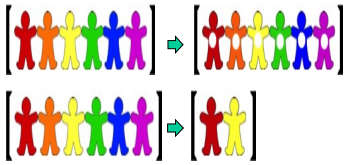
List Processing Patterns and List Comprehension



CS111 Computer Programming

Department of Computer Science
Wellesley College

Review: Lists

Concepts in this slide: Summary of what we know about lists.

A list is a sequence type (like strings and tuples), but that differently from them is mutable (it can change). Lists can store elements of different types (e.g., numbers, booleans, strings). Lists can be nested to create a list of lists. They are usually homogeneous (all elements of the same type), but Python allows heterogeneous lists too. A list with no elements is called an empty list.

```
primes = [2,3,5,7,11,13,17,19] # List of primes less than 20
bools   = [1<2, 1==2, 1>2]
houses  = ['Gryffindor', 'Hufflepuff', 'Ravenclaw', 'Slytherin']
strings = ['ab' + 'cd', 'ma'*4]
people = ['Hermione Granger', 'Harry Potter',
          'Ron Weasley', 'Luna Lovegood']

# A list of string lists
animalLists = [['duck', 'raccoon'],
               ['fox', 'raven', 'gosling'], [], ['turkey']]

# A heterogeneous list
stuff = [17, True, 'foo', None, [42, False, 'bar']]

empty = [] # An empty list
```

New example

Review: membership operations in sequences

```
people = ['Hermione Granger', 'Harry Potter',
          'Ron Weasley', 'Luna Lovegood']
```

```
In []: 'Hermione Granger' in people
Out[]: True
In []: 'Hagrid' in people
Out[]: False
In []: 'Luna' in people
Out[]: False
```

in simplifies `isVowel` and `isValidGesture`:

```
def isVowel(char):
    return char.lower() in 'aeiou'
```

```
def isValidGesture(gesture):
    return gesture in
      ['rock', 'paper', 'scissors']
```

in on strings

x in s

determines if **x** is a **substring in s**, not just if **x** is a **character in s**.

```
I[]:'e' in 'Hermione Granger'
O[]: True

I[]:'x' in 'Hermione Granger'
O[]: False

I[]:'Hermione' in \
    'Hermione Granger'
O[]: True

I[]:'oneG' in 'Hermione Granger'
O[]: False

I[]:'one G' in  \
    'Hermione Granger'
O[]: True
```


Review: accumulation of values

Concepts in this slide: The steps of the accumulation pattern.

```
In []: sumList([8,3,10,4,5])
Out[]: 30
```

Iteration table

step	n	sumSoFar
0		0
1	8	8
2	3	11
3	10	21
4	4	25
5	5	30

```
def sumList(nums):
    sumSoFar = 0
    for n in nums:
        sumSoFar += n
    return sumSoFar
```

initialize accumulator

update accumulator

return accumulator

Concepts in this slide:
How to add to a list.

Accumulation with a list

Lists can be accumulated using the method `.append` which adds a new element to the end of the list. The method `.append` **mutates** the original list by changing its content. We will learn much more in the next lecture about **mutation** and other list methods.

```
In []: a = [1, 2, 3]
In []: a.append(4)# mutate the list assigned to a by appending 4
In []: a
Out[]: [1, 2, 3, 4]
```

We can also accumulate lists using concatenation. Note that concatenation returns a new list instead of mutating the original list.

```
In []: a = [1, 2, 3]
In []: a = a + [4] # create a new list through concatenation and reassign
a to the new list
In []: a
Out[]: [1, 2, 3, 4]
```


Accumulation with a list

Concepts in this slide:
Modify accumulation pattern to work with lists.

Recall printHalves from Iteration I:

```
def printHalves(n):
    '''Prints positive successive
       halves of n'''
    while (n > 0):
        print(n)
        n = n//2
```

Iteration table

step	n
0	22
1	11
2	5
3	2
4	1
5	0

```
In []: printHalves(22)
Out[]:
22
11
5
2
1
```

append plays a key role:

```
def halves(n):
    result = []
    while (n > 0):
        result.append(n)
        n = n//2
    return result
```

values of n are collected into result

```
In []: halves(22)
Out[]: [22, 11, 5, 2, 1]
```


Double accumulation: `partialSums`

Use loops to build the list:

1. Start with an empty list [ ]
2. Use a loop to `append` elements to this list one at a time

➔ modify the **`sumList`** function to return a list of the partial sums calculated along the way:

```
def partialSums(nums):
    sumSoFar = 0
    partials = []
    for n in nums:
        sumSoFar += n
        partials.append(sumSoFar)
    return partials
```

initialize accumulators

update accumulators

return accumulator

step	n	sumSo Far	partials
0		0	[]
1	8	8	[8]
2	3	11	[8,11]
3	10	21	[8,11,21]
4	4	25	[8,11,21,25]
5	5	30	[8,11,21,25, 30]

```
In []: partialSums([8,3,10,4,5])
Out[]: [8,11,21,25,30]
```


Exercise 1: `prefixes`


```
In [ ]: prefixes('Paula')
Out[ ]:['P','Pa','Pau','Paul','Paula']
```

step	char	prefixSoFar	prefix
0		''	[]
1	'P'	'P'	['P']
2	'a'	'Pa'	['P','Pa']
3	'u'	'Pau'	['P','Pa','Pau']
4	'l'	'Paul'	['P','Pa','Pau','Paul']
5	'a'	'Paula'	['P','Pa','Pau','Paul','Paula']

```
def prefixes(phrase):
    '''Given a string, returns a list of nonempty prefixes of
    the string, ordered from shortest to longest
    '''
```

Will do this in the notebook in class.

List patterns: map & filter

Concepts in this slide: Definitions for mapping and filtering patterns.

```
people = ['Hermione Granger', 'Harry Potter',
          'Ron Weasley', 'Luna Lovegood']
```

1. MAPPING: return a new list that results from performing an operation on each element of a given list.

E.g. Return a list of the first names in **people**

```
['Hermione', 'Harry', 'Ron', 'Luna']
```

2. FILTERING: return a new list that results from keeping those elements of a given list that satisfy some condition

E.g. Return a list of names with last names ending in 'er' in **people** `['Granger', 'Potter']`

Mapping pattern: an example

Concepts in this slide: Mapping has the same steps as accumulation.

We can produce a new list simply by performing an operation on every element in a given list. This is called the **mapping pattern**.

```
def mapDouble(nums):
    '''Takes a list of numbers and returns a new list in
    which each element is twice the corresponding
    element in the input list
    '''
    result = []
    for n in nums:
        result.append(2*n)
    return result
```

```
mapDouble([8,3,10,5,4]) returns [16,6,20,10,8]
mapDouble([17,42,6]) returns [34,84,12]
mapDouble([]) returns []
```


Exercise 2: mapLumos


```
def mapLumos(theList):
    '''Given a list of strings, returns a new list in which
    'Lumos' is added to the end of each string
    '''
```

Will do this in the notebook in class.

```
In [ ]: mapLumos (people)
Out[ ]: ['Hermione GrangerLumos', 'Harry PotterLumos',
         'Ron WeasleyLumos', 'Luna LovegoodLumos']

In [ ]: mapLumos (['Eni', 'Sohie', 'Lyn'])
Out[ ]: ['EniLumos', 'SohieLumos', 'LynLumos',]

In [ ]: mapLumos ([])
Out[ ]: []
```


Exercise 3: mapFirstWord


```
def mapFirstWord(strings):
    '''Given a list of (possibly multiword) strings,
    returns a new list in which each element is the first
    word
    '''
```

Will do this in the notebook in class.

```
In [ ]: mapFirstWord(people)
Out[ ]: ['Hermione', 'Harry', 'Ron', 'Luna']

In [ ]: mapFirstWord (['feisty smelly dog', 'furry white bunny',
                       'orange clown fish')]
Out[ ]: ['feisty', 'furry', 'orange']

In [ ]: mapFirstWord (['Eni', 'Sohie', 'Lyn'])
Out[ ]: ['Eni', 'Sohie', 'Lyn']
```

Filtering Pattern: an example

Concepts in this slide: Filtering has also the same steps as accumulation.

Another common way to produce a new list is to filter an existing list, keeping only those elements that satisfy a certain predicate. This is called the **filtering pattern**.

```
def filterEvens(nums):
    '''Takes a list of numbers and returns a new list
    of all numbers in the input list that are
    divisible by 2
    '''
    result = []
    for n in nums:
        if n%2 == 0:
            result.append(n)
    return result
```

`filterEvens([8,3,10,4,5])` returns `[8,10,4]`
`filterEvens([8,2,10,4,6])` returns `[8,2,10,4,6]`
`filterEvens([7,3,11,3,5])` returns `[]`

Exercise 4: Filtering strings by containment


```
def filterElementsContaining(val, aList):
    '''Return a new list whose elements are all the
     elements of aList that contain val
    '''
```

Will do this in the notebook in class.

```
people = ['Hermione Granger', 'Harry Potter',
          'Ron Weasley', 'Luna Lovegood']
```

```
In [ ]: filterElementsContaining('Harry', people)
Out[ ]:  ['Harry Potter']

In [ ]: filterElementsContaining('er', people)
Out[ ]: ['Hermione Granger', 'Harry Potter']

In [ ]: filterElementsContaining('Voldemort', people)
Out[ ]: [ ]

In [ ]: filterElementsContaining('smelly',['feisty smelly dog',
                                           'furry white bunny', 'orange clown fish'])
Out[ ]: ['feisty smelly dog']
```


Simplifying mapping & filtering with list comprehension

```
nums = [17,42,6]
result = []
for x in nums:
    result.append(x*2)
```

List Comprehension for mapping

```
result = [x*2 for x in nums]
```

```
result = []
for n in nums:
    if n%2 == 0:
        result.append(n)
```

List Comprehension for filtering

```
result = [n for n in nums if n%2 == 0]
```


List comprehension syntax

Concepts in this slide: List comprehension creates a new list in a single statement.

List Comprehension for mapping

```
newSequence =[expression for item in sequence]
```

List Comprehension for filtering

```
newSequence = \
[item for item in sequence if conditional]
```

To notice:

- List comprehension starts with an expression (note that a variable like item is an expression), for example, **x*2** or **n**.
- Never use **append** in this position. We are using list comprehension to avoid creating a list with **append**.

List comprehensions with Mapping and Filtering

```
newSequence = \
[expression for item in sequence if conditional]
```

The example below shows a list comprehension that extracts the even numbers from a range object and creates a list of their squares. The code to the right is analogous and shows the same process with iteration.

```
result = []
for n in range(10):
    if n%2 == 0:
        result.append(n**2)
result
```

List Comprehension for filtering and mapping

```
result = [n**2 for n in range(10) if n%2 == 0]
```


Summary

1. Lists are mutable data types that can change through assignment or through methods such **append**, **pop**, and **insert**.
2. The most used list method is **append**, because it is used to create new lists in different patterns: accumulation, mapping, and filtering.
3. In a function that implements accumulation we have three steps: 1) initialize accumulator (e. g., an empty list); 2) update of the accumulator (e.g., through **append**); 3) return the created accumulator.
4. Mapping and filtering are special cases of accumulation. They always need a sequence as a starting point (there is no such requirement for accumulation).
5. In mapping, the initial sequence and the mapped sequence will always have the same length, since the purpose of mapping is to apply an operation to all elements of the initial sequence.
6. In filtering, the initial sequence and the mapped sequence will have varying lengths, since the purpose of filtering is to keep only the elements that fulfill some criteria.
7. **List comprehension** is a Python syntactic idiom that simplifies the implementation of mapping and filtering patterns.

Test your knowledge

1. Suppose we have `lst = [1]` and perform `lst = lst.append(2)`. Try to guess the outcome and then print it in the console. Was it what you expected? How can you explain it?
2. We can add two lists, for example: `lst = [1]; lst + [2]`. How does this operation differ from the `lst.append(2)` above, since they both result in the list `[1, 2]`.
3. Review the method **insert** from the previous lecture on lists and memory diagrams. What are its similarities and differences with **append**?
4. Write a function that given a single integer number return a lists of tuples like below: `makeSquarePairs(5)` returns `[(1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]`. Try to do it in two ways: using **append** and then using list comprehension. Remember, you shouldn't use append in the list comprehension idiom.