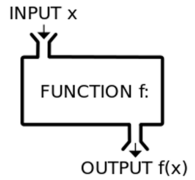
Higher-Order List Operations



CS111 Computer Programming

Department of Computer Science
Wellesley College

Overview

Today we will see how so-called higher-order list operations capture list-manipulation patterns.

Higher-order list operations may be much simpler than loops at performing many list manipulation tasks.

The map function captures the mapping pattern

Previously, we saw examples of the list mapping pattern, in which the output list is the same length as the input list, and each element of the output list is the result of applying some function to the corresponding element of the input list.

```
mapDouble( [8,  3,  6,  7,  2,  4] )
            |   |   |   |   |   |
           *2  *2  *2  *2  *2  *2
            |   |   |   |   |   |
          [16,  6, 13, 14,  4,  8]
```

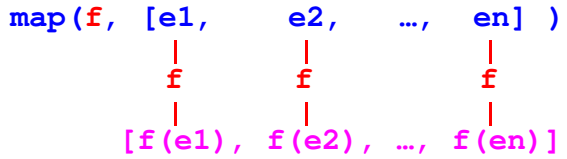

The map function captures the mapping pattern

Previously, we saw examples of the list mapping pattern, in which the output list is the same length as the input list, and each element of the output list is the result of applying some function to the corresponding element of the input list.

```
mapPluralize( ['donut', 'muffin', 'bagel'] )
                  |         |         |
                 +s        +s        +s
                  |         |         |
            ['donuts', 'muffins', 'bagels']
```

The map function captures the mapping pattern

Python provides a map function that captures this pattern. When invoked as map(*function*, *inlist*), it returns a new output list that's the same length as *inlist* in which every element is the result of applying *function* to the corresponding element of *inlist*.



map examples

Suppose we define the following functions:

```
def double(n):
  return 2*n

def pluralize(st):
  return st+'s'
```

Then we can use map to apply these functions to each element of a list:

```
In [1]: map(double, [8,3,6,7,2,4])
Out[1]: [16, 6, 12, 14, 4, 8]

In [2]: map(pluralize, ['donut','muffin','bagel'])
Out[2]: ['donuts', 'muffins', 'bagels']
```


map examples

```
In [1]: map(double, [8,3,6,7,2,4])
Out[1]: [16, 6, 12, 14, 4, 8]

In [2]: map(pluralize, ['donut','muffin','bagel'])
Out[2]: ['donuts', 'muffins', 'bagels']
```

Note that the first argument to map in these examples is a one-argument function. It is just the function itself and not the result of applying the function to anything. There's a big difference between these two situations:

```
In [3]: double  # just the function itself
Out[3]: <function __main__.double>

In [4]: double(8)  # result of applying function
Out[4]: 16
```


mapDouble and mapPluralize without loops

```
def mapDouble(nums):
  return map(double, nums)

def mapPluralize(strings):
  return map(pluralize, strings)


In [5]: mapDouble([8,3,6,7,2,4])
Out[5]: [16, 6, 12, 14, 4, 8]

In [6]: mapPpluralize(['donut','muffin','bagel'])
Out[6]: ['donuts', 'muffins', 'bagels']
```

Customized map function

How does map work? To illustrate, we can define our own version of Python's map function as follows:

```
def myMap(f, elts):
  result = []
  for e in elts:
    result.append(f(e))
  return result
```

```
In [7]: myMap(double, [8,3,6,7,2,4])
Out[7]: [16,6,12,14,4,8]
```


Customized map function

```
def myMap(f, elts):
  result = []
  for e in elts:
    result.append(f(e))
  return result
```

All myMap does is capture the mapping pattern we've seen before in one function so that we don't have to repeat it over and over again. We write a standard loop that accumulates a list result in myMap exactly once, and then we never need to write this loop again for the mapping pattern.

Customized map function

```
def myMap(f, elts):
  result = []
  for e in elts:
    result.append(f(e))
  return result
```

The definition of myMap depends critically on being able to pass a function as a parameter. Not all programming languages permit this, but Python does. Functions that take other functions as parameters or return them as results are called **higher-order functions**. Thus, map is an example of a **higher-order list function**, i.e., a higher order function that manipulates lists.

Exercise

Using the map function, define a mapSquare function that takes a list of numbers and returns a list of its squares:

```
In [8]: mapSquare([8,3,5])
Out[8]: [64,9,25]
```

Using the map function, define a mapUpper function that takes a list of strings and returns the result of uppercasing each string. Use the string .upper() method to uppercase strings.

```
In [9]: mapUpper(['ant','bat','cat'])
Out[9]: ['ANT','BAT','CAT']
```

Defining local functions to use with map

In the previous examples, we defined a global function **double** to use within the **mapDouble** function definition. It would be nicer to define everything within one function, and Python lets us do this by defining the **double** function inside the **mapDouble** function:

```
def mapDouble(nums):
  # Locally define double function within mapDouble.
  # Can only be used inside mapDouble and not outside
  def double(n):
    return 2*n
  return map(double, nums)
```


Defining local functions to use with map

Sometimes we ***must*** define the function used by **map** locally within another function because it needs to refer to a parameter or some other piece of information that is local to the enclosing function.

```
def mapScale(factor,nums):
  # Can't define this outside of mapScale, because
  # it needs to refer to parameter named "factor"
  def scale(n):
    return factor*n
  return map(scale, nums)
```

```
In [10]: mapScale(3, [8,3,6,7,2,4])
Out[10]: [24,9,18,21,6,12]

In [11]: mapScale(10, [8,3,6,7,2,4])
Out[11]: [80,30,60,70,20,40]
```


Defining local functions to use with map

Sometimes we ***must*** define the function used by **map** locally within another function because it needs to refer to a parameter or some other piece of information that is local to the enclosing function.

```
def mapPrefixes(string):
  # Can't define this outside of mapPrefixes, because
  # it needs to refer to parameter named "string"
  def prefix(i):
    return string[:i+1]
  return map(prefix, range(len(string)))
```

```
In [12]: mapPrefixes('program')
Out[12]: ['p','pr','pro','prog','progr','progra','program']
```


Exercise

Using the **map** function, define a **mapPreConcat** function that takes a string **pre** and a list of strings and returns the result of concatenating **pre** in front of every string in the list.

```
In [13]: mapPreConcat('com', ['puter','pile','mute'])
Out[13]: ['computer','compile','commute']
```