2, 12), (2, 13), (2, 14), (2, 15), (2, 16), (3, 11), (3, 12), (3, 13), (3, 14), (3, 15), (3, 16), (4, 11), (4, 12), (4, 13), (4, 14), (4, 15), (4, 16), (5, 11), (5, 12), (5, 13), (5, 14), (5, 15), (5, 16), (6, 11), (6, 12), (6, 13), (6, 14), (6, 15), (6, 16)]

***Next is Closures***

Closure is nothing but a just function inside the function to remove the code duplication. [Main function would act like class here]I don’t have the proper example real time scenario with me to explain better, however here is the simple example for you,

In the following example we create a simple closure for adding numbers.

def add\_number(num):

def adder(number):

'adder is a closure'

return num + number

return adder

a\_10 = add\_number(10)

print a\_10(21)

a\_5 = add\_number(5)

print a\_5(8)

Output is,

31

13

***adder*** is a closure which adds a given number to a pre-defined one.

***Decorators***

Similar to closure, Decorator is way to dynamically add some new behavior to some objects.

Let’s say, we have function called as pretty print and which prints the message and we will have decorator function to prints line with # symbol and ends with line created using # symbol

def my\_decorator(func):

def wrapper(\*args, \*\*kwargs):

print("###############################")

result = func(\*args, \*\*kwargs)

print("###############################")

return wrapper

@my\_decorator

def pretty\_print(myword):

print myword

pretty\_print("Ram")

In the above example, when we call pretty\_print function with word “Ram”. As we have generator assigned to this function (assigned with @ sign), the control goes to decorator which prints # symbolled line, calls the pretty print function and again print # symbolled line.

So output will be,

###############################

Ram

###############################

That’s all we have today. Seems so lengthy not so clear document to me and am extremely sorry about it. But I strongly believe these concepts helps in future when you end up creating some framework like stuff.

Feel free to reach me in case of any doubts.

Note: Try below functionalities which is available in python which is also uses the topics we discussed above,

<https://docs.python.org/2/library/functions.html#zip>

<https://docs.python.org/2/library/functions.html#xrange>