Anonymous Functions

CS111 Computer Programming

Department of Computer Science
Wellesley College

Higher Order Functions: Review

Functions that take other functions as parameters or return them as results are called **higher-order functions**.

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

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


lambda notation creates anonymous functions

Sometimes it is inconvenient to define a named function just in order to pass it as the functional argument to `map`. Python provides the alternative of using so-called **lambda notation** to create an **anonymous function**. For example, the notation

```
lambda n: n*2
```

creates an anonymous function that takes a single parameter named `n` and returns the value `n*2`. You can pronounce this as "I am a function that takes a parameter `n` and returns `n*2`."

lambda notation creates anonymous functions

```
lambda st: st+'s'
```

creates an anonymous function that takes a single parameter named `st` and returns the value `st+'s'`.

Note that you do not include an explicit `return` statement in the body of a `lambda` function; `return` is implicit in `lambda`.

Example mapping functions using **lambda**

```
In [2]: map(lambda n:n*2, [8,3,6,7,2,4])
Out[2]: [16, 6, 12, 14, 4, 8]

In [3]: map(lambda n:n*n, [8,3,6,7,2,4])
Out[3]: [64, 9, 36, 49, 4, 16]

In [4]: map(lambda st:st+'s',['donut','muffin','bagel'])
Out[4]: ['donuts','muffins','bagels']
```


Example mapping functions using **lambda**

```
def mapScale(factor, nums):
  return map(lambda n: factor*n, nums)

In [5]: mapScale(3, [8,3,6,7,2,4])
Out[5]: [24,9,18,21,6,12]

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


Example mapping functions using **lambda**

```
def mapPrefixes(string):
  return map(lambda i: string[:i+1], range(len(string)))

In [7]: mapPrefixes('cat')
Out[7]: ['c','ca','cat']

In [8]: mapPrefixes('program')
Out[8]: ['p','pr','pro','prog','progr','progra','program']
```


Example mapping functions using **lambda**

words.txt

```
ant
bat
cat
```

```
In [9]: open('words.txt').readlines()
Out[9]: ['ant\n','bat\n','cat\n']
```

```
def linesFromFile(filename):
  f = open(filename, 'r')
  lines = map(lambda line:line.strip(), f.readlines())
  f.close()
  return lines

In [10]: linesFromFile('words.txt')
Out[10]: ['ant','bat','cat']
```

mapPreConcat

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

```
def mapPreConcat(pre,strings):
  def concat(i):
    return pre+strings[i]
  return map(concat,range(len(strings)))
```

```
In [11]: mapPreConcat('com', ['puter','pile','mute'])
Out[11]: ['computer','compile','commute']
```

Redefine the **mapPreConcat** function from above so that it uses **lambda** notation rather than a locally defined function.

Mapping Over Multiple Lists

The **map** function allows any number of list arguments as long as the supplied function takes a number of arguments that's the same as the number of lists.

```
def add2(a,b):
  return a+b
```

```
In [12]: map(add2, [8,3,5], [10,20,30])
Out[12]: [18, 23, 35]

In [13]: map(lambda a,b: a+b, [8,3,5], [10,20,30])
Out[13]: [18, 23, 35]

In [14]: map(lambda a,b: a*b, [8,3,5], [10,20,30])
Out[14]: [80, 60, 150]

In [15]: map(lambda a,b: (a,b), [8,3,5], [10,20,30])
Out[15]: [(8,10), (3,20), (5,30)]
```


Mapping Over Multiple Lists

When mapping over multiple lists, all the lists must have the same length; if not, an exception will be raised.

```
def add2(a,b):
  return a+b
```

```
In [16]: map(add2, [8,3,5,6], [10,20,30])
```

```
---------------------------------------------------------------------
TypeError                           Traceback (most recent call last)
<ipython-input-93-e73284726> in <module>()
----> 1 map(add2, [8,3,5,6], [10,20,30])

TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```


Using map on Other (non-list) sequences

The **map** function can be used on any sequence, but always returns a list.

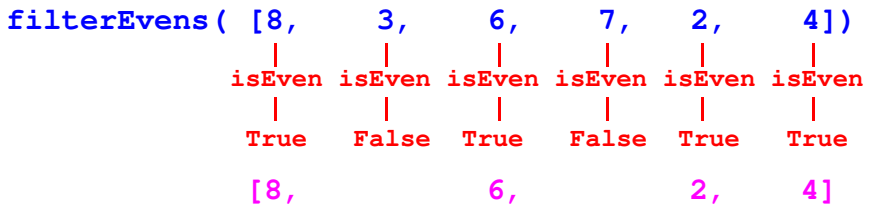
```
In [17]: map(lambda s: s.upper(), 'foo')
Out[17]: ['F', 'O', 'O']

In [18]: map(lambda s: s.upper(), ('ant','bat','cat')
Out[18]: ['ANT', 'BAT', 'CAT']
```

For mapping over each letter of a string, we can get a string from the resulting list of strings by using the **join** method.

```
In [19]: ''.join(map(lambda s: s.upper(), 'foo'))
Out[19]: 'FOO'
```

The Filtering Pattern

We've seen examples of the **filtering pattern**, in which an output list includes only those input elements for which a certain predicate is `True`.



The `filter` funtion captures the filtering pattern

The Python `filter` function captures this pattern. It takes a predicate function and a list and returns a list of all items for which the predicate function returns `True`.

```
def isEven(n):
  return n%2==0

In [20]: filter(isEven, [8,3,6,7,2,4])
Out[20]: [8, 6, 2, 4]


def filterEven(nums):
  return filter(isEven, nums)

In [21]: filterEven([8,3,6,7,2,4])
Out[21]: [8, 6, 2, 4]
```


The `filter` funtion captures the filtering pattern

The Python `filter` function captures this pattern. It takes a predicate function and a list and returns a list of all items for which the predicate function returns `True`.

```
In [22]: filter(lambda n:n%2==0, [8,3,6,7,2,4])
Out[22]: [8, 6, 2, 4]


def filterEven(nums):
  return filter(lambda n:n%2==0, nums)

In [23]: filterEven([8,3,6,7,2,4])
Out[23]: [8, 6, 2, 4]
```


Exercise

Using the `filter` function, define a `filterPositive` function that takes a list of numbers and returns a list of its positive elements.

```
In [24]: filterPositives([8,-3,6,7,-2,-4,5])
Out[24]: [8, 6, 7, 5]
```

Using the `filter` function, define a `filterSameLength` function that takes a string `st` and a list of strings and returns all the strings in the list that have the same length as `st`.

```
In [25]: filterSameLength('ant',['the','gray','cat','is','big'])
Out[25]: ['the','cat','big']

In [26]: filterSameLength('purr',['the','gray','cat','is','big'])
Out[26]: ['gray']

In [27]: filterSameLength('q',['the','gray','cat','is','big'])
Out[27]: []
```

The `reduce` funtion captures the accumulation pattern

Python supplies a **`reduce`** function that can be used to accumulate the elements of a list into a result that may not be a list.

```
In [28]: reduce(lambda a,b:a+b, [8,12,5,7,6,4], 0)
Out[28]: 42

In [29]: reduce(lambda a,b:a+b, ['I','have','a','dream'], '')
Out[29]: 'Ihaveadream'

In [30]: reduce(lambda a,b:a+b, [[8,2,5],[17],[],[6,3]], [])
Out[30]: [8,2,5,17,6,3]
```


Exercise

Using the **`reduce`** function, define a **`productList`** function that takes a list of numbers and returns the product of all the elements in the list.

```
In [31]: productList([4,5,2,3])
Out[31]: 120
```

Using the **`reduce`** function, define an **`areAllPositive`** function that takes a list of numbers and returns **`True`** if all the elements are positive. Otherwise the function returns **`False`**.

```
In [32]: areAllPositive([4,5,2,3])
Out[32]: True

In [33]: areAllPositive([4,5,-2,3])
Out[33]: False

In [34]: areAllPositive([-4,5,2,-3])
Out[34]: False
```