CSC110 Lecture 24: Analyzing Built-In Data Type Operations

David Liu and Tom Fairgrieve, Department of Computer Science

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

Announcements and Today's Plan

Announcements

- Assignment 4 has been posted
  - Check out the A4 FAQ (+ corrections)
    - Important simplifying precondition for Part 4b, `find_collision_len_times_sum`
  - Additional TA office hours
  - Review advice on academic integrity

Story so far

Up to this point, we've focused on running-time analysis where the complexity has been mainly due to **loops**.

The individual expressions/statements have been constant time (other than comprehensions):

- Arithmetic and comparison operations on numbers ($+$, $<$)
- Assignment statements (all data types)
- Calling `print` on numbers; calling `len` on collections
- Returning from a function

Today, we'll study the running time of individual operations on collection data types (`list`/`set`/`dict`) and data classes.

Today you'll learn to...

1. State the running times of operations on built-in collection data types and data classes.
2. Explain these running times for `list` operations based on how Python implements lists.
3. Analyze the running time of functions that use these operations.
4. Analyze the running time of functions that have multiple arguments (of different sizes).

Running time of list operations

Suppose we want to add an element to a list:

Add to the back:

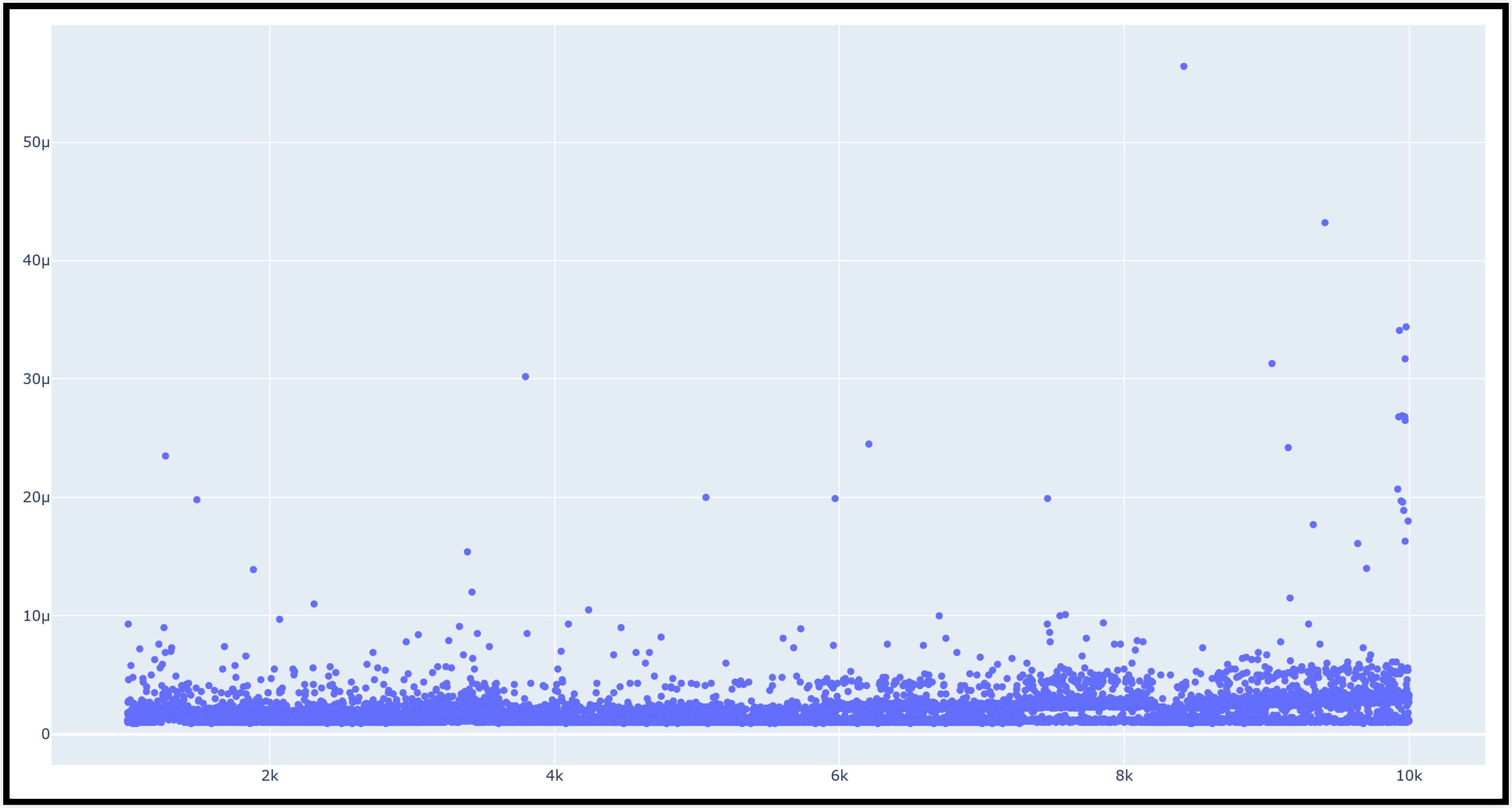
```
>>> lst = [0] * 10000000
>>> list.append(lst, 1234)
```

Add to the front:

```
>>> lst = [0] * 10000000
>>> list.insert(lst, 0, 1234)
```

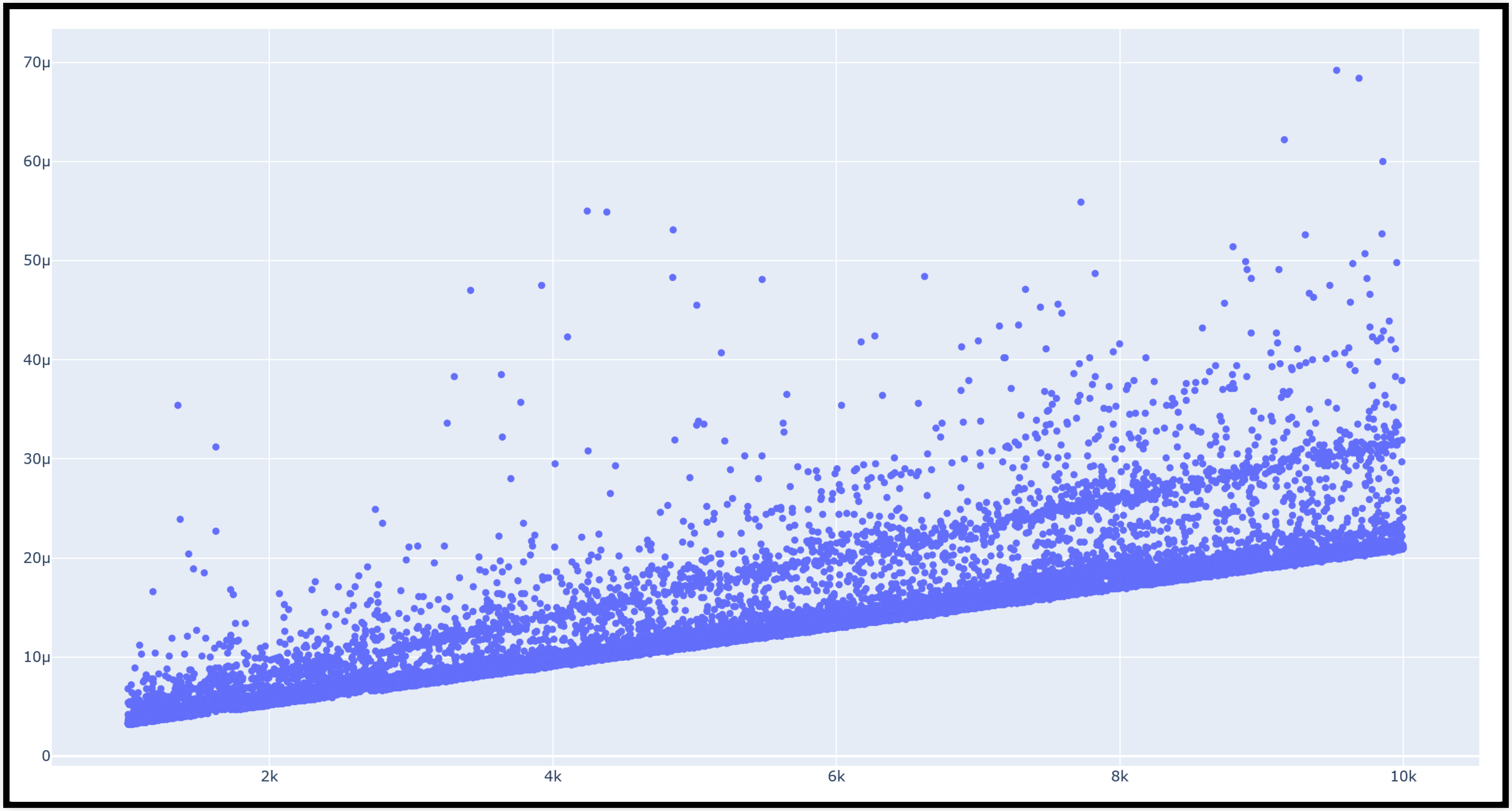
List length vs. time taken

```
list.append(lst, 1234)
```

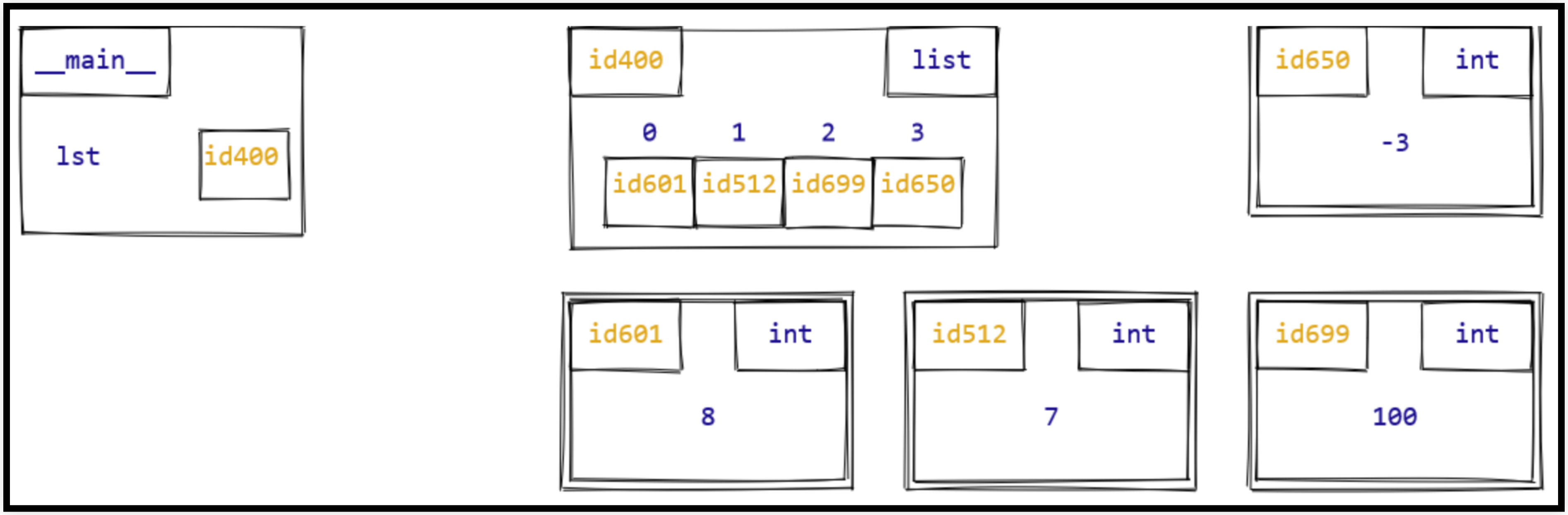


List length vs. time taken

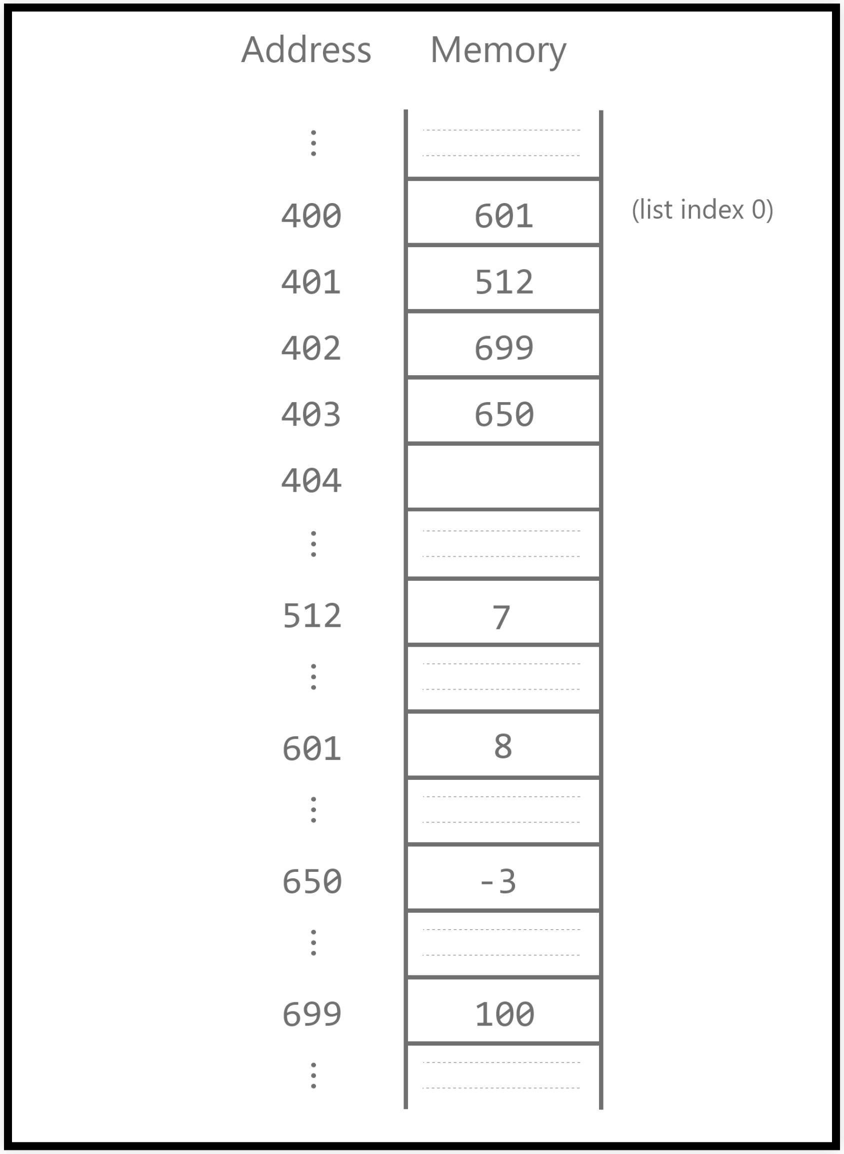
```
list.insert(lst, 0, 1234)
```



list: object-based memory model diagram



list: linear memory model diagram



How the Python interpreter stores lists

Each Python list has references to its elements stored in a contiguous block of memory.

(contiguous = consecutive memory locations with no gaps)

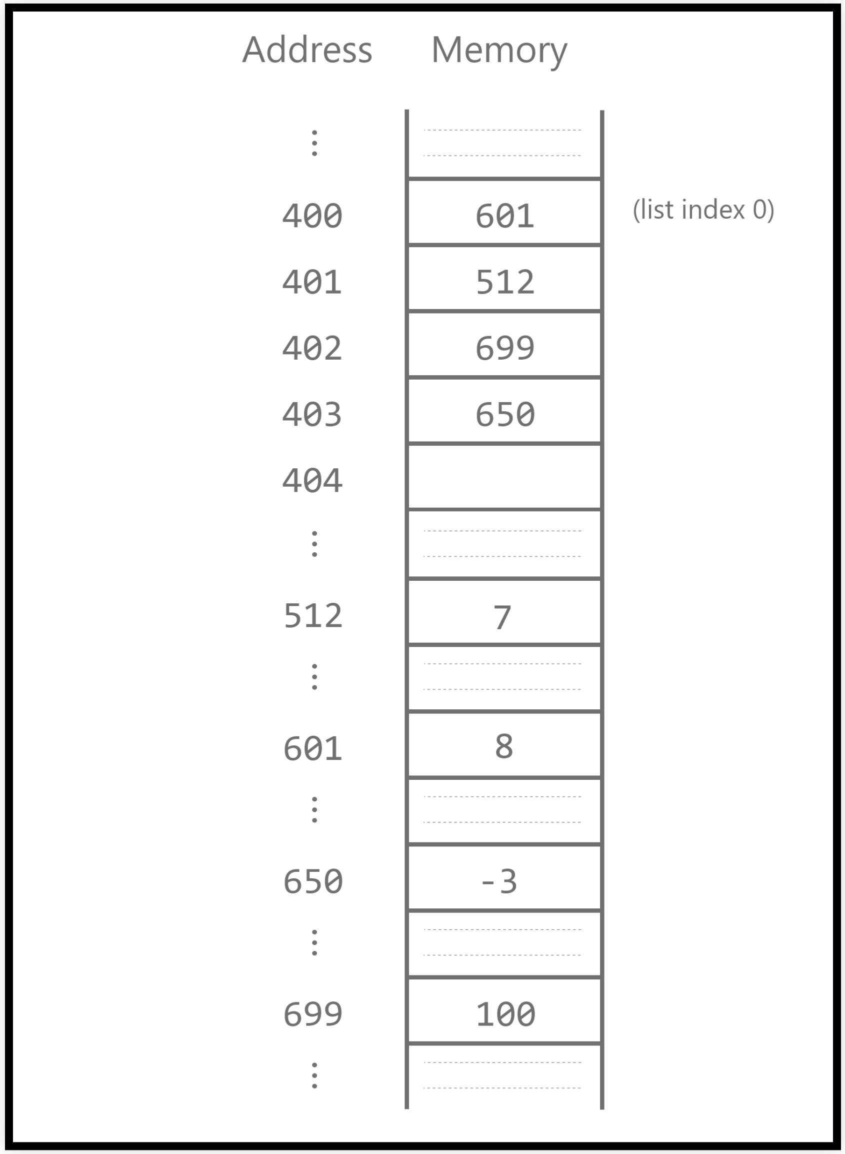
This is called an **array-based list implementation**.

Benefits: constant-time indexing!

To lookup the value of `lst[i]`:

1. Get the starting address of the list.
   - e.g., 400
2. Add `i` to the starting address. That block contains the reference to `lst[i]`.
   - e.g., `lst[3]` is at address 400 + 3 = 403.

List indexing (`lst[i]`) is $\Theta(1)$, **independent** of the size of the list or the index `i`.



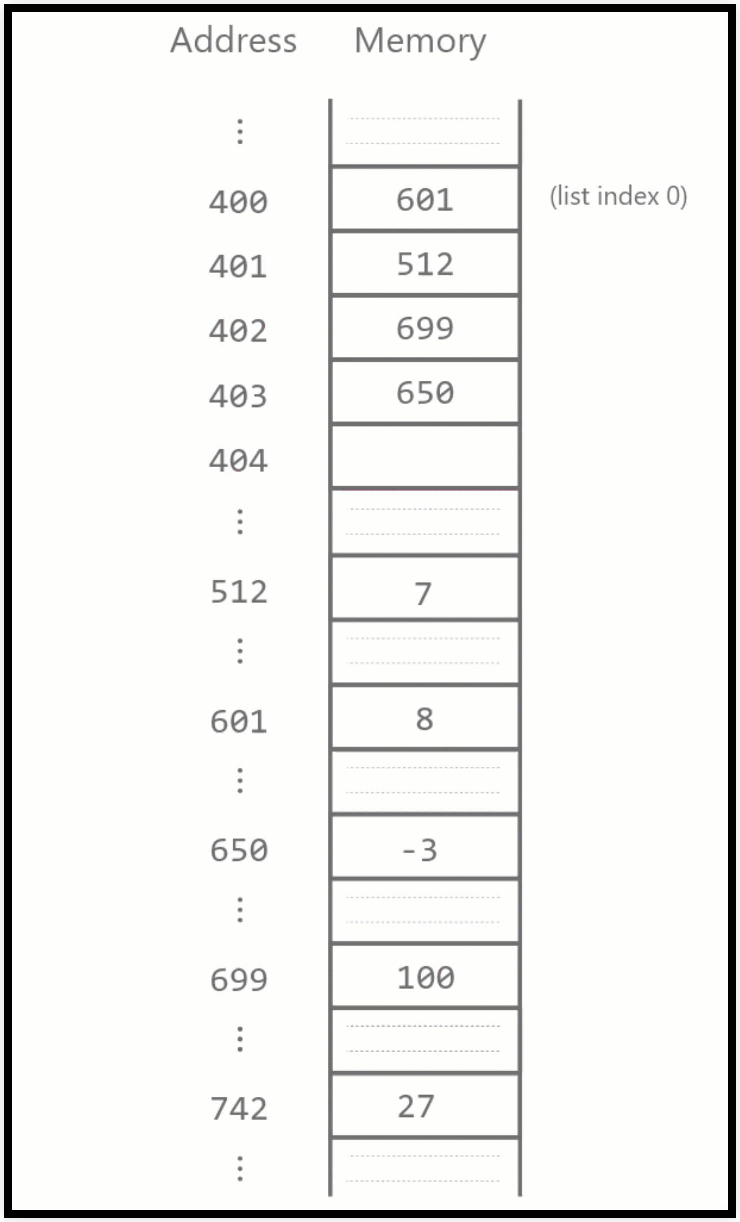
Costs: insertion/deletion can be slow

All modifications to the list must preserve the contiguity of the list elements in memory.

When an element is inserted into a list, all elements after it must be shifted over to make room.

When an element is deleted from a list, all elements after it must be shifted back to fill in the gap.

list: animation of list.insert(lst, 2, 27) steps



For a list of length n, insertion/deletion at index i takes $\Theta(n - i)$ time.

- When $i = n - 1$ (end of the list), running time is $\Theta(1)$.
  - `list.append(lst, item)`
  - `list.pop(lst)`
- When $i = 0$, (front of list), running time is $\Theta(n)$.
  - `list.insert(lst, 0, item)`
  - `list.pop(lst, 0)`

Summary

Operation	Running time
List indexing (`lst[i]`)	$\Theta(1)$
List index assignment (`lst[i] = ...`)	$\Theta(1)$
List insertion at index i (`list.insert(lst, i, ...)`)	$\Theta(n - i)$
List deletion at index i (`list.pop(lst, i)`)	$\Theta(n - i)$
List insertion at end (`list.append(lst, ...)`)	$\Theta(1)$
List deletion at end (`list.pop(lst)`)	$\Theta(1)$

Exercise 1: Running time of list operations

Analysing running time for functions with multiple parameters

```
def my_extend(lst1: list, lst2: list) -> None:
    """Add each element in lst2 to the end of lst1."""
    for item in lst2:
        list.append(lst1, item)
```

Analysis.

Let n_1 be the length of `lst1` and n_2 be the length of `lst2`.

The loop iterates n_2 times, and each time takes constant time (1 step).

The total running time is $n_2 \cdot 1 = n_2$ steps, which is $\Theta(n_2)$.

```python
def my_concat(lst1: list, lst2: list) -> list:
    """Return a new list built from lst1 and lst2."""
    new_list = []

    for item in lst1:                        # Loop 1
        list.append(new_list, item)
    for item in lst2:                        # Loop 2
        list.append(new_list, item)

    return new_list
```

Analysis. Let n_1 be the length of `lst1` and n_2 be the length of `lst2`.

- The initial assignment statement takes 1 step.
- Loop 1 takes n_1 steps (n_1 iterations, 1 step per iteration)
- Loop 2 takes n_2 steps (n_2 iterations, 1 step per iteration)
- The return statement takes constant time.

The total running time is $1 + n_1 + n_2 + 1 = n_1 + n_2 + 2$, which is $\Theta(n_1 + n_2)$.

Exercise 2: Running-time analysis with multiple parameters

Sets, dictionaries, and data classes

Sets

Set operations:

- `x in my_set`
- `set.add`
- `set.remove`

Each of these operations take $\Theta(1)$—constant time!

Sets are implemented in Python using hash tables, which are based on arrays.

Dictionaries

Dictionary operations:

- key search (`k in my_dict`)
- key lookup/assignment (`my_dict[k]`, `my_dict[k] = ...`)

Also $\Theta(1)$, very similar implementation to sets!

Data classes

Data class operations:

- attribute lookup (`david.age`)
- attribute assignment (`david.age = ...`)

Also $\Theta(1)$, very similar implementation to dictionaries (and sets)!

Exercise 3: Sets, dictionaries, and data classes (if time)

Some built-in aggregation functions

Most aggregation functions take $\Theta(n)$ time (where n is the size of the collection).

`sum`, `max`, `min`

`len` is special: it takes $\Theta(1)$ time—independent of the size of the collection!

Behind the scenes: a “hidden” size attribute stored by the Python interpreter for every collection object.

`any`/`all` are special, because they're implemented with an early return.

- `any` can stop as soon as it encounters a `True`
- `all` can stop as soon as it encounters a `False`

The running time can vary, depending on the contents of the collection.

To be continued (and more!) next class…

Summary

Today you learned to...

1. State the running times of operations on built-in collection data types and data classes.
2. Explain these running times for `list` operations based on how Python implements lists.
3. Analyze the running time of functions that use these operations.
4. Analyze the running time of functions that have multiple arguments (of different sizes).

Homework

- Readings:
  - Review: 9.5, 9.6
  - From today: 9.7
  - For Thursday: 9.8
- Work on Assignment 4!