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
click 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 membership & early return

Only one of the following correctly determines if `val` is an element in the list `aList`. Which one and why?

```python
def isElementOf1(val, aList):
    for elt in aList:
        if val == elt:
            return True
    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
```

Review: membership operations in sequences

```python
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:

```python
def isVowel(char):
    return char.lower() in 'aeiou'
def isValidGesture(gesture):
    return gesture in ['rock', 'paper', 'scissors']
```

in on strings

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

In []: 'e' in 'Hermione Granger'
Out[]: True
In []: 'x' in 'Hermione Granger'
Out[]: False
In []: 'oneG' in 'Hermione Granger'
Out[]: False
In []: 'one G' in 'Hermione Granger'
Out[]: True
```
**Review: accumulation of values**

The steps of the accumulation pattern.

<table>
<thead>
<tr>
<th>In []: sumList([8,3,10,4,5])</th>
<th>Out[]: 30</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>step</strong></td>
<td><strong>n</strong></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Concepts in this slide:**
- Initialize accumulator
- Update accumulator
- Return accumulator

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

**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>In []: partialSums([8,3,10,4,5])</th>
<th>Out[]: [8,11,21,25,30]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>step</strong></td>
<td><strong>n</strong></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Accumulation with a list**

Recall printHalves from Iteration I:

```python
def printHalves(n):
    '''Prints positive successive halves of n.'''
    result = []
    while (n > 0):
        result.append(n)
        n = n/2
    return result
```

<table>
<thead>
<tr>
<th>In []: printHalves(22)</th>
<th>Out[]: [22, 11, 5, 2, 1]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>step</strong></td>
<td><strong>n</strong></td>
</tr>
<tr>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

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

- `append` plays a key role:
- Values of `n` are collected into result

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

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

**Exercise 1: prefixes**

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

<table>
<thead>
<tr>
<th><strong>step</strong></th>
<th><strong>prefix</strong></th>
<th><strong>char</strong></th>
<th><strong>prefixSoFar</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>['P']</td>
<td>'P'</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>['P', 'Pa']</td>
<td>'Pa'</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>['P', 'Pa', 'Pau']</td>
<td>'Pau'</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>['P', 'Pa', 'Pau', 'Paul']</td>
<td>'Paul'</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>['P', 'Pa', 'Pau', 'Paul', 'Paula']</td>
<td>'Paula'</td>
<td></td>
</tr>
</tbody>
</table>

Will do this in the notebook in class.
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`
```
people = ['Hermione', 'Harry', 'Ron', 'Luna']
```

```
[ Hermione, Harry, Ron, Luna ]  ➔  [ H, H, R, L ]
```

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`
```
people = ['Granger', 'Potter']
```

```
[ Granger, Potter ]  ➔  [ G, P ]
```

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

```
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 []
```

List Patterns 12-9

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
    result = []
    for s in theList:
        result.append(s + 'Lumos')
    return result
```

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

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

Will do this in the notebook in class.

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
    result = []
    for s in strings:
        result.append(s.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.

\[ \text{def } \text{filterEvens}(\text{nums}) : \]

\[ \quad \text{\"\"\"Takes a list of numbers and returns a new list of all numbers in the input list that are divisible by 2 \"\"\"} \]

\[ \quad \text{result} = [\]

\[ \quad \text{for } n \text{ in } \text{nums} : \]

\[ \quad \quad \text{if } n\%2 == 0 : \]

\[ \quad \quad \text{result}.\text{append}(n) \]

\[ \quad \text{return } \text{result} \]

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

\[ \text{def } \text{filterElementsContaining}(\text{val}, \text{aList}) : \]

\[ \quad \text{\"\"\"Return a new list whose elements are all the elements of aList that contain val \"\"\"} \]

\[ \text{In [ ]: } \text{filterElementsContaining('Harry', people)} \]

\[ \text{Out[ ]: } ['\text{Harry Potter}'] \]

\[ \text{In [ ]: } \text{filterElementsContaining('er', people)} \]

\[ \text{Out[ ]: } ['\text{Hermione Granger}', '\text{Harry Potter}'] \]

\[ \text{In [ ]: } \text{filterElementsContaining('Voldemort', people)} \]

\[ \text{Out[ ]: } [] \]

\[ \text{In [ ]: } \text{filterElementsContaining('smelly', ['feisty smelly dog', 'furry white bunny', 'orange clown fish'])} \]

\[ \text{Out[ ]: } ['\text{feisty smelly dog}'] \]

List Patterns 12-13

Simplifying mapping & filtering with list comprehension

\[ \text{nums} = [17,42,6] \]

\[ \text{result} = [] \]

\[ \text{for } \text{x in } \text{nums} : \]

\[ \quad \text{result}.\text{append}(\text{x} * 2) \]

\[ \text{result} = [\text{x} * 2 \text{ for } \text{x in } \text{nums}] \]

\[ \text{result} = [] \]

\[ \text{for } \text{n in } \text{nums} : \]

\[ \quad \text{if } \text{n}\%2 == 0 : \]

\[ \quad \text{result}.\text{append}(\text{n}) \]

\[ \text{result} = [\text{n} \text{ for } \text{n in } \text{nums if } \text{n}\%2 == 0] \]

List Patterns 12-15

List comprehension syntax

List Comprehension for mapping

\[ \text{newSequence} = [\text{expression for } \text{item in } \text{sequence}] \]

List Comprehension for filtering

\[ \text{newSequence} = [\text{expression for } \text{item in } \text{sequence if } \text{conditional}] \]

To notice:

- List comprehension starts with an expression, for example, \[ \text{x} * 2 \text{ or } \text{n} \] (see slide 12-17).
- Never use \[ \text{append} \] in this position. We are using list comprehension to avoid creating a list with \[ \text{append} \].

List Patterns 12-16
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 `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.