CH7: Functional programming tools

Python Function programming tools mainly aim to manipulation a sequence in an easier way. It’s a way to transform a sequence to another sequence or generate a single data from a sequence. Python provides some functions which enable a functional approach to programming. These functions are all convenience which can be written in Python easily. Functional programming is all about expressions. Thus the Functional programming is an expression oriented programming.

Expression oriented functions of python provides are:

* list comprehension
* map(aFunction, aSequence) – Convert a sequence into another after applying some func
* filter(aFunction, aSequence) - Filter a sequence based on a function
* reduce(aFunction, aSequence) – Generate a result from a sequence after applying a func

**7.1 List Comprehensions**

List comprehensions provide a concise way to create lists. When we want to create a list from another list after doing some operation, we can do that using for loop. List compression provides another way to do that more concisely. For example, if we want to create a list of squares, we can do that as below:

squares = []

for x in range(10):

squares.append(x\*\*2)

print squares #[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

We can achieve same result by using list compression as this.

squares = [x\*\*2 for x in range(10)]

We can even do list compression, which involve two loops: for example,

lst =[]

for i in [1,2,3,4,5]:

for j in [1,2,3,4,5]:

lst.append(i\*j)

print lst

can be compressed as

print [ i\*j for in in [1,2,3,4,5] for j in [1,2,3,4,5]]

Here, we can compact two for loops in a single statements.

We can also compact for loops and if else block into list compression. For example, given two list find out all pairs not having same elements.

lst =[]

for i in [1,2,3,4,5]:

for j in [1,2,3,4,5]:

if i ! = j:

lst.append(i\*j)

print lst

The code sing list compression is like this;

print [ i\*j for in in [1,2,3,4,5] for j in [1,2,3,4,5] if i ! =j ]

Some more examples are as below:

**Transposing a Matrix**

The following list comprehension will transpose rows and columns:

matrix = [

[1, 2, 3, 4],

[5, 6, 7, 8],

[9, 10, 11, 12],

]

>>>[[row[i] for row in matrix] for i in range(4)]

[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

this example is equivalent to:

transposed = []

for i in range(4):

transposed.append([row[i] for row in matrix])

print transposed

[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

**7. 2 Map:**  As the name says, this a function which can convert a sequence into another sequence after applying some function to each member of the input sequence.

Let’s start with an example: “**Given a list of numbers, fund the square of each number”.** We can do it in two ways as below:

Way 1: using for loop

items = [1, 2, 3, 4, 5]

squared = []

for x in items:

squared.append(x \*\* 2)

print squared

**Output:**

[1, 4, 9, 16, 25]

Way2: Using List compression:

items = [1, 2, 3, 4, 5]

print [ x\*\*2 for x in items]

**Output:**

[1, 4, 9, 16, 25]

This kind of operation is very common in real life word. For example, find out cumulative marks for each student in all semester in B-Tech exam. Here we can write a function to calculate cumulative marks for a student and apply the same formula for all students. Hence, the common thing, we do with list of student (and other sequences) is applying an operation to each item and collect the result.

Since this is such a common operation, python offers a build-in function to do that. The map(aFunction, aSequence) function applies a passed-in function to each item in a sequence and returns a list containing all the function call results.

The Map version of the above code is :

**>>> items = [1, 2, 3, 4, 5]**

**>>> def sqr(x): return x \*\* 2**

**>>> list(map(sqr, items)) # Apply sqr function to each element of items**

**[1, 4, 9, 16, 25]**

We can also use lambda function to make it more comprehensive -

**>>> list(map((lambda x: x \*\*2), items)) # [1, 4, 9, 16, 25]**

The map function is doing nothing but executing the following logic:

**>>> def mymap(aFunc, aSeq):**

**result = []**

**for x in aSeq: result.append(aFunc(x))**

**return result**

**>>> mymap(sqr, [1, 2, 3]) # [1, 4, 9]**

Another example to **finding out the power of 3 for each item in this list [1,2,3,4,5]** is:

**>>> list(map((lambda x: pow(x,3), items)) # [1, 4, 9, 16, 25]**

Now, Let’s make some changes in our statement: **“Find out the power of <3,4,5,6> for <1,2,3,4> respectively**”. That means output should be <pow(1,3), pow(2,4), pow(3,5), pow(4,6)>.

As you can see, basically there are 3 input, one pow function (which takes too parameters) and two list (power-list and item list). Fortunately, python map function can handle this kind of situation as well.

The code is

**>>> list(map(pow,[3, 4, 5, 6], [1, 2, 3, 4])) # result [3, 16, 125, 1296]**

As in the example above, with multiple sequences, map() expects an N-argument function for N sequences. In the example, pow function takes two arguments on each call.

The map call is similar to the list comprehension expression. But map applies a function call to each item instead of an arbitrary expression (we can have some more if –else block). Because of this limitation, it is somewhat less general tool. In some cases, however, map may be faster to run than a list comprehension such as when mapping a built-in function. And map requires less coding.

A special Case: What happen if input function in the map is None? What is the output of the following code?

**>>> x = [1,2,3]**

**>>> y = [1,4,9]**

**>>> new\_tuple = map(None, x, y) # output ?**

**It will return => [(1, 1), (2, 4), (3, 9)]**

If function is None, the identity function is assumed and it will return a list of tuple as in input.

**7.3 FILTER**: As the name suggests filter is used for filleting out some value from a list of values based on a Boolean function.

As an example, the following filter call picks out all even numbers:

**>>> list( filter((lambda x: x %2 == 0), [1,2,3,4,4,5,6,76,7,8,8,9,0]))**

A filter function does noting but the following code segment:

**result = []**

**for x in [1,2,3,4,4,5,6,76,7,8,8,9,0]:**

**if x % 2 == 0:**

**result.append(x)**

**7.4 Reduce Function: As** the same suggest, it will reduce a sequence into a single value. It will take a function (which takes two input) and a sequence and apply the function on the sequence several times and return a single result at end.

For example, following reduce function return the multiplication of each number:

**>>> reduce( (lambda x, y: x \* y), [1, 2, 3, 4] )**

**24**

**>>> reduce( (lambda x, y: x / y), [1, 2, 3, 4] )**

**0.041666666666666664**

**>>>**

Basically, It operates in the following way. First it take first two number from the sequence and get the result. Then it took this result and 3rnd number, apply the function again and find out the result and so on until all the list is reduced to single result - As it is reducing the list to single object, it is called reduce.

The equivalent for loop version is as below:

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

**>>> result = L[0]**

**>>> for x in L[1:]:**

**result = result \* x**

**>>> result**

**24**

Let's make our own version of reduce.

**>>> def myreduce(fnc, seq):**

**tally = seq[0]**

**for next in seq[1:]:**

**tally = fnc(tally, next)**

**return tally**

**>>> myreduce( (lambda x, y: x \* y), [1, 2, 3, 4])**

**24**

Some more Example of reduce function is as below:

**Example 1: String Concatenation**

**L = ['Testing ', 'shows ', 'the ', 'presence', ', ','not ', 'the ', 'absence ', 'of ', 'bugs']**

**functools.reduce( (lambda x,y:x+y), L)**

**'Testing shows the presence, not the absence of bugs'**

**Example2: Making a List unique through functional paradigm?**

Input : [1,2,2,3,3,3,4] **and** Output: [1,2,3,4] (In order preserving manner)

If you need to just delete adjacent occurrences try this:

**reduce(lambda x,y: x+[y] if x==[] or x[-1] != y else x, your\_list,[])**

If you need to delete all but one occurrence try this:

**reduce(lambda x,y: x+[y] if not y in x else x, your\_list,[])**