CS 134 Lecture 10:

List Comprehensions

Announcements & Logistics

- **HW 4** due Monday at 10 pm
- **Lab 4** Part 1 checkpoint: 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
    - Supplemental POGIL activities

Do You Have Any Questions?

Last Time

- Introduced and used **nested lists**
- More examples of iteration:
    - 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
- Introduce list comprehensions
- Discuss modules vs scripts

Oscar 2024 Wrap Up

Helper Function: `count_nominations`

```
def count_nominations(movie, nominations_lists):
    '''Function that takes two arguments:  movie (str) and
    nominations_lists (list of lists) and returns the count
    (int) of the number of times movie is nominated.'''

    # initialize accumulation variable
    count =
                ___

# iterate over list of nominations
for ______ in ___________:
    for _______ in ___________:
        # is the movie name a prefix of nomination?
        if is_prefix(movie, nominee):
            count += ___ # match! count the nomination
return _____
```

Helper Function: `count_nominations`

```
def count_nominations(movie, nominations_lists):
    '''Function that takes two arguments:  movie (str) and
    nominations_lists (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 nominations_lists:
        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 = ___ # 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:
            #  track movie as "most nominated so far"
            ____________________
            ____________________

        # found a movie tied for most nominations so far
        elif num == most_so_far:
            # track this movie too
            ____________________

    return most_so_far
```

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:
            #  track movie as "most nominated so far"
            most_so_far = num
            most_list = [movie]

        # found a movie tied for most nominations so far
        elif num == most_so_far:
            # track this movie too
            most_list += [movie]

    return most_so_far
```

How would find least nominations?

- When looking for the "largest" among elements
    - Initialize a `most_so_far` variable to be `0`
    - Update every time we see a ***bigger*** value (`if num > most_so_far`)
- How would we find the "least" among elements?
    - Initialize a `least_so_far` variable to be `___`?
    - Update every time we see a ***smaller*** value (`if num < least_so_far`)

Pick a number larger than largest possible value so that we **have to** find a smaller value in our iteration.

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**

A supplemental Python-specific feature

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. range of 10 integers) and create a new list that contains the square of the numbers

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

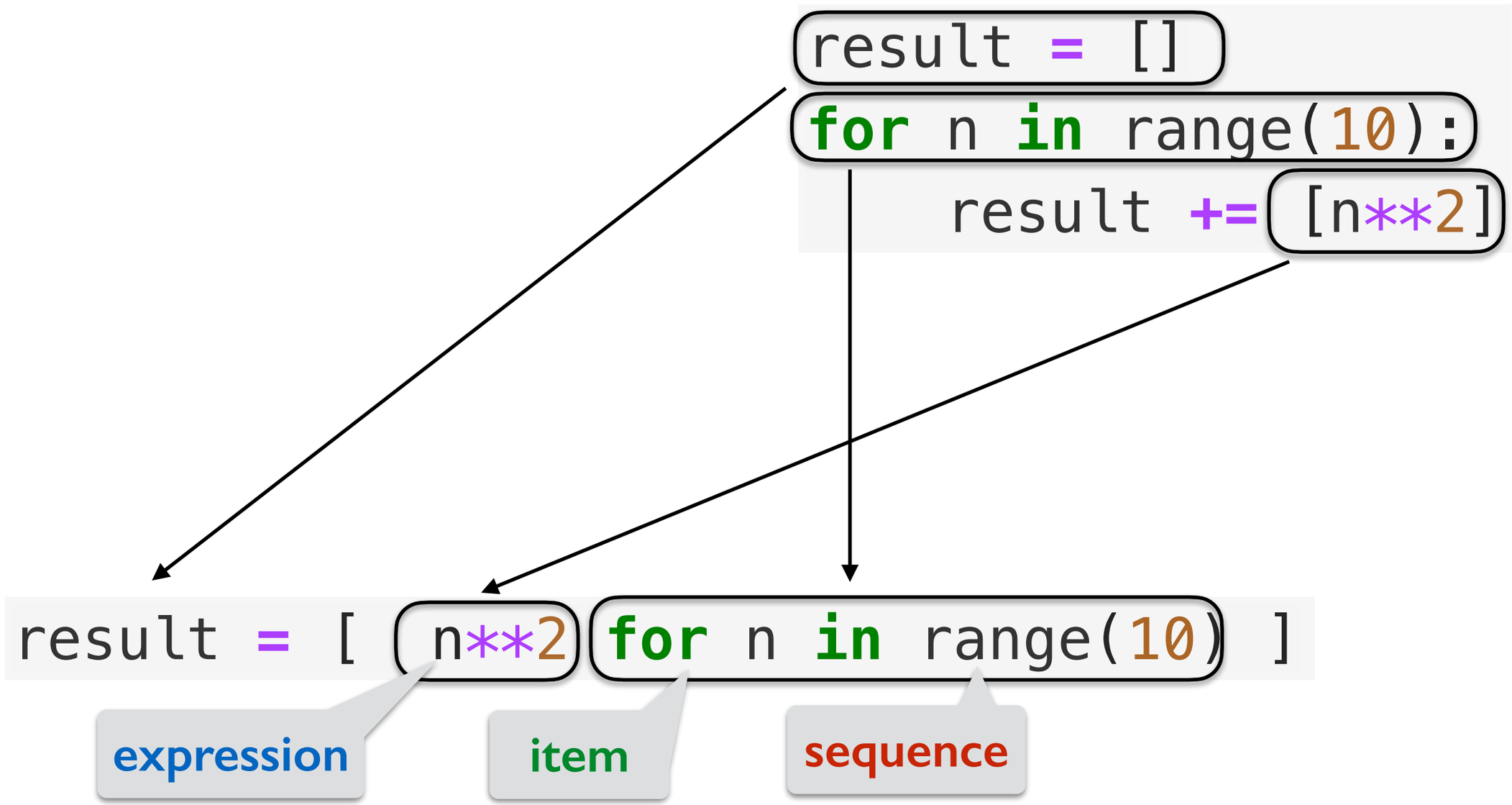
Accumulate squares in `result`

- We can rewrite this loop a list comprehension in Python

Mapping: List Comprehensions

Mapping List Comprehension (perform operation on each element)

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



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

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

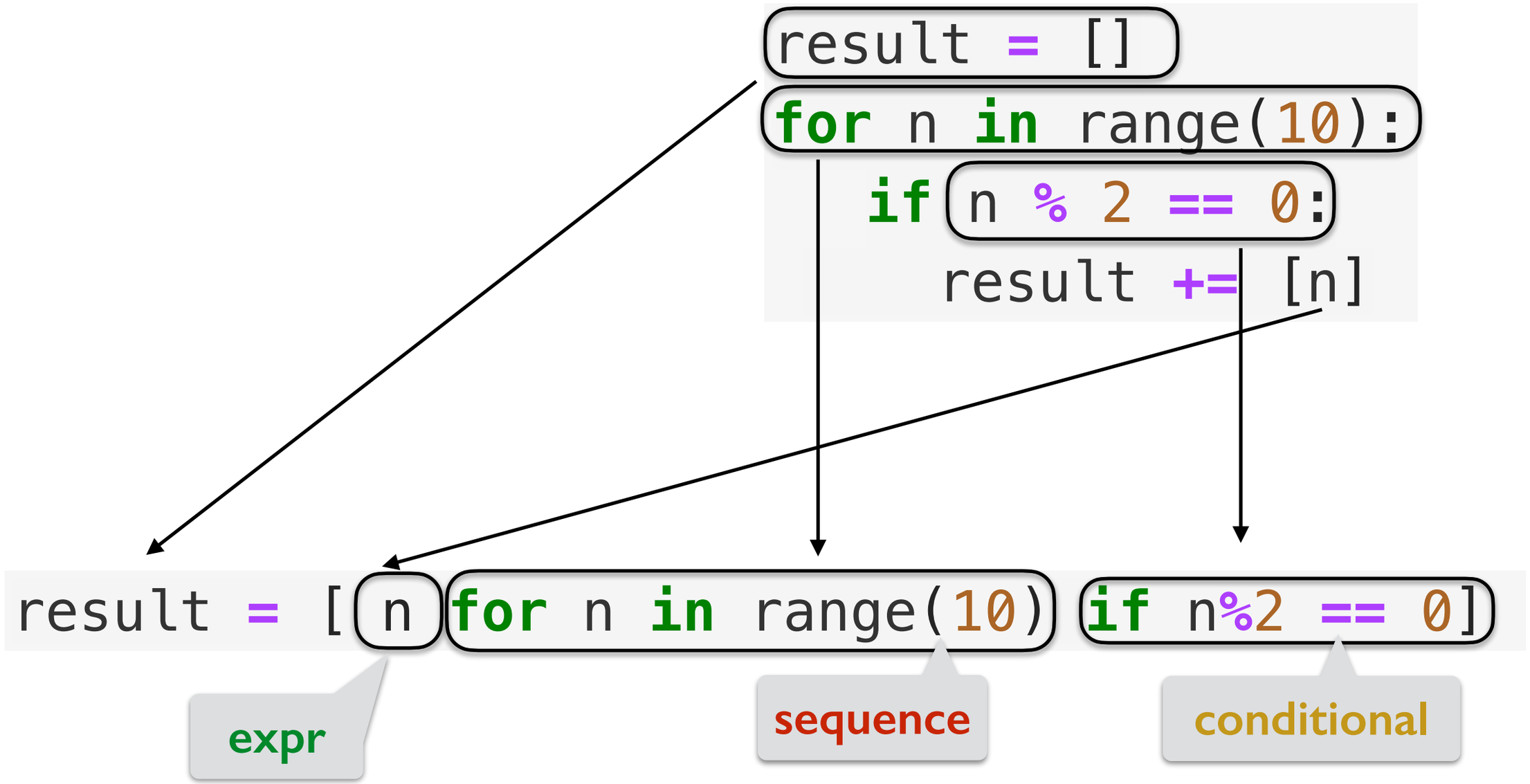
Accumulate even numbers in `result`

- We can rewrite this loop a list comprehension in Python

Filtering: List Comprehensions

Filtering List Comprehension (only keep some elements)

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



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

Mapping & Filtering: Using Loops

- **Mapping & Filtering:** 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

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

Accumulate square of even numbers in `result`

- We can rewrite this loop a list comprehension in Python

General List Comprehension

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

Can use functions or any operations here

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

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

expression **item** **sequence** **conditional**

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

List Comprehensions

```
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 can be used instead of a list accumulation variable (accumulation variables always need 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 (just an option): can always write a for loop instead
  - Just a handy shortcut for common code patterns

List Comprehensions

Mapping List Comprehension (perform operation on each element)

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

Filtering List Comprehension (only keep some elements)

```
new_lst = [item for item in sequence if conditional]
```

- Important points:
  - List comprehensions always start with an **expression** (a variable name like `item` is an expression)
  - We **never use += (append)** inside of list comprehensions
  - We can **combine mapping and filtering** into a single list comprehension:

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

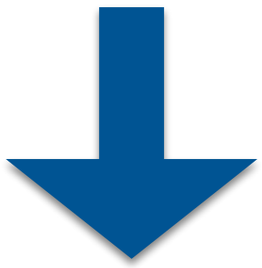
Using List Comprehensions

- **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`

```
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`

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

item

sequence

conditional

expression

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

expression

item

sequence

conditional

Helper Function

```
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
```

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

```python
# foo.py
# test the role of __name__ variable
print("__name__ is set to", __name__)
```

Running foo.py as a **script**

```
shikhasingh@Shikhas-iMac cs134 % python3 foo.py
__name__ is set to __main__

shikhasingh@Shikhas-iMac cs134 % python3
Python 3.10.0 (v3.10.0:b494f5935c, Oct  4 2021,
14:59:20) [Clang 12.0.5 (clang-1205.0.22.11)] on
darwin
Type "help", "copyright", "credits" or "license"
for more information.
>>> import foo
__name__ is set to foo
```

Importing it as a **module**

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