CSC110 Lecture 12: Repeated Execution with For Loops

David Liu, Department of Computer Science

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

Announcements and today's plan

- Assignment 2 has been posted—please start early!
  - Check out the A2 FAQ
  - Additional TA office hours (starting today)
  - Review advice on academic integrity

University is closed on Monday for Thanksgiving! No lecture or office hours on Monday.

Story so far (control flow)

Single expressions in the Python console:

```
>>> 2 + 3
5
```

Story so far (control flow)

Sequential statement execution:

```
def calculate_distance(x1: float, y1: float,
                       x2: float, y2: float) -> float:
    """..."""
    dx_squared = (x2 - x1) ** 2
    dy_squared = (y2 - y1) ** 2
    exact_distance = (dx_squared + dy_squared) ** 0.5
    return round(exact_distance, 1)
```

Story so far (control flow)

Conditional statement execution:

```
def get_status(scheduled: int, estimated: int) -> str:
    """..."""
    if estimated <= scheduled:
        return 'On time'
    else:
        return 'Delayed'
```

Now...

Repeated statement execution!

Today you'll learn to...

1. Repeat execution of Python statements using a for loop.
2. Define and apply the accumulator pattern with for loops.
3. Write code combining if statements and for loops.
4. Define and apply the early return pattern with for loops.
5. Describe how loops behave across different data types: `list`, `set`, `dict`, `str`, `range`.

Implementing `sum`

```
def my_sum(numbers: list[int]) -> int:
    """Return the sum of the given numbers.

    >>> my_sum([10, 20, 30])
    60
    """
```

Suppose we wanted to implement `my_sum` without using the built-in `sum` function.

Version 1: assume length 3, one expression

```
def my_sum(numbers: list[int]) -> int:
    """Return the sum of the given numbers.

    >>> my_sum([10, 20, 30])
    60
    """
    return numbers[0] + numbers[1] + numbers[2]
```

`my_sum` Version 2: assume length 3, using variable reassignment

```
    sum_so_far = 0

    sum_so_far = sum_so_far + numbers[0]
    sum_so_far = sum_so_far + numbers[1]
    sum_so_far = sum_so_far + numbers[2]
```

Statement	`sum_so_far`
`sum_so_far = 0`	0
`sum_so_far = sum_so_far + numbers[0]`	10
`sum_so_far = sum_so_far + numbers[1]`	30
`sum_so_far = sum_so_far +`	60

Key idea: we need to repeat the statement

```
sum_so_far = sum_so_far + numbers[...]
```

once for each number in `numbers`.

Comprehensions to the rescue...?

Comprehensions let us evaluate some code once per element of a collection.

```
def my_sum(numbers: list[int]) -> int:
    sum_so_far = 0

    [sum_so_far = sum_so_far + number for number in number

    return sum_so_far
```

```
[<expression> for <variable> in <collection>]
```

The left-hand part of a comprehension must be an expression, not an arbitrary statement!

For loops

Key idea: we need to repeat the statement

```
sum_so_far = sum_so_far + numbers[...]
```

For loop (syntax)

The **for loop** is a compound Python statement that causes repeated execution of one or more statements.

```
for <variable> in <collection>:
    <statement>
    ...              # Any number of additional statements
```

The statements indented within the for loop are called the **body of the for loop**, or “loop body” for short.

For loop (execution)

```
for <variable> in <collection>:
    <statement>
    ...             # Any number of additional statements
```

When the Python interpreter executes a for loop:

1. The loop variable is assigned the first value in the collection.
2. The statements in the loop body are executed.
3. Steps 1 and 2 are repeated for the second element of the collection, then the third, etc. until all elements of the collection have been assigned to the loop variable exactly once.

Each repetition of the loop body is called a **loop iteration**.

We say that the for loop **iterates over** `<collection>`.

my_sum Version 3: arbitrary length, using for loop

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

    for number in numbers:
        sum_so_far = sum_so_far + number

    return sum_so_far
```

- No repeated code: the Python interpreter repeats the loop body for us
- No list indexing: the Python interpreter extracts the elements of `numbers` for us

The accumulator pattern

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

    for number in numbers:
        sum_so_far = sum_so_far + number

    return sum_so_far
```

`sum_so_far` is called an **accumulator variable**: the value it refers to changes at each loop iteration.

It stores an accumulated result based on the values "seen so far" by the loop.

Loop accumulation table

```
def my_sum(numbers: list[int]) -> int:
    sum_so_far = 0

    for number in numbers:
        sum_so_far = sum_so_far + number

    return sum_so_far
```

Iteration	`number`	`sum_so_far`
0	N/A	0
1	10	10
2	20	30
3	30	60

Exercise 1: Practice with for loops

The accumulator pattern

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

    for number in numbers:
        sum_so_far = sum_so_far + number

    return sum_so_far
```

The accumulator pattern (generalized)

```
# ACCUMULATOR <x>_so_far: keep track of ...
<x>_so_far = <default_value>

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

return <x>_so_far
```

- `<x>` should be named according to the value being accumulated
- `<default_value>` is the "starting value" of the accumulator
  - (usually) equal to the value that should be computed for an empty collection
- `... <x>_so_far ... element ...` is the code that updates the accumulator
  - can be more than one statement!

A variation: counting elements

```
def my_len(items: list) -> int:
    """Return the size of the given list.

    >>> my_len([10, 20, 30])
    3
    """
```

```
    # ACCUMULATOR len_so_far: keep track of the
    # number of the elements in numbers seen so far.
    len_so_far = ...

    for element in items:
        len_so_far = ... len_so_far ... element ...

    return len_so_far
```

A variation: counting elements

```
def my_len(items: list) -> int:
    len_so_far = 0

    for _ in items:
        len_so_far = len_so_far + 1

    return len_so_far
```

```
def my_sum(numbers: list[int]) -> int:
    sum_so_far = 0

    for number in numbers:
        sum_so_far = sum_so_far + numbe

    return sum_so_far
```

Multiple accumulators

```
def my_avg(numbers: list[int]) -> int:
    """Return the average of the given numbers.

    Preconditions:
    - numbers != []
    """
```

```
    return sum(numbers) / len(numbers)
```

```
def my_avg(numbers: list[int]) -> int:
    """Return the average of the given numbers.

    Preconditions:
    - numbers != []
    """
    len_so_far = 0
    sum_so_far = 0
```

```
    for number in numbers:
        len_so_far = len_so_far + 1
        sum_so_far = sum_so_far + number
```

```
    return sum_so_far / len_so_far
```

Exercise 2: Marriage licenses, re-revisited

Conditional execution in for loops

Filtering revisited

Suppose we want to calculate the sum of the even numbers in a collection.

```
def sum_evens(numbers: list[int]) -> int:
    """Return the sum of the given numbers that are even.

    >>> sum_evens([10, 3, 4])
    14
    """
```

```
    return sum(
        [number for number in numbers if number % 2 == 0]
    )
```

sum_evens using a loop (1)

```
def sum_evens(numbers: list[int]) -> int:
    """Return the sum of the given numbers that are even."
    # ACCUMULATOR sum_so_far: keep track of the
    # sum of the elements in numbers seen so far.
    sum_so_far = 0

    for number in numbers:
        sum_so_far = sum_so_far + number

    return sum_so_far
```

Problem: we only want to add `number` to the accumulator if `number` is even.

sum_evens using a loop (2)

```
def sum_evens(numbers: list[int]) -> int:
    """Return the sum of the given numbers that are even."
    # ACCUMULATOR sum_so_far: keep track of the
    # sum of the elements in numbers seen so far.
    sum_so_far = 0

    for number in numbers:
        if number % 2 == 0:
            sum_so_far = sum_so_far + number

    return sum_so_far
```

Using if statement in the loop, our code performs a **conditional update of the accumulator**.

Existential search

Given a list of strings, return whether at least one string contains the string `'cool'`.

```
def any_contains_cool(strings: list[str]) -> bool:
    """Return whether at least one given string contains the string 'cool

    >>> any_contains_cool(['David', 'is', 'very cool', 'of', 'course'])
    True
    """
```

```
    return any({'cool' in string for string in strings})
```

```
def any_contains_cool(strings: list[str]) -> bool:
    """Return whether at least one given string contains the string 'cool

    >>> any_contains_cool(['David', 'is', 'very cool', 'of', 'course'])
    True
    """
    # ACCUMULATOR <x>_so_far: keep track of ...
    <x>_so_far = <default_value>

    for string in strings:
        <x>_so_far = ... <x>_so_far ... string ...

    return <x>_so_far
```

In this case, what does the accumulator keep track of?

```
def any_contains_cool(strings: list[str]) -> bool:
    """Return whether at least one given string contains the string 'cool

    >>> any_contains_cool(['David', 'is', 'very cool', 'of', 'course'])
    True
    """
    # ACCUMULATOR cool_so_far: keep track of
    # whether 'cool' is in any string seen so far.
    cool_so_far = <default_value>

    for string in strings:
        cool_so_far = ... cool_so_far ... string ...

    return cool_so_far
```

What should the initial value of the accumulator be?

```
def any_contains_cool(strings: list[str]) -> bool:
    """Return whether at least one given string contains the string 'cool

    >>> any_contains_cool(['David', 'is', 'very cool', 'of', 'course'])
    True
    """
    # ACCUMULATOR cool_so_far: keep track of
    # whether 'cool' is in any string seen so far.
    cool_so_far = False

    for string in strings:
        cool_so_far = ... cool_so_far ... string ...

    return cool_so_far
```

How should we update the accumulator inside the loop?

```
def any_contains_cool(strings: list[str]) -> bool:
    """Return whether at least one given string contains the string `'coo

    >>> any_contains_cool(['David', 'is', 'very cool', 'of', 'course'])
    True
    """
    # ACCUMULATOR cool_so_far: keep track of
    # whether 'cool' is in any string seen so far.
    cool_so_far = False

    for string in strings:
        if 'cool' in string:
            cool_so_far = True

    return cool_so_far
```

Improving `any_contains_cool`

Consider the loop accumulation table for

```
>>> any_contains_cool(['David', 'is', 'very cool', 'of', 'course
```

Iteration	`string`	`cool_so_far`
0	N/A	False
1	`'David'`	False
2	`'is'`	False
3	`'very cool'`	True
4	`'of'`	True
5	`'course'`	True

Iteration	`string`	`cool_so_far`
0	N/A	False
1	`'David'`	False
2	`'is'`	False
3	`'very cool'`	True
4	`'of'`	True
5	`'course'`	True

`cool_so_far` only ever changes its value (at most) once, from `False` to `True`.

As soon as `cool_so_far` changes to `True`, we know what the final return value will be: we don't need to check the remaining strings!

`any_contains_cool` using an early return

```
def any_contains_cool(strings: list[str]) -> bool:
    """Return whether at least one given string contains the string `'coo

    >>> any_contains_cool(['David', 'is', 'very cool', 'of', 'course'])
    True
    """
    for string in strings:
        if 'cool' in string:
            return True

    return False
```

This for loop does not use an accumulator, but does use the **early return** pattern, where the loop has a conditional return statement that can stop the loop before all iterations are complete.

Looping over other data types

Every data type that can be used as the `<collection>` in a comprehension can also be used as the `<collection>` in a for loop.

```
[... for <variable> in <collection>]
```

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

set

```
for element in my_set:
    ...
```

`element` represents a single element of the set

str

```
for char in my_string:
    ...
```

`char` represents a single character of the string

range

```
for n in range(..., ...):
    ...
```

`n` represents a single number in the range

dict

```
for key in my_dict:
    ...
```

`key` represents a single KEY in the dictionary

(Can use `my_dict[key]` to look up the corresponding value)

Exercise 3: More loop practice

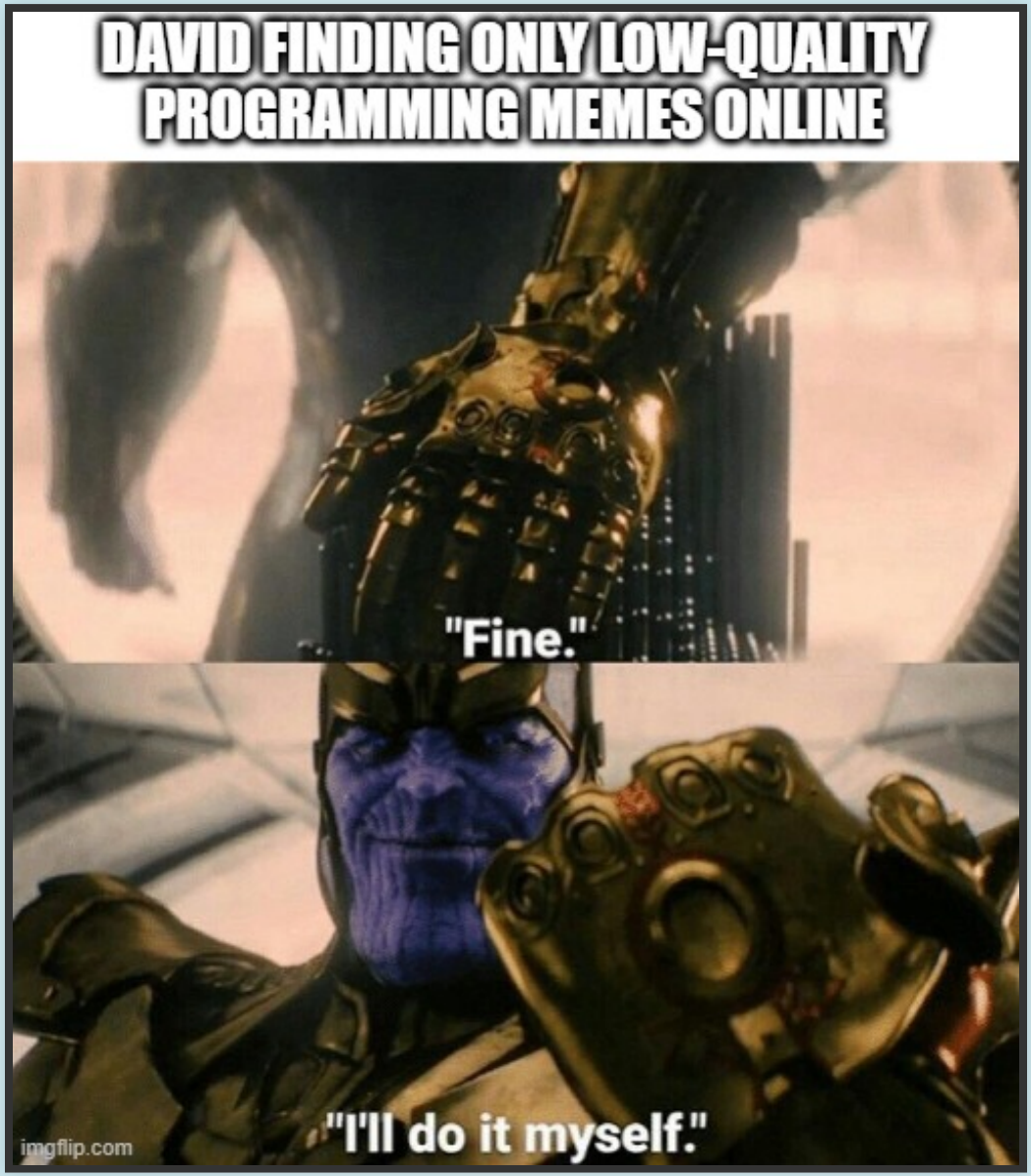
Summary

Today you learned to...

1. Repeat execution of Python statements using a for loop.
2. Define and apply the accumulator pattern with for loops.
3. Write code combining if statements and for loops.
4. Define and apply the early return pattern with for loops.
5. Describe how loops behave across different data types: `list`, `set`, `dict`, `str`, `range`.

Homework

- Work on **Assignment 2**!
- Prep 5 to be posted after class
  - Note: due date extended to Tuesday at 9am (Thanksgiving)
- Readings from today: 5.4, 5.5
- Readings for next class: 5.6, 5.7, 5.8


DAVID FINDING ONLY LOW-QUALITY PROGRAMMING MEMES ONLINE
"Fine."
"I'll do it myself."
imgflip.com