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.

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

animals = [['duck','raccoon'], ['fox','raven','gosling'], [], ['turkey']]

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

Review: List Operations

In [1]: listA = [1, 2, 3]
Out[1]: [1, 2, 3]

In [2]: listB = [4, 5, 6]
Out[2]: [4, 5, 6]

In [3]: listA + listB
Out[3]: [1, 2, 3, 4, 5, 6]  # list concatenation

In [4]: listA * 3
Out[4]: [1, 2, 3, 1, 2, 3]  # list concatenation

In [5]: listA[0] = 10
Out[5]: [10, 2, 3]

List Methods

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

In [7]: listA
Out[7]: [10, 2, 3, 100]

In [8]: listA.pop()  # this mutates list AND returns a value
Out[8]: 100

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
Out[12]: [4, 5, 6, 3]

In [13]: listA
Out[13]: [10, 3]

In [14]: listB
Out[14]: [4, 5, 6, 3, 10]

In [15]: listB.append(listA)
Out[15]: [4, 5, 6, 3, 10, 10]

In [16]: listA[0] = 200  # mutating listA
Out[16]: [200, 10, 3]

In [17]: listB
Out[17]: [4, 5, 6, 3, [200]]

In [18]: listA
Out[18]: [200, 10, 3]

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

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

# An empty list
Review: List membership & early return

Only one of the following correctly determines if \texttt{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
    else:
      return False

  def isElementOf2(val, aList):
    for elt in aList:
      if val == elt:
        return True
      return False

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

Review: membership operations in sequences

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

\begin{verbatim}
  In []: 'Hermione Granger' in people
  Out[]: True

  In []: 'Hagrid' in people
  Out[]: False

  In []: 'Luna' in people
  Out[]: False
\end{verbatim}

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

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

Review: accumulation of values

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

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

Accumulation with a list

Recall \texttt{printHalves} from Iteration I:

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

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

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

\begin{verbatim}
In []: halves(22)
Out[]: [22, 11, 5, 2, 1]
\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

Modify 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
```

<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></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>

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

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

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

```
[8,3,10,5,4] → [16,6,20,10,8]
```

```python
def mapDouble(nums):
    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.''
    result = []
    for n in theList:
        result.append(n + 'Lumos')
    return result

In [ ]: mapLumos (people)
Out[ ]: ['Hermione GrangerLumos', 'Harry PotterLumos', 'Ron WeasleyLumos', 'Luna LovegoodLumos']
In [ ]: mapLumos (['Eni', 'Sohie', 'Lyn'])
Out[ ]: ['EniLumos', 'SohieLumos', 'LynLumos']
In [ ]: mapLumos ([])
Out[ ]: []
Will do this in the notebook in class.
```

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.''
    result = []
    for n in strings:
        result.append(n.split(' ')[0])
    return result

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']
Will do this in the notebook in class.
```

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 []
```

Exercise 4: Filtering strings by containment

```python
def filterElementsContaining(val, aList):
    '''Return a new list whose elements are all the elements of aList that contain val.''
    result = []
    for n in aList:
        if val in n:
            result.append(n)
    return result

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']
```

Concepts in this slide:
Filtering has also the same steps as accumulation.
**Simplifying mapping & filtering with list comprehension**

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

**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*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`.

**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'''

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

printByCategory(foodCategories)
```

**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

**List patterns:**
- List comprehension creates a new list in a single statement.
- List comprehension starts with an expression, for example, `x*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`.

**Concepts in this slide:**
- List comprehension creates a new list in a single statement.
- List comprehension starts with an expression, for example, `x*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`.

**List Patterns 12-17**

**List Patterns 12-18**

**List Patterns 12-19**

**List Patterns 12-20**
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 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` 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.