CS134:
Range & Nested Lists/Loops
Announcements & Logistics

• **Lab 3** due today/tomorrow
  • More involved than previous labs, so please utilize help hours
  • Reminder: do **NOT** use utilities 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?)

• **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
- Can extract subsequences using `[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`
• **Sequences** in Python represent *ordered collections of elements*: e.g., lists, strings, ranges, etc.

• Strings are immutable sequences of characters

• Ranges are immutable sequences of numbers

• Lists can be **heterogenous** (strings, ints, floats, etc)
  
  • Example: `my_list = ["Hello", 42, 23.5, True]`
  
  • In CS, we use **zero-indexing**, so we say that 'Hello' is at index 0, 42 is at index 1, and so on

• We can access each character of a list using these **indices**
## Sequence Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq[i]</td>
<td>The i'th item of seq, when starting with 0</td>
</tr>
<tr>
<td>seq[si:ee]</td>
<td>slice of seq from si to ee</td>
</tr>
<tr>
<td>seq[si:ee:s]</td>
<td>slice of seq from si to ee with step s</td>
</tr>
<tr>
<td>len(seq)</td>
<td>length of seq</td>
</tr>
<tr>
<td>seq1 + seq2</td>
<td>The concatenation of seq1 and seq2</td>
</tr>
<tr>
<td>x in seq</td>
<td>True if x is contained within seq</td>
</tr>
<tr>
<td>x not in seq</td>
<td>False if x is contained within seq</td>
</tr>
</tbody>
</table>
Iterating Over Ranges

A common use of a `range` is to repeatedly execute some task

- With a `for` loop and `range(n)`, can repeat a loop `n` times

```python
# what does this print?
for i in range(5):
    print('$' * i)
```

Looks a lot like `[0, 1, 2, 3, 4]`
Using Range For Parallel Iteration

• Ranges also give a convenient way for iterating over two lists in parallel
• Say we wanted to iterate over two lists:
  • `chars = ['a', 'b', 'c'] and nums = [1, 2, 3]`
  • And form a new list `['a1', 'b2', 'c3']`
• Here's how we'd do it

```python
chars = ['a', 'b', 'c']
nums = [1, 2, 3]
# initialize accumulation variable
# for each item in chars
    # add current char to matching num
    # accumulate in a list

>>> char_nums
['a1', 'b2', 'c3']
```
Using Range For Parallel Iteration

- This also a really convenient way for iterating over two lists in parallel
- Say we wanted to iterate over two lists
- \texttt{chars = ['a', 'b', 'c']} and \texttt{nums = [1, 2, 3]}
- And form a new list ['a1', 'b2', 'c3']
- Here's how we'd do it

```python
chars = ['a', 'b', 'c']
nums = [1, 2, 3]
char_nums = []

for i in range(0, len(chars)):
    cnum = chars[i] + str(nums[i])
    char_nums = char_nums + [cnum]

>>> char_nums
['a1', 'b2', 'c3']
```
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)

mystery_print('123', 'abc')
```
# What does this do?

def mystery_print(word1, word2):
    '''Prints something'''
    for char1 in word1:
        for char2 in word2:
            print(char1 + char2)

mystery_print('123', 'abc')
Nested Loops

• What is printed by the nested loop below?

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

```python
for letter in ['b', 'd', 'r', 's']:
    for suffix in ['ad', 'ib', 'ump']:
        print(letter + suffix)
```

- **letter = 'b'**
  - suffix = 'ad'
  - bad
  - 'ib'
  - bib
  - 'ump'
  - bump

- **letter = 'd'**
  - suffix = 'ad'
  - dad
  - 'ib'
  - dib
  - 'ump'
  - dump

- **letter = 'r'**
  - suffix = 'ad'
  - rad
  - 'ib'
  - rib
  - 'ump'
  - rump

- **letter = 's'**
  - suffix = 'ad'
  - sad
  - 'ib'
  - sib
  - 'ump'
  - sump

Inner loop (w/ suffixes) runs to completion on each iteration of the outer loop (w/ prefixes)
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?
```python
for i in range(5):
    print('$' * i)
for j in range(5):
    print('*' * j)
```

• # what does this print?
```python
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?

```python
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.

These for loops are **sequential**. One follows **after** the other.
# what does this print?

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

# what does this print?

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

On right, for loops are **nested**. One loop is **inside** the other.
# what does this print?

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

$ 
$i = 0$
$* $
$ $$
$i = 1$
$**$
$ *$
$ $$$$
$i = 2$
$ ***$
$ *$
$ $$$$ i = 3$
$ ***$
$ *$
$ $$$$$ i = 4$
$ ***$
$ *$
$ $*$, not $j$!

# what does this print?

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

$ 
$i = 0$
$* $
$ $$
$i = 1$
$**$
$ *$
$ $$$$
$i = 2$
$ ***$
$ *$
$ $$$$ i = 3$
$ ***$
$ *$
$ $*$, not $j$!
Knowing How and When to Leave
Leaving a Function

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` can also communicate a value that "replaces" the function call

When we do not include an explicit `return` statement, we exit the function when our execution 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!
Leaving a Loop

We can exit from a loop using a `break` statement.

- `break` causes the execution of your code to resume at the location immediately following the loop body.
- If your code breaks out of a nested loop, execution may begin a new iteration of the "outer" loop.

```python
def first_locations_of(string_list, char):
    '''Returns a list that contains the index where char first appears within each string in string_list'''
    locations = []
    for string in string_list:
        i = 0
        for c in string:
            if c == char:
                break  # we've found the index
            i += 1
        locations += [i]
    return locations
```
Leaving a Loop

```python
def first_locations_of(string_list, char):
    '''Returns a list that contains the index where char first appears within each string in string_list'''
    locations = []
    for string in string_list:
        i = 0
        for c in string:
            if c == char:
                break  # we've found the index
                i += 1
        locations += [i]
    return locations

>>> first_locations_of(["eat", "more", "vegetables"], "e")
[0, 3, 1]
```
break Controversy

- **break** is a part of python, but its use is often discouraged for **stylistic** reasons

- "Jumping" around in our code makes it hard to reason about what our program is doing

- We can often structure our code in a way that using break is unnecessary, so avoid it if possible

- Part of becoming a good programmer is understanding the **spirit** of the rules (and when to break them!)

```python
def first_locations_of(string_list, char):
    '''Returns a list that contains the index where char first appears within each string in string_list'''
    locations = []
    for string in string_list:
        locations += [first_location_of(string, char)]
    return locations
```
def first_location_of(string, char):
    '''Returns the index where char first appears within string. If it does not appear, returns len(string)'''
    i = 0
    for c in string:
        if c == char:
            return i
        i += 1
    return i

def first_locations_of(string_list, char):
    '''Returns a list that contains the index where char first appears within each string in string_list'''
    locations = []
    for string in string_list:
        locations += [first_location_of(string, char)]
    return locations

By making the "loop" a "function", we can return instead of "break"
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
# foo.py
# test the role of __name__ variable
print("__name__ is set to", __name__)
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
Summary

• Range is a flexible sequence type 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

• Break is how we exit a loop
  • We can often rewrite our code to avoid using break