List Processing Patterns and List Comprehension

Review: 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

Review: List Operations [1]

```
In [1]: listA = [1, 2, 3]
In [2]: listB = [4, 5, 6]
In [3]: listA + listB  # list concatenation
Out[3]: [1, 2, 3, 4, 5, 6]
In [4]: listA * 3  # list concatenation
Out[4]: [1, 2, 3, 1, 2, 3]
In [5]: listA[0] = 10  # assigning new value

In [6]: listA.append(100)  # this mutates list, returns None
In [7]: listA
Out[7]: [10, 2, 3, 100]

In [8]: listA.pop()  # this mutates list AND returns a value
Out[8]: 10  # .pop() removes the last element
In [9]: listA
Out[9]: [10, 2, 3]
In [10]: listA.pop(1)  # remove element at index 1
Out[10]: 2
In [11]: listA
Out[11]: [10, 3]
```


```
In [12]: listB.append(listA.pop())  # nested expressions
In [13]: listB
Out[13]: [4, 5, 6, 3]
In [14]: listA
Out[14]: [10]
In [15]: listB.append(listA)
Out[15]: [4, 5, 6, 3, [10]]
In [16]: listA[0] = 200  # mutating listA
In [17]: listB
Out[17]: [4, 5, 6, 3, [200]]  # HOW DID THIS HAPPEN?

State after [12]
```

```
State after [16]
```

```
In [18]: listB
Out[18]: [4, 5, 6, 3, [200]]
In [19]: listA
Out[19]: [10]
```

```
In [1]: listA = [1, 2, 3]
In [2]: listB = [4, 5, 6]
In [3]: listA + listB  # list concatenation
Out[3]: [1, 2, 3, 4, 5, 6]
In [4]: listA * 3  # list concatenation
Out[4]: [1, 2, 3, 1, 2, 3]
In [5]: listA[0] = 10  # assigning new value

In [6]: listA.append(100)  # this mutates list, returns None
In [7]: listA
Out[7]: [10, 2, 3, 100]

In [8]: listA.pop()  # this mutates list AND returns a value
Out[8]: 10  # .pop() removes the last element
In [9]: listA
Out[9]: [10, 2, 3]
In [10]: listA.pop(1)  # remove element at index 1
Out[10]: 2
In [11]: listA
Out[11]: [10, 3]
```

```
State after [12]
```

```
State after [16]
```

```
In [18]: listB
Out[18]: [4, 5, 6, 3, [200]]
In [19]: listA
Out[19]: [10]
```

```
In [1]: listA = [1, 2, 3]
In [2]: listB = [4, 5, 6]
In [3]: listA + listB  # list concatenation
Out[3]: [1, 2, 3, 4, 5, 6]
In [4]: listA * 3  # list concatenation
Out[4]: [1, 2, 3, 1, 2, 3]
In [5]: listA[0] = 10  # assigning new value

In [6]: listA.append(100)  # this mutates list, returns None
In [7]: listA
Out[7]: [10, 2, 3, 100]

In [8]: listA.pop()  # this mutates list AND returns a value
Out[8]: 10  # .pop() removes the last element
In [9]: listA
Out[9]: [10, 2, 3]
In [10]: listA.pop(1)  # remove element at index 1
Out[10]: 2
In [11]: listA
Out[11]: [10, 3]
```

```
State after [12]
```

```
State after [16]
```
List membership & early return

Only one of the following correctly determines if \textit{val} is an element in the list \texttt{aList}. Which one and why?

\begin{Verbatim}
def isElementOf1(val, aList):
    for elt in aList:
        if val == elt:
            return True
    return False
\end{Verbatim}

\begin{Verbatim}
def isElementOf2(val, aList):
    for elt in aList:
        if val == elt:
            return True
    return False
\end{Verbatim}

\begin{Verbatim}
def isElementOf3(val, aList):
    for elt in aList:
        if val == elt:
            return True
    return False
\end{Verbatim}

\begin{Verbatim}
list membership & early return
\end{Verbatim}

\textbf{In}: \\
\texttt{["Hermione Granger", "Harry Potter", "Ron Weasley", "Luna Lovegood"]}
\textbf{Out}:
\begin{Verbatim}
In [0]: 'Hermione Granger' in people
Out[0]: True
In [1]: 'Hagrid' in people
Out[1]: False
In [2]: 'Luna' in people
Out[2]: False
\end{Verbatim}

\begin{Verbatim}
in simplifies isVowel and isValidGesture:
\end{Verbatim}

\begin{Verbatim}
def isVowel(char):
    return char.lower() in 'aeiou'
\end{Verbatim}

\begin{Verbatim}
def isValidGesture(gesture):
    return gesture in ['rock', 'paper', 'scissors']
\end{Verbatim}

\textbf{List Patterns} 12-5

Membership operations in sequences

\begin{Verbatim}
people = ['Hermione Granger', 'Harry Potter', 'Ron Weasley', 'Luna Lovegood']
\end{Verbatim}

\begin{Verbatim}
In [0]: 'e' in 'Hermione Granger'
Out[0]: True
In [1]: 'x' in 'Hermione Granger'
Out[1]: False
In [2]: 'Hermione' in \ 'Hermione Granger'
Out[2]: True
In [3]: 'one G' in \ 'Hermione Granger'
Out[3]: False
\end{Verbatim}

\begin{Verbatim}
in on strings \textbf{x} \textit{in} \texttt{s}
determines if \textit{x} is a substring in \texttt{s}, not just if \textit{x} is a character in \texttt{s}.
\end{Verbatim}

\begin{Verbatim}
Review: membership operations in sequences
\end{Verbatim}

\textbf{List Patterns} 12-6

Review: accumulation of values

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

\begin{Verbatim}
Iteration table
\end{Verbatim}

\begin{Verbatim}
In []: sumList([8,3,10,4,5])
Out[]: 30
\end{Verbatim}

\begin{Verbatim}
def sumList(nums):
    sumSoFar = 0
    for n in nums:
        sumSoFar += n
    return sumSoFar
\end{Verbatim}

\begin{Verbatim}
Accumulation with a list
Recall printHalves from Iteration I:

\begin{Verbatim}
def printHalves(n):
    '''Prints positive successive halves of \textit{n}'''
    while (n > 0):
        print(n)
        n = n/2
\end{Verbatim}

append plays a key role:

\begin{Verbatim}
def halves(n):
    result = []
    while (n > 0):
        result.append(n)
        n = n/2
    return result
\end{Verbatim}

\begin{Verbatim}
In []: printHalves(22)
Out[]: [22, 11, 5, 2, 1]
\end{Verbatim}

\begin{Verbatim}
Iteration table
\end{Verbatim}

\begin{Verbatim}
In []: sumList([8,3,10,4,5])
Out[]: 30
\end{Verbatim}

\begin{Verbatim}
List Patterns 12-7
\end{Verbatim}

\begin{Verbatim}
List Patterns 12-8
\end{Verbatim}

\begin{Verbatim}
List Patterns 12-5
\end{Verbatim}

\begin{Verbatim}
List Patterns 12-6
\end{Verbatim}

\begin{Verbatim}
List Patterns 12-7
\end{Verbatim}

\begin{Verbatim}
List Patterns 12-8
\end{Verbatim}
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

gather the sumList function to return a list of the partial sums calculated along the way:

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

Double accumulation:
partialSums

Use loops to build the list:

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

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']

List patterns: map & filter

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

<table>
<thead>
<tr>
<th>step</th>
<th>n</th>
<th>sumSoFar</th>
<th>partials</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>[]</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>8</td>
<td>[8]</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>11</td>
<td>[8,11]</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>21</td>
<td>[8,11,21]</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>25</td>
<td>[8,11,21,25]</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>30</td>
<td>[8,11,21,25,30]</td>
</tr>
</tbody>
</table>

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
   ```python
   ['Granger', 'Potter']
   ```

Mapping pattern: an example

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

```python
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

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

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

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

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

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.

```python
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 []
```
Simplifying mapping & filtering with list comprehension

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

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

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

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

List comprehension syntax

**List Comprehension for mapping**

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

**List Comprehension for filtering**

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

To notice:
- List comprehension starts with an expression, for example, \(x \times 2\) or \(n\) (see slide 12-17).
- Never use `append` in this position. We are using list comprehension to avoid creating a list with `append`.

Review: Nested Loops with Lists

```python
pets = ['bunny', 'cat', 'dog']
parts = ['two eyes', 'four legs', 'fur']

for pet in pets:
    for part in parts:
        print 'A', pet, 'has', part
```

A bunny has two eyes
A bunny has four legs
A bunny has fur
A cat has two eyes
A cat has four legs
A cat has fur
A dog has two eyes
A dog has four legs
A dog has fur

Exercise 5: Nested Loops with Lists

```python
def printByCategory(categoryItemsPairs):
    '''Given a list of categories and a list of nested lists of items prints a category and all its corresponding items'''
    printByCategory(foodCategories)
```

```
foodCategories = [
    ('dairy', ['cheese', 'milk', 'yogurt']),
    ('fruits', ['apples', 'bananas', 'grapes', 'oranges']),
    ('veggies', ['cabbage', 'kale', 'lettuce'])
]
```

Will do this in the notebook in class.
Summary

1. Lists are mutable data types that can change through assignment or through methods such as `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 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` in Lecture 10 (slide 10-22, 23). What are its similarities and differences with `append`?
4. Mapping is equivalent to the concept of functions in math: think of square, cube, square root, logarithm, sin, cos, etc. How would you modify the `mapDouble` function in slide 12-12 to implement such mappings?
5. 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.