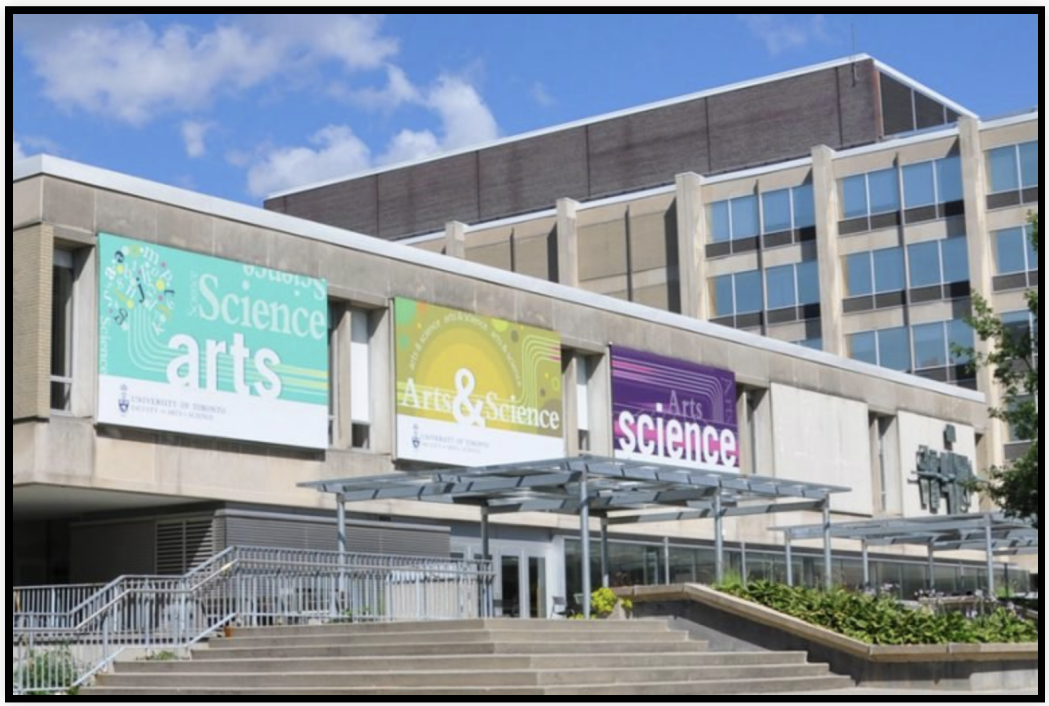
CSC110 Lecture 13: More with For Loops

David Liu and Tom Fairgrieve, Department of Computer Science

*Navigation tip for web slides: press **?** to see keyboard navigation controls.*

UofT Building of the Day

Sidney Smith Hall



Home of the Faculty of Arts and Science registrar's office

Announcements and today's plan

- Assignment 2 has been posted—submit what you have finished!
  - Check out the A2 FAQ
  - Review advice on academic integrity
- PythonTA initial survey
  - has been announced on Quercus

Today you'll learn to...

1. Write index-based for loops to solve new problems.
2. Write Python functions that work on nested data using nested loops or loops with separate function calls.
3. Choose the appropriate for loop structure to use to solve a given problem.
4. Use PythonTA to display loop accumulation tables when loops run.

Index-based for loops (recap from prep)

`my_sum`, element-based

```
def my_sum(numbers: list[int]) -> int:
    """..."""
    # ACCUMULATOR sum_so_far: keep track of the
    # sum of the numbers seen so far in the loop.
    sum_so_far = 0

    for number in numbers:
        sum_so_far = sum_so_far + number

    return sum_so_far
```

This implementation of `my_sum` uses an element-based for loop.

The loop variable `number` refers to an element of the collection of numbers.

`my_sum`, index-based

```
def my_sum(numbers: list[int]) -> int:
    """..."""
    # ACCUMULATOR sum_so_far: keep track of the
    # sum of the numbers seen so far in the loop.
    sum_so_far = 0

    for i in range(0, len(numbers)):
        number = numbers[i]
        sum_so_far = sum_so_far + number

    return sum_so_far
```

This implementation of `my_sum` uses an index-based for loop.

The loop variable `i` refers to an index of the collection of numbers.

When to use index-based for loops

Two cases where an index-based for loop should be used:

1. When the loop needs to take into account the position of each element.
2. When the loop needs to iterate over two sequences in parallel.

1. When position matters

```
def count_adjacent_repeats(string: str) -> int:
    """Return the number of repeated adjacent characters
    in string.

    >>> count_adjacent_repeats('look')
    1
    """
    repeats_so_far = 0

    for i in range(0, len(string)):
        if string[i] == string[i + 1]:
            repeats_so_far = repeats_so_far + 1

    return repeats_so_far
```

2. When iterating over two lists in parallel

```
def count_money(counts: list[int], denoms: list[float]) -> float:
    """Return the total amount of money for the given coin
    counts and denominations.

    >>> count_money([2, 4, 3], [0.05, 0.10, 0.25])
    1.25
    """
    money_so_far = 0.0

    for i in range(0, len(counts)):
        money_so_far = money_so_far + counts[i] * values[i]

    return money_so_far
```

Exercise 1: Looping with indexes

Nested loops

Story so far

We use a for loop to iterate over a collection of data.

```
for <element> in <collection>:
    ...
```

In some cases, each `<element>` is itself a collection!

`sum` revisited

```
def sum_all(lists_of_numbers: list[list[int]]) -> int:
    """Return the sum of all the numbers in the given
    lists_of_numbers.

    >>> sum_all([[1, 2, 3], [10, -5], [100]])
    111
    """
```

```
    sum_so_far = 0

    for numbers in lists_of_numbers:
        # numbers is a list[int]
        sum_so_far = sum_so_far + ...

    return sum_so_far
```

Updating the accumulator using `my_sum`

```
def sum_all(lists_of_numbers: list[list[int]]) -> int:
    """..."""
    sum_so_far = 0

    for numbers in lists_of_numbers:
        # numbers is a list[int]
        sum_so_far = sum_so_far + my_sum(numbers)

    return sum_so_far
```

Updating the accumulator using a nested loop

```
def sum_all(lists_of_numbers: list[list[int]]) -> int:
    """..."""
    sum_so_far = 0

    for numbers in lists_of_numbers:
        # numbers is a list[int]
        for number in numbers:
            # number is an int
            sum_so_far = sum_so_far + number

    return sum_so_far
```

A for loop body can contain any type of statement—even another for loop! This is called a **nested for loop**.

- The `for numbers in lists_of_numbers` loop is the **outer loop**.
- The `for number in numbers` loop is the **inner loop**.

```
sum_all([[1, 2, 3], [10, -5], [100]])
```

Outer loop iteration	Outer loop variable (`list_of_numbers`)	Inner loop iteration	Inner loop variable (`number`)	Accumulator (`sum_so_far`)
0				0
1	[1, 2, 3]	0		0
1	[1, 2, 3]	1	1	1
1	[1, 2, 3]	2	2	3
1	[1, 2, 3]	3	3	6
2	[10, -5]	0		6
2	[10, -5]	1	10	16
2	[10, -5]	2	-5	11
3	[100]	0		11
3	[100]	1	100	111

Inner and outer accumulators

```
def multiply_adjacent_repeats(strings: list[str]) -> int:
    """Return the product of the numbers of times in each
    given string that two adjacent characters are equal.

    >>> multiply_adjacent_repeats([
            'look',      # 1 repeat
            'Davviid',   # 2 repeats
            'bbccaaa'    # 4 repeats
        ])
    8
    """
```

Inner and outer accumulators

```
def multiply_adjacent_repeats(strings: list[str]) -> int:
    """Return the product of the numbers of times in each
    given string that two adjacent characters are equal.
    """
```

```
    product_so_far = 1

    for s in strings:
        repeats = count_adjacent_repeats(s)
        product_so_far = product_so_far * repeats

    return product_so_far
```

Using nested loops

```
def multiply_adjacent_repeats(strings: list[str]) -> int:
    product_so_far = 1

    for s in strings:
        # inner loop calculates the number of adjacent
        # repeats in s
        repeats_so_far = 0
        for i in range(0, len(s) - 1):
            if s[i] == s[i + 1]:
                repeats_so_far = repeats_so_far + 1

        product_so_far = product_so_far * repeats_so_far

    return product_so_far
```

Exercise 2: Nested loops

DEMO: PythonTA and Loop Accumulation Tables (Read more in 5.8)

Notes on For Loops and Code Design

Two loop patterns

Accumulation pattern

```
<x>_so_far = <default_value>

for element in collection:
    <x>_so_far = ... <x>_so_far ... element ...

return <x>_so_far
```

Early return pattern

```
for element in collection:
    if <condition>:
        return <early_value>

return <default_value>
```

Elements vs. indexes

Element-based for loop

```
for item in collection:
    ...
```

Use when you can process each element individually, regardless of the element's position (index).

Index-based for loop

```
for i in range(0, len(collection)
    ...
```

Use when you need each element's index, or when processing two sequences in parallel.

Working with nested data

Nested loop

```
for item1 in collection1:
    for item2 in collection2:
        ...
```

Loop with helper function

```
for item1 in collection1:
    ... helper_function(item1, collection2) ...
```

Use nested loops if they're short and simple; pull out the inner loop into a helper function if the inner loop is getting complicated.

Comprehensions vs. for loops

Comprehension

```
sum([f(element) for element in collection if is_good(element)])
```

The typical structure for a comprehension is:

1. Range over a collection. (`for element in collection`)
2. Perform filtering, if required. (`if is_good(element)`)
3. Perform transformation, if required. (`f(element)`, or just `element`)
4. Perform an aggregation on the result (`sum`)

Comprehensions vs. for loops

```
sum([f(element) for element in collection if is_good(element)])
```

For loop

```
sum_so_far = 0

for element in collection:
    if is_good(element):
        sum_so_far = sum_so_far + f(element)
```

For loops and comprehensions both enable ranging over a collection, filtering elements, and transforming elements.

For loops have more flexibility for computing an aggregated value, at the expense of expressing the computation more indirectly.

Comprehensions vs. for loops

Use a comprehension when:

- You are computing and returning a new collection (`list`/`set`/`dict`)
- You can use a simple built-in aggregation function (e.g., `sum`, `max`, etc.)

Use a for loop when:

- You need to perform a custom aggregation.
- You want to use an early return.
- You want to repeat code that causes side effects like printing to the Python console or drawing a shape in Pygame.

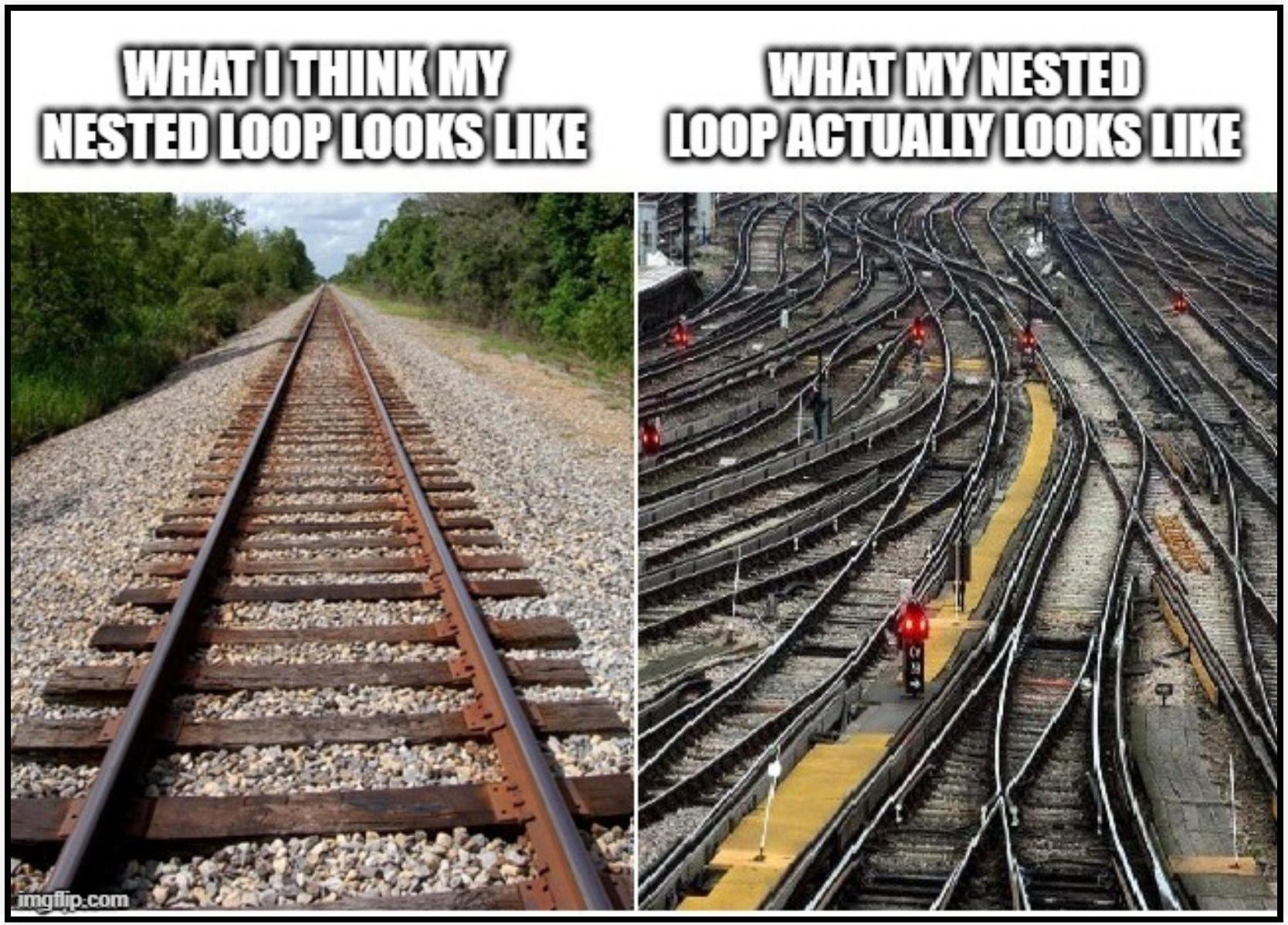
Summary

Today you learned to...

1. Write index-based for loops to solve new problems.
2. Write Python functions that work on nested data using nested loops or loops with separate function calls.
3. Choose the appropriate for loop structure to use to solve a given problem.
4. Use PythonTA to display loop accumulation tables when loops run.

Homework

- Work on Assignment 2
- Complete the PythonTA Survey


WHAT I THINK MY NESTED LOOP LOOKS LIKE
WHAT MY NESTED LOOP ACTUALLY LOOKS LIKE
imgflip.com