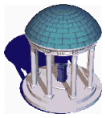


The University of North Carolina at Chapel Hill

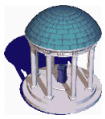
COMP 144 Programming Language Concepts

Spring 2002

Lecture 21: Functional Programming in Python

Felix Hernandez-Campos

March 1

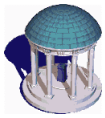


List Comprehensions
Haskell

- Lists can be defined by enumeration using *list comprehensions*
  - Syntax:

Generator

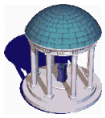
```
[ f x | x <- xs ]
[ (x,y) | x <- xs, y <- ys ]
```



List Comprehensions
Python

```
>>> freshfruit = ['  banana', '  loganberry
  ', 'passion fruit  ']
>>> [weapon.strip() for weapon in
  freshfruit]
['banana', 'loganberry', 'passion fruit']
```

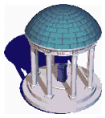
COMP 144 Programming Language Concepts
Felix Hernandez-Campos
3



List Comprehensions
Python

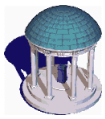
```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

COMP 144 Programming Language Concepts
Felix Hernandez-Campos
4



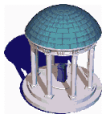
List Comprehensions
Python

```
>>> [{x: x**2} for x in vec]
[{2: 4}, {4: 16}, {6: 36}]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec]       # error -
  parens required for tuples
File "<stdin>", line 1, in ?
  [x, x**2 for x in vec]
           ^
SyntaxError: invalid syntax
```

List Comprehensions
Python

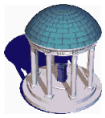
```
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```



List Comprehension
Python

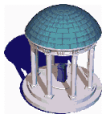
- Quicksort example

```
quicksort [] = []
quicksort (x:xs) = quicksort [y | y <- xs, y<x ]
                   ++ [x]
                   ++ quicksort [y | y <- xs, y>=x]
```

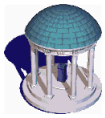
List Comprehensions
Python

```
def quicksort(list):

    if (len(list) == 0):
        return []

    else:
        pivot = list[0]
        l = []
        l = l + quicksort([x for x in list[1:] if x < pivot])
        l.append(pivot)
        l = l + quicksort([x for x in list[1:] if x >= pivot])
        return l
```



Higher-Order Functions

- Higher-order functions are functions that take other functions as arguments
- They can be use to implement algorithmic *skeletons*
  - Generic algorithmic techniques
- Three predefined higher-order functions are specially useful for working with list
  - `map`
  - `fold`
  - `filter`



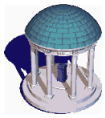
Map
Haskell

- Applies a function to all the elements of a list

```
map           :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x : xs) = f x : map f xs
```

  - Examples

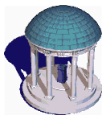
```
map square [9, 3]     ⇒     [81, 9]

map (<3) [1, 5]       ⇒     [True, False]
```



Map
Python

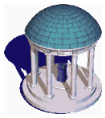
- "map(function, sequence)" calls function(item) for each of the sequence's items and returns a list of the return values.
- For example, to compute some cubes:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729,
 1000]
```

Map
Python

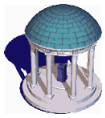
- More than one sequence may be passed
- the function must then have as many arguments as there are sequences
- It is called with the corresponding item from each sequence (or None if some sequence is shorter than another). If None is passed for the function, a function returning its argument(s) is substituted.



Map
Python

- Combining these two special cases, we see that "map(None, list1, list2)" is a convenient way of turning a pair of lists into a list of pairs.
- For example

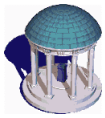
```
>>> seq = range(8)
>>> def square(x): return x*x
...
>>> map(None, seq, map(square, seq))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16),
  (5, 25), (6, 36), (7, 49)]
```

Zip

- Zip combines two lists into a list of pairs

```
zip                 :: [a] -> [b] -> [(a,b)]

zip [] ys           = []
zip (x:xs) []       = []
zip (x:xs) (y:ys) = (x,y):zip(xs,ys)
```



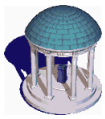
Filter
Haskell

- Extracts the elements of a list that satisfy a boolean function

```
filter            :: (a -> Bool) -> [a] -> [a]
filter p []        = []
filter p (x : xs) = if p x then x : filter p xs
                           else filter p xs
```

– Example

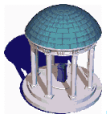
```
filter (>3) [1, 5, -5, 10, -10]   ⇒     [5, 10]
```

Filter
Python

- filter(function, sequence)" returns a sequence (of the same type, if possible) consisting of those items from the sequence for which function(item) is true.
- For example, to compute some primes:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```



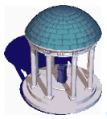
Fold

- Takes in a function and *folds* it in between the elements of a list
- Two flavors:
  - *Right-wise* fold: $[x_1, x_2, x_3] \Rightarrow x_1 \oplus (x_2 \oplus (x_3 \oplus e))$

Fold Operator

Base Element

```
foldr           :: (a -> b -> b) -> b -> [a] -> [a]
foldr f e []     = []
foldr f e (x:xs) = f x (foldr f e xs)
```

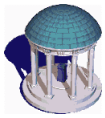
Foldl

- *Left-wise* fold: $[x_1, x_2, x_3] \Rightarrow ((e \oplus x_1) \oplus x_2) \oplus x_3$

```
foldl           :: (a -> b -> b) -> b -> [a] -> [a]
foldl f e []     = []
foldl f e (x:xs) = foldl f (f e x) xs
```

- Example

```
max a b = if a > b then a else b
foldl max 0 [1,2,3]   ⇒     3
```

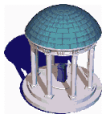


Folding in Python: Reduce

- "reduce(*func*, sequence)" returns a single value constructed by calling the binary function *func* on the first two items of the sequence, then on the result and the next item, and so on.
- For example, to compute the sum of the numbers 1 through 10:

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

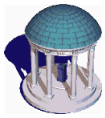
- If there's only one item in the sequence, its value is returned; if the sequence is empty, an exception is raised.



Reduce

- A third argument can be passed to indicate the starting value. In this case the starting value is returned for an empty sequence, and the function is first applied to the starting value and the first sequence item, then to the result and the next item, and so on.
- For example,

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```



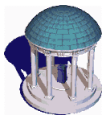
Lambda Abstractions

- Anonymous functions are also useful
  - They are known as lambda abstractions
- Haskell

```
map (\x->3*x) [1,2,3]
```

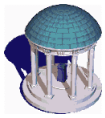
- Python

```
>>> car = lambda lst: lst[0]
>>> cdr = lambda lst: lst[1:]
>>> sum2 = lambda lst: car(lst)+car(cdr(lst))
>>> sum2(range(10))
1
```

More on Python Functional Programming

- Articles by David Mertz
- http://www-106.ibm.com/developerworks/linux/library/l-prog.html
- http://www-106.ibm.com/developerworks/library/l-prog2.html



Reading Assignment

- Python tutorial
  - List comprehensions
    - » http://www.python.org/doc/current/tut/node7.html#SECTION007140000000000000000
  - List displays
    - » http://www.python.org/doc/current/ref/lists.html#l2h-238
  - Higher-order programming with list
    - » http://www.python.org/doc/current/tut/node7.html#SECTION007130000000000000000