CS 134 Lecture 10:
List Comprehensions
Announcements & Logistics

• **HW 4** due Monday at 10 pm

• **Lab 4** Part 1 check point due Wed/Thurs 10 pm
  • We will review the code for the prelab together at the start of lab

• Reminder that Midterm is March 14
  • Evening exam with two slots: 6-7.30 pm, 8-9.30 pm
  • Room TBD
  • We will have a midterm review earlier that week (in the evening)

• How to study: review lectures
  • Practice past HW and labs on pencil and paper

**Do You Have Any Questions?**
Last Time

- Introduced and used nested lists
- Did more examples of nested for loops:
  - Iterate over nested sequences and collect/filter useful statistics
- Discussed how to count using nested loops/lists
- Introduced idea of accumulation variable to find "most"
Today's Plan

- Wrap up the oscars example
- Discuss modules vs scripts
- Introduce list comprehensions
Oscar 2024 Wrap Up
def count_nominations(movie, nomination_list):
    '''Function that takes two arguments: movie (str) and nomination_list (list of lists) and returns the count (int) of the number of times movie is nominated.'''

    # initialize accumulation variable
    count = 0

    # iterate over list of nominations
    for category in nomination_list:
        for nominee in category:
            # is the movie name a prefix of nomination?
            if is_prefix(movie, nominee):
                count += 1

    return count
Exercise: most_nominations

def most_nominations(movie_list, nomination_list):
    '''Returns list of movies with most nominations'''
    most_so_far = 0  # keeps track of most # nominations
    most_list = []  # remember the movie names
    for movie in movie_list:
        num = count_nominations(movie, nomination_list)
        # found a movie with more nominations
        if num > most_so_far:
            most_so_far = num
            # remember the movie
            most_list = [movie]
        # what to do if there is a tie?
        elif num == most_so_far:
            # remember this movie as well
            most_list += [movie]
    return most_so_far
What about least nominations?

- When looking for the "maximum" among elements
  - Initialize a most_so_far variable to zero
  - Update every time we see a **bigger** value
- How would we find the "least" among elements?
  - Initialize a least_so_far variable to ____?
  - Update every time we see a **smaller** value
Modules vs Scripts
Importing Functions vs Running as a Script

• **Question.** If you only have function definitions in a file `funcs.py`, and run it as a script, what happens?
  
  ```
  % python3 funcs.py
  ```

• For testing functions, we want to call /invoke them on various test cases, in Labs, we do this in a separate file called `runtests.py`
  
  • To add function calls in `runtests.py`, we put them inside the guarded block `if __name__ == '__main__':`
    
  • The statements within this special guarded are only run when the file is run as a `script` but not when it is imported as a `module`

• Let's see an example
import foo

__name__ is set to foo
Takeaway: `if __name__ == "__main__"`

• If you want some statements (like test calls) to be run **ONLY when the file is run as a script**
  • Put them inside the guarded `if __name__ == "__main__"` block

• When we run our automatic tests on your functions we **import them** and this means name is NOT set to main
  • So nothing inside the guarded `if __name__ == "__main__"` block is executed

• This way your testing /debugging statements do not get in the way
List Comprehensions
List Patterns: Map & Filter

- When using lists and loops, there are common patterns that appear

- **Filtering:** Iterate over a list and return a new list that results from keeping only elements of the original list that satisfy some condition
  - E.g., take a list of integers `num_lst` and return a new list which contains only the even numbers in `num_lst`

- **Mapping:** Iterate over a list and return a new list that results from performing an operation on each element of original list
  - E.g., take a list of integers `num_lst` and return a new list which contains the square of each number in `num_lst`

- Python allows us to implement these patterns succinctly using list comprehensions
Mapping Example: Using Loops

- **Mapping**: Iterate over a list and return a new list that results from *performing an operation on each element* of original list.
- Example: Iterate through a sequence of numbers (e.g. list or range) and creates a new list that contains the square of the numbers.

```python
result = []
for n in range(10):
    result += [n**2]
```

- We can rewrite this loop a list comprehension in Python.
Mapping: List Comprehensions

Mapping List Comprehension (perform operation on each element)

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

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

\[
\text{for } n \text{ in range}(10):
\]

\[
\text{result} += [n**2]
\]

\[
\text{result} = [n**2 \text{ for } n \text{ in range}(10)]
\]

Note: All list comprehensions are "short hands" common for loop patterns.
Filtering Example: Using Loops

- **Filtering**: Iterate over a list and return a new list that results from keeping only elements of the original list that satisfy some condition.

- Example: Iterate through a sequence of numbers (list or range) and create a new list only containing even numbers.

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

- We can rewrite this loop a list comprehension in Python.
Filtering: List Comprehensions

Filtering List Comprehension (only keep some elements)

```python
new_list = [item for item in sequence if conditional]
```

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

Note: All list comprehensions are "short hands" common for loop patterns.
Mapping & Filtering: Using Loops

- **Mapping & Filters:** Iterate over a list and return a new list that results from *performing an operation on some elements of the original list (that satisfy some condition)*

- Example: Iterate through a sequence of numbers (list or range) and create a new list only containing the squares of the even numbers

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

- We can rewrite this loop a list comprehension in Python
result = [n**2 for n in range(10) if n%2 == 0]

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

Note: All list comprehensions are "short hands" common for loop patterns.
List Comprehensions

```python
new_list = [expression for item in sequence if conditional]
```

- Important points:
  - List comprehensions always start with an `expression` (a variable name like `item` is an expression)
  - A list comprehension is used `instead` of a list accumulation variable (that always needs to be initialized)
    - So, it always creates a `new list` that we store in var `new_list`
    - We never use `+=` inside a list comprehension
  - We `don't need to use` a list comprehension: can always write a for loop instead
    - Just a handy shortcut for common code patterns in Python
Using List Comprehensions

- **List comprehensions** are convenient when working with sequences
- Recall our list of movie names from the oscar data
- **Example**: How can we find the list of movie names that begin with a vowel?
  - **Hint**: we can use a helper function `starts_with_vowel()`
  - **Idea**:
    - Iterate over movies (list of strings)
    - For each name in list, check if first letter is a vowel
    - If it is, add name to result list
Using List Comprehensions

- List comprehensions are convenient when working with sequences.
- Assume we have a helper function `starts_with_vowel`.

```python
result = []
for m in movies:
    if starts_with_vowel(m):
        result += [m]
```

result = [m for m in movies if starts_with_vowel(m)]
Using List Comprehensions

- **List comprehensions** are convenient when working with sequences
- Assume we have a helper function `starts_with_vowel`

```python
result = []
for m in movies:
    if starts_with_vowel(m):
        result += [m]
result = [m for m in movies if starts_with_vowel(m)]
```
def starts_with_vowel(word):
    '''Takes a word (string) as input and returns True if it starts with a vowel, otherwise returns False.''
    if len(word) != 0:
        # check first letter is a vowel
        return word[0] in 'aeiouAEIOU'
    # if word is empty string
    return False
Takeaways

• For loops let us iterate over sequences *a fixed number of items*: equal to the length of the sequence

• They are useful to iterate over simple (e.g. strings) as well as nested sequences (list of lists of strings)
  • Accumulation variables let us keep track of information as we loop through sequences: e.g., counts, accumulating lists or strings, as well stats such as max and min

• List comprehensions are loop expressions that you write within a list
  • Common shorthand for mapping and filtering code patterns
  • "Pythonic" feature: not general to other languages

• Importing modules/functions does not run code in if name is main block (which is only run when the file is run as a script)