23.1. Introduction: Map, Filter, List Comprehensions, and Zip

Let’s revisit the [accumulator pattern](https://fopp.umsi.education/runestone/static/fopp/Iteration/TheAccumulatorPattern.html#accum-pattern). We have frequently taken a list and produced another list from it that contains either a subset of the items or a transformed version of each item. When each item is transformed we say that the operation is a **mapping, or just a map** of the original list. When some items are omitted, we call it a **filter**.

Python provides built-in functions map and filter. Python also provides a new syntax, called **list comprehensions**, that lets you express a mapping and/or filtering operation. Just as with named functions and lambda expressions, some students seem to find it easier to think in terms of the map and filter functions, while other students find it easier to read and write list comprehensions. You’ll learn both ways; one may even help you understand the other. Most python programmers use list comprehensions, so make sure you learn to read those. In this course, you can choose to learn to write list comprehensions or to use map and filter, whichever you prefer. You should learn to read both list comprehensions and map/filter.

Other common accumulator patterns on lists aggregate all the values into a single value.

Map, and filter are commands that you would use in high-performance computing on big datasets. See [MapReduce on Wikipedia](http://en.wikipedia.org/wiki/MapReduce).

# 23.2. Map

You previously were introduced to accumulating a list by transforming each of the elements. Here we revisit that pattern.

The following function produces a new list with each item in the original list doubled. It is an example of a mapping, from the original list to a new list of the same length, where each element is doubled.

def doubleStuff(a\_list):

""" Return a new list in which contains doubles of the elements in a\_list. """

new\_list = []

for value in a\_list:

new\_elem = 2 \* value

new\_list.append(new\_elem)

return new\_list

​

things = [2, 5, 9]

print(things)

things = doubleStuff(things)

print(things)

​

The doubleStuff function is an example of the accumulator pattern, in particular the mapping pattern. On line 3, new\_list is initialized. On line 5, the doubled value for the current item is produced and on line 6 it is appended to the list we’re accumulating. Line 7 executes after we’ve processed all the items in the original list: it returns the new\_list. Once again, codelens helps us to see the actual references and objects as they are passed and returned.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Python 3.3   |  |  |  | | --- | --- | --- | |  | 1 | def doubleStuff(a\_list): | | 2 | """ Return a new list in which contains doubles of the elements in a\_list. """ | | 3 | new\_list = [] | | 4 | for value in a\_list: | | 5 | new\_elem = 2 \* value | | 6 | new\_list.append(new\_elem) | | 7 | return new\_list | | 8 |  | | 9 | things = [2, 5, 9] | | 10 | things = doubleStuff(things) | | |  |  | | --- | --- | |  | Objects | |
|  | |

This pattern of computation is so common that python offers a more general way to do mappings, the map function, that makes it more clear what the overall structure of the computation is. map takes two arguments, a function and a sequence. The function is the mapper that transforms items. It is automatically applied to each item in the sequence. You don’t have to initialize an accumulator or iterate with a for loop at all.

**Note**

Technically, in a proper Python 3 interpreter, the map function produces an “iterator”, which is like a list but produces the items as they are needed. Most places in Python where you can use a list (e.g., in a for loop) you can use an “iterator” as if it was actually a list. So you probably won’t ever notice the difference. If you ever really need a list, you can explicitly turn the output of map into a list: list(map(...)). In the runestone environment, map actually returns a real list, but to make this code compatible with a full python environment, we always convert it to a list.

As we did when passing a function as a parameter to the sorted function, we can specify a function to pass to map either by referring to a function by name, or by providing a lambda expression.

def triple(value):

return 3\*value

​

def tripleStuff(a\_list):

new\_seq = map(triple, a\_list)

return list(new\_seq)

​

def quadrupleStuff(a\_list):

new\_seq = map(lambda value: 4\*value, a\_list)

return list(new\_seq)

​

things = [2, 5, 9]

things3 = tripleStuff(things)

print(things3)

things4 = quadrupleStuff(things)

print(things4)

​

Of course, once we get used to using the map function, it’s no longer necessary to define functions like tripleStuff and quadrupleStuff.

things = [2, 5, 9]

​

things4 = map((lambda value: 4\*value), things)

print(list(things4))

​

# or all on one line

print(list(map((lambda value: 5\*value), [1, 2, 3])))

23.3. Filter

Now consider another common pattern: going through a list and keeping only those items that meet certain criteria. This is called a filter.

def keep\_evens(nums):

new\_list = []

if num % 2 == 0:

new\_list.append(num)

return new\_list

​

print(keep\_evens([3, 4, 6, 7, 0, 1]))

​

Again, this pattern of computation is so common that Python offers a more compact and general way to do it, the filter function. filter takes two arguments, a function and a sequence. The function takes one item and return True if the item should. It is automatically called for each item in the sequence. You don’t have to initialize an accumulator or iterate with a for loop.

def keep\_evens(nums):

new\_seq = filter(lambda num: num % 2 == 0, nums)

return list(new\_seq)

​

print(keep\_evens([3, 4, 6, 7, 0, 1]))

**Check Your Understanding**

**1.** Write code to assign to the variable filter\_testing all the elements in lst\_check that have a w in them using filter.

​

lst\_check = ['plums', 'watermelon', 'kiwi', 'strawberries', 'blueberries', 'peaches', 'apples', 'mangos', 'papaya']

​

​

**2.** Using filter, filter lst so that it only contains words containing the letter “o”. Assign to variable lst2. Do not hardcode this.

​

lst = ["witch", "halloween", "pumpkin", "cat", "candy", "wagon", "moon"]

​

23.4. List Comprehensions

Python provides an alternative way to do map and filter operations, called a **list comprehension**. Many programmers find them easier to understand and write. List comprehensions are concise ways to create lists from other lists. The general syntax is:

[<transformer\_expression> **for** <loop\_var> **in** <sequence> **if** <filtration\_expression>]

where the if clause is optional. For example,

things = [2, 5, 9]

​

yourlist = [value \* 2 for value in things]

​

print(yourlist)

​

The transformer expression is value \* 2. The item variable is value and the sequence is things. This is an alternative way to perform a mapping operation. As with map, each item in the sequence is transformed into an item in the new list. Instead of the iteration happening automatically, however, we have adopted the syntax of the for loop which may make it easier to understand.

Just as in a regular for loop, the part of the statement for value in things says to execute some code once for each item in things. Each time that code is executed, value is bound to one item from things. The code that is executed each time is the transformer expression, value \* 2, rather than a block of code indented underneath the for statement. The other difference from a regular for loop is that each time the expression is evaluated, the resulting value is appended to a list. That happens automatically, without the programmer explicitly initializing an empty list or appending each item.

The if clause of a list comprehension can be used to do a filter operation. To perform a pure filter operation, the expression can be simply the variable that is bound to each item. For example, the following list comprehension will keep only the even numbers from the original list.

def keep\_evens(nums):

new\_list = [num for num in nums if num % 2 == 0]

return new\_list

​

print(keep\_evens([3, 4, 6, 7, 0, 1]))

​

You can also combine map and filter operations by chaining them together, or with a single list comprehension.

things = [3, 4, 6, 7, 0, 1]

#chaining together filter and map:

# first, filter to keep only the even numbers

# double each of them

print(map(lambda x: x\*2, filter(lambda y: y % 2 == 0, things)))

​

# equivalent version using list comprehension

print([x\*2 for x in things if x % 2 == 0])

​

**Check your understanding**

AdAccum-4-1: What is printed by the following statements?

alist = [4,2,8,6,5]

blist = [num\*2 **for** num **in** alist **if** num%2==1]

**print**(blist)

Top of Form

A. [4,2,8,6,5]  
B. [8,4,16,12,10]  
C. 10  
D. [10]

23.5. Zip

One more common pattern with lists, besides accumulation, is to step through a pair of lists (or several lists), doing something with all of the first items, then something with all of the second items, and so on. For example, given two lists of numbers, you might like to add them up pairwise, taking [3, 4, 5] and [1, 2, 3] to yield [4, 6, 8].

One way to do that with a for loop is to loop through the possible index values.

L1 = [3, 4, 5]

L2 = [1, 2, 3]

L3 = []

​

for i in range(len(L1)):

L3.append(L1[i] + L2[i])

​

print(L3)

​

You have seen this idea previously for iterating through the items in a single list. In many other programming languages that’s really the only way to iterate through the items in a list. In Python, however, we have gotten used to the for loop where the iteration variable is bound successively to each item in the list, rather than just to a number that’s used as a position or index into the list.

Can’t we do something similar with pairs of lists? It turns out we can.

The zip function takes multiple lists and turns them into a list of tuples (actually, an iterator, but they work like lists for most practical purposes), pairing up all the first items as one tuple, all the second items as a tuple, and so on. Then we can iterate through those tuples, and perform some operation on all the first items, all the second items, and so on.

L1 = [3, 4, 5]

L2 = [1, 2, 3]

L4 = list(zip(L1, L2))

print(L4)

​

Here’s what happens when you loop through the tuples.

L1 = [3, 4, 5]

L2 = [1, 2, 3]

L3 = []

L4 = list(zip(L1, L2))

​

for (x1, x2) in L4:

L3.append(x1+x2)

​

print(L3)

​

Or, simplifying and using a list comprehension:

L1 = [3, 4, 5]

L2 = [1, 2, 3]

L3 = [x1 + x2 for (x1, x2) in list(zip(L1, L2))]

print(L3)

Bottom of Form