CS 134 Lecture 8: Nested Loops

Announcements & Logistics

- **Lab 3** due today/tomorrow at 10 pm
  - More involved than previous labs, so please utilize help hours
  - Reminder: do **NOT** use techniques not discussed in class
    - We've carefully designed the labs to require only functions & concepts *discussed in class meetings*
    - We've intentionally ordered material to emphasize algorithmic thinking and benefit your development as a computer scientist rather than as a Python-specific programmer
      - This means no string.index() or list.index()! (Why?)
- **Lab 2 graded feedback** will be returned today
- **HW 4** posted today on Glow

Do You Have Any Questions?

Last Time

- `for` loops allow us to look at each element in a sequence
  - The **loop variable** defines what the name of that element will be in the loop
  - An optional **accumulator variable** is useful for keeping a running tally of properties of interest
  - Indentation works the same as with if--statements: if it's indented under the loop, it's executed as part of the loop
- Extract subsequences with `[start:end:step]` syntax (**slicing**)
- `range` is a type of sequence that is often useful for indexing

Different problems may require different decisions with respect to loop variables, accumulator variables, and whether you need to index/slice or not!

Today's Plan

- Use more examples of the `range` sequence type
- Explore different combinations of loops
  - Loop(s) within a loop (called nesting)
- Exiting loops early
  - `break` vs. `return`

Nested Loops

Nested Loops

- A **for loop** body can contain one (or more!) additional **for loops**:
    - Called **nesting for loops**
    - Conceptually similar to nested conditionals
- Example: What do you think is printed by the following Python code?

```python
# What does this do?
def mystery_print(word1, word2):
    '''Prints something'''
    for char1 in word1:
        for char2 in word2:
            print(char1 + char2)
```

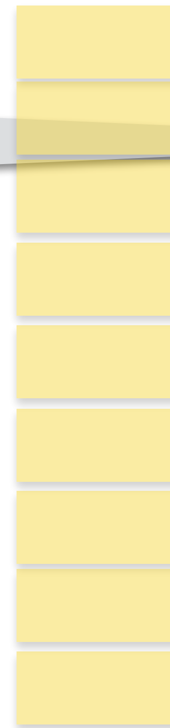
```python
mystery_print('123', 'abc')
```

```python
# What does this do?
def mystery_print(word1, word2):
    '''Prints something'''
    for char1 in word1:
        for char2 in word2:
            print(char1 + char2)
```

```python
mystery_print('123', 'abc')
```

```
1a
1b
1c
2a
2b
2c
3a
3b
3c
```

Inner loop (w/ char2 and word2) runs to completion on **each iteration** of the outer loop



```
char1 = 1 char2 = a
          char2 = b
          char2 = c
char1 = 2 char2 = a
          char2 = b
          char2 = c
char1 = 3 char2 = a
          char2 = b
          char2 = c
```

Nested Loops

- What is printed by the nested loop below?

```
# What does this print?
for char1 in ['b', 'd', 'r', 's']:
    for suffix in ['ad', 'ib', 'ump']:
        print(char1 + suffix)
```

```python
# What does this print?
for char1 in ['b', 'd', 'r', 's']:
    for suffix in ['ad', 'ib', 'ump']:
        print(char1 + suffix)
```

```
char1= 'b'     suffix = 'ad'      bad
                        'ib'      bib
                        'ump'     bump
char1= 'd'     suffix = 'ad'      dad
                        'ib'      dib
                        'ump'     dump
char1= 'r'     suffix = 'ad'      rad
                        'ib'      rib
                        'ump'     rump
char1= 's'     suffix = 'ad'      sad
                        'ib'      sib
                        'ump'     sump
```

Inner for loop runs to completion on **each iteration** of the outer for loop

Nested Loops and Ranges

Loops and Ranges to Print Patterns

We previously used a single **for loop** and a single range to **repeat** a task.

- What if we had multiple for loops and multiple ranges? The following loops print a pattern to the screen. (Look closely at the indentation!)
-

```
# what does this print?

for i in range(5):
    print('$' * i)
for j in range(5):
    print('*' * j)
```

```
# what does this print?

for i in range(5):
    print('$' * i)
    for j in range(i):
        print('*' * j)
```

What are the values of `i` and `j`?

Iterating Over Ranges

```
# what does this print?

for i in range(5):
    print('$' * i)
for j in range(5):
    print('*' * j)
```

We've seen this for loop and pattern before

Same pattern, but with '*' instead

```
        i = 0
$       i = 1
$$      i = 2
$$$     i = 3
$$$$    i = 4
        j = 0
*       j = 1
**      j = 2
***     j = 3
****    j = 4
```

These for loops are **sequential**.
One follows **after** the other.

Iterating Over Ranges

```python
# what does this print?

for i in range(5):
    print('$' * i)
for j in range(5):
    print('*' * j)
```

```python
# what does this print?

for i in range(5):
    print('$' * i)
    for j in range(i):
        print('*' * j)
```

```
        i = 0
$       i = 1
$$      i = 2
$$$     i = 3
$$$$    i = 4

        j = 0
*       j = 1
**      j = 2
***     j = 3
****    j = 4
```

On right, for loops are **nested**.
One loop is **inside** the other.

```
        i = 0
$       i = 1
            j = 0
$$      i = 2
            j = 0
*           j = 1
$$$     i = 3
            j = 0
*           j = 1
**          j = 2
$$$$    i = 4
            j = 0
*           j = 1
**          j = 2
***         j = 3
```

Iterating Over Ranges

```
# what does this print?

for i in range(5):
    print('$' * i)
    for j in range(i):
        print('*' * i)
```

i, not j!

```
                i = 0
$               i = 1
*                       j = 0
$$              i = 2
**                      j = 0
**                      j = 1
$$$             i = 3
***                     j = 0
***                     j = 1
***                     j = 2
$$$$            i = 4
****                    j = 0
****                    j = 1
****                    j = 2
****                    j = 3
```

```
# what does this print?

for i in range(5):
    print('$' * i)
    for j in range(i):
        print('*' * j)
```

```
                i = 0
$               i = 1
                        j = 0
$$              i = 2
                        j = 0
*                       j = 1
$$$             i = 3
                        j = 0
*                       j = 1
**                      j = 2
$$$$            i = 4
                        j = 0
*                       j = 1
**                      j = 2
***                     j = 3
```

Knowing How and When to Leave

Leaving a Function: `return`

We exit from a **function** using a `return` statement.

- `return` causes the execution of your code to resume at the location where the function was called (or invoked)
- `return`*(ed)* value "**replaces**" the function call

If there is no explicit `return`, the function is exited when the reaches the end of the function body, and the function implicitly returns **None**

- What happens when we have a return statement inside a loop?
  - We exit the function, so we also exit the loop!
- What happens when we have a return statement inside a nested loop?
  - We exit the function, so we exit every loop!

Example: `first_index_of()`

```
def first_index_of(word, char):
    '''Takes as input a string word and a character
    char and returns the index in word where the
    char first appears. If the char does not appear
    in word, return -1.'''

    for i in range(len(word)):
        # if the ith letter in word same as char
         if word[i] == char:
            # found first index
            return i
    return -1
```

Summary

- `Range()` is a function that returns a sequence of `int`s
  - Often used for indexing or for executing a loop a certain number of times
- Loops can be **nested** inside other loops
  - Inner loops execute once *per iteration* of their containing loop
- Return is how we exit a function
  - Return inside loops/conditionals, means you exit out of everything

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

Nested Lists

Nested Lists

- Remember, any object can be an element of a list. This includes other lists!
- That is, we can have **lists of lists** (sometimes called a two-dimensional list)!
- Suppose we have a **list of lists of strings** called `myList`

Nested Lists

- Remember, any object can be an element of a list. This includes other lists!
- That is, we can have **lists of lists** (sometimes called a two-dimensional list)!
- Suppose we have a **list of lists of strings** called `myList`
- `word = myList[row][element] (# word is a string)`
  - `row` is index into "***outer***" list (identifies ***which inner list*** we want). In other words, defines the "row" you want.
  - `element` is index into "***inner***" list (identifies ***which element*** within the inner list). In other words, defines the "column" you want.

element

```
myList = [ ['cat', 'frog'],
           ['dog', 'toad'],
           ['cow', 'duck'] ]
```

row

myList[1][0]?

'dog'

Nested Loops

- Trace through the code below:

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts

list_of_lists = [[1,2,3], [4,5,6], [7,8,9]]
print(mystery2(list_of_lists))
```

Nested Loops

new_lstlsts	row	new_row	item
[]	[1,2,3]	[]	

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

```
lst_lsts = [[1,2,3],
            [4,5,6],
            [7,8,9]]
```

Nested Loops

new_lstlsts	row	new_row	item
[]	[1,2,3]	[]	
		[1]	1
	[1,2,3]	[1,4]	2
	[1,2,3]	[1,4,9]	3
[[1,4,9]]			

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

```
lst_lsts = [[1,2,3],
            [4,5,6],
            [7,8,9]]
```

Nested Loops

new_lstlsts	row	new_row	item
[]	[1,2,3]	[]	
		[1]	1
	[1,2,3]	[1,4]	2
	[1,2,3]	[1,4,9]	3
[[1,4,9]]	[4,5,6]	[]	
		[16]	4
	[4,5,6]	[16,25]	5
	[4,5,6]	[16,25,36]	6
[[1 ,4 ,9], [16,25,36]]			

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

```
lst_lsts = [[1,2,3],
            [4,5,6],
            [7,8,9]]
```

Nested Loops

new_lstlsts	row	new_row	item
[]	[1,2,3]	[]	
		[1]	1
	[1,2,3]	[1,4]	2
	[1,2,3]	[1,4,9]	3
[[1,4,9]]	[4,5,6]	[]	
		[16]	4
	[4,5,6]	[16,25]	5
	[4,5,6]	[16,25,36]	6
[[1 ,4 ,9], [16,25,36]]	[7,8,9]	[]	
		[49]	7
	[7,8,9]	[49,64]	8
[[1 ,4 ,9], [16,25,36], [49,64,81]]	[7,8,9]	[49,64,81]	9

```python
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

```python
lst_lsts = [[1,2,3],
            [4,5,6],
            [7,8,9]]
```

Nested Loops

new_lstlsts	row	new_row	item
[]	[1,2,3]	[]	
		[1]	1
	[1,2,3]	[1,4]	2
	[1,2,3]	[1,4,9]	3
[[1,4,9]]	[4,5,6]	[]	
		[16]	4
	[4,5,6]	[16,25]	5
	[4,5,6]	[16,25,36]	6
[[1 ,4 ,9], [16,25,36]]	[7,8,9]	[]	
		[49]	7
	[7,8,9]	[49,64]	8
[[1 ,4 ,9], [16,25,36], [49,64,81]]	[7,8,9]	[49,64,81]	9

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts
```

```
lst_lsts = [[1,2,3],
            [4,5,6],
            [7,8,9]]
```

Nested Loops

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts

list_of_lists = [[1,2,3], [4,5,6], [7,8,9]]
print(mystery2(list_of_lists))
```

Accumulation variable

Accumulation variable

Note the [ ]

Why?!

Nested Loops

```
def mystery2(lst_lsts):
    new_lstlsts = []
    for row in lst_lsts:
        new_row = []
        for item in row:
            new_row = new_row + [item*item]
        new_lstlsts = new_lstlsts + [new_row]
    return new_lstlsts

list_of_lists = [[1,2,3], [4,5,6], [7,8,9]]
print(mystery2(list_of_lists))
```

Accumulation variable

Accumulation variable

Note the [ ]

Why?!

Why 2 accumulation variables?!

The square brackets ensure that we're adding a `list` to a `list`!

The inner loop accumulates the items for the row, the outer loop accumulates the rows

What would be a good function name for `mystery2`?

Something like `power_table`

Loops Takeaways

- `for` loops allow us to look at each element in a sequence
  - The **loop variable** defines what the name of that element will be in the loop
  - An optional **accumulator variable** is useful for keeping a running tally of properties of interest
  - Indentation works the same as with if--statements: if it's indented under the loop, it's executed as part of the loop
- **Nested `for` loops** allow us to do the same for multiple lists (often lists of lists or lists of strings)

Different problems may require different decisions with respect to loop variables, accumulator variables, and whether you need a nested loop or not!