CS2204

Program Design and Data Structures for Scientific Computing

Announcements

- Project #1 is due soon
  - We are creating a class that allows us to represent & manipulate DNA strands
  - We used a string as our underlying storage container

List Comprehensions

- A *list comprehension* is a programming language construct for creating a list based on existing lists
  - Haskell, Erlang, Scala and Python have them
- Why "comprehension"? The term is borrowed from math's *set comprehension* notation for defining sets in terms of other sets
- A powerful and popular feature in Python
  - Generate a new list by applying a function to every member of an original list
- Python's notation:

```
[ expression for name in list ]
```

List Comprehensions

- The syntax of a *list comprehension* is somewhat tricky

```
[x-10 for x in grades if x>0]
```

- Syntax suggests that of a *for*-loop, an *in* operation, or an *if* statement

- All three of these keywords (‘*for*’, ‘*in*’, and ‘*if*’) are also used in the syntax of forms of list comprehensions

List Comprehensions

```
>>> li = [3, 6, 2, 7]
>>> [elem*2 for elem in li]

[6, 12, 4, 14]
```

[expression for name in list] just like for-loop, but shorter

- Where expression is some calculation or operation acting upon the variable name.
- For each member of the list, the list comprehension
  1. sets name equal to that member,
  2. calculates a new value using expression,
- It then collects these new values into a list which is the return value of the list comprehension.

List Comprehensions

- If list contains elements of different types, then expression must operate correctly on the types of all of list members.

- If the elements of list are other containers, then the name can consist of a container of names that match the type and "shape" of the list members.

```
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [ n * 3 for (x, n) in li]
```

[3, 6, 21]

[(x, n * 3) for (x, n) in li]
[('a', 3), ('b', 6), ('c', 21)]

[n * 3 for (x, n) in li if x>5]
21

List Comprehensions

- If list contains elements of different types, then expression must operate correctly on the types of all of list members.

- If the elements of list are other containers, then the name can consist of a container of names that match the type and "shape" of the list members.

```
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [ n * 3 for (x, n) in li]

 [3, 6, 21]
```

List Comprehensions

- expression can also contain user-defined functions.

```
>>> def subtract(a, b):
      return a - b

>>> oplist = [(6, 3), (1, 7), (5, 5)]
>>> [subtract(y, x) for (x, y) in oplist]
```

List Comprehensions

- expression can also contain user-defined functions.

```
>>> def subtract(a, b):
      return a - b

>>> oplist = [(6, 3), (1, 7), (5, 5)]
>>> [subtract(y, x) for (x, y) in oplist]


[-3, 6, 0]
```

Filtered List Comprehension

```
[ expression for name in list if filter]
```

- Filter determines whether expression is performed on each member of the list.
- For each element of list, checks if it satisfies the filter condition.
- If the filter condition returns *False*, that element is omitted from the list before the list comprehension is evaluated.

Filtered List Comprehension

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem*2 for elem in li if elem > 4]
```

Filtered List Comprehension

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem*2 for elem in li if elem > 4]

[12, 14, 18]
```

Filtered List Comprehension

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem*2 for elem in li if elem > 4]

[12, 14, 18]
```

- Only 6, 7, and 9 satisfy the filter condition
- So, only 12, 14, and 18 are produced.

Nested List Comprehensions

Nested List Comprehensions

- Since list comprehensions take a list as input and produce a list as output, they are easily nested

```
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in
        [item+1 for item in li] ]
```

Nested List Comprehensions

- Since list comprehensions take a list as input and produce a list as output, they are easily nested

```
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in
        [item+1 for item in li] ]

[8, 6, 10, 4]
```

Nested List Comprehensions

- Since list comprehensions take a list as input and produce a list as output, they are easily nested

```
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in
        [item+1 for item in li] ]

[8, 6, 10, 4]
```

- The inner comprehension produces: [4, 3, 5, 2]
- So, the outer one produces: [8, 6, 10, 4]

For Loops & Lists

- REMINDER: A for-loop steps through, or iterate over, each of the items in a sequence (such as a list, tuple, or string), or any other type of object which is "*iterable*".

```
for item in collection:
    statements
```

- If `collection` is a list or a tuple, then the loop steps through each element of the sequence.

- If `collection` is a string, then the loop steps through each character of the string.

```
for someChar in "Hello World":
   print(someChar)
```

Analysis of Algorithms

- Next we are going to switch to lecture on the whiteboard and discuss:
  1. Sorting an array of data
  2. The Selection Sort algorithm
  3. Analysis of algorithms and the analysis of Selection Sort