LAB 06

Mutability, Iterators

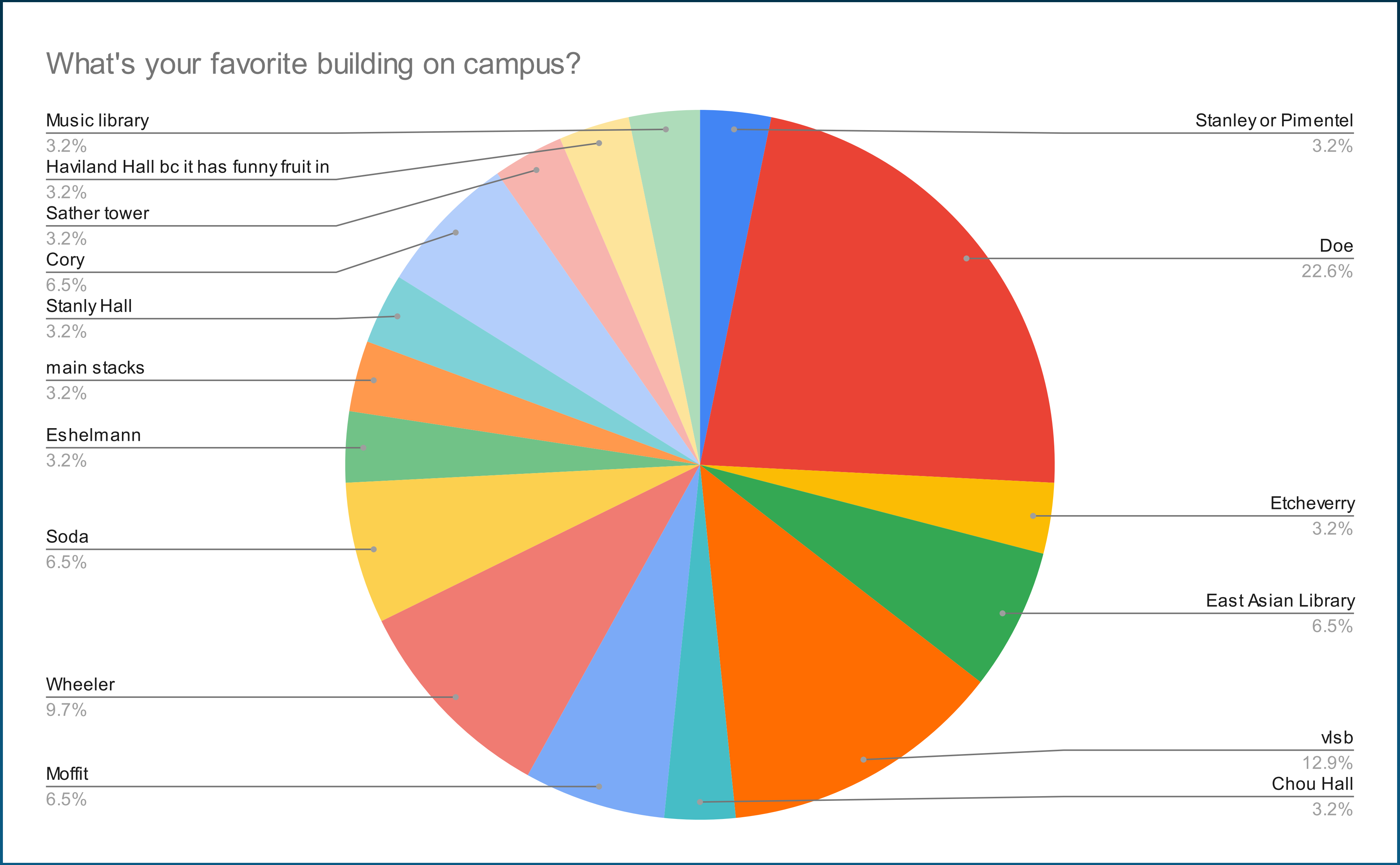
Mingxiao Wei
mingxiaowei@berkeley.edu

Feb 28, 2023

LOGISTICS 🏡

- Lab 06 due tomorrow 03/01
- Homework 04 due this Thu 03/01
    - The first problem is a survey asking for your mid-semester feedback, which is **mandatory**. Your feedback for me will be **anonymized** before they are sent to me. So feel free to share anything! I'd love to hear about your opinions and make the section better for y'all :)
- If you have issues with your discussion/lab scores on Gradescope, please email me!

FROM LAST TIME 👀



AI MINI-LECTURE TIME 🥳

Now let's welcome one of our fav AIs * Evelyn Cheng to give a mini-lecture on mutability and iterators!

Slides are here

* Don't worry Jeremy and Jessica I'll say this too when you two mini-lecture :)

NOW IT'S LAB TIME 🤠

- Get started on the lab and raise your hand whenever you need help!
- Get to know your neighbors and collaborate if you'd like!
- Slides: go.cs61a.org/mingxiao-index
- Leave any anonymous feedback here: go.cs61a.org/mingxiao-anon

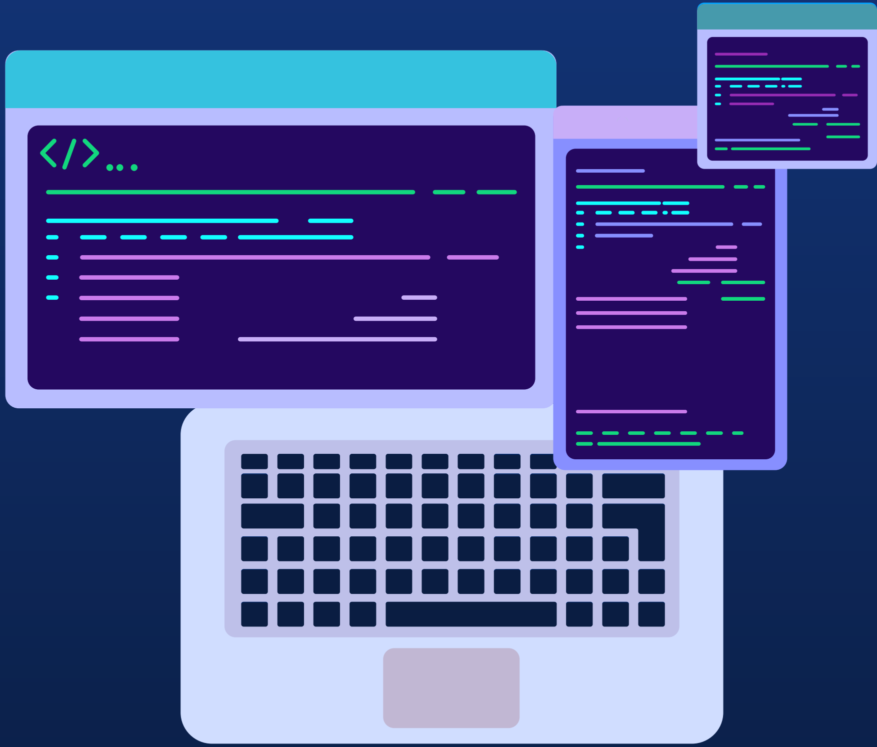
AND REMEMBER TO GET CHECKED OFF! 👒

---

go.cs61a.org/mingxiao-att

The secret phrase is ...
(NOT 3 dots! I'll announce it 🙉)

Lab 6: Mutability & Iterators



01

Mutability

Mutable vs Immutable

Mutable

Definition: Contents or state can be changed

- Lists
- Dictionaries

Immutable

Definition: Can't be changed once created

- Numeric Types
- Tuples
- Strings

Important: We can reassign values, but we can't change the original value

Mutability

```
>>> a = [4, 3]

>>> a[1] = 100

>>> a

[4, 100]
```

Immutability

```
>>> c = (10, 20)

>>> c[1] = 100

TypeError: ‘tuple’ object doesn’t support item
assignment

>>> c

(10, 20)

>>> x = 4

>>> x = 6 #reassigning values

>>> x

6
```

List Mutation

List Mutation Overview

lst.method(arg) → dot notation

append(el)	Add el to the end of the list → returns None
extend(lst)	Extend the list by concatenating it with lst → returns None
insert(i, el)	Insert el at index i, doesn’t replace any existing elements but shifts elements → returns None
remove(el)	Removes first occurrence of el in list → returns None or errors if el is not in the list
pop(i)	Remove element at index i → returns removed element

Append Vs Extend

append(element)

Definition: Add element to the end of the list
- returns None
- Element can be of any type
- If element is a list, will insert the list as a nested list

```
>>> lst = [7, 8]
>>> lst.append(100)
>>> lst
[7, 8, 100]
>>> print(lst.append([5, 6]))
None
>>> lst
>>> [7, 8, 100, [5, 6]]
```

extend(lst)

Definition: Extend the list by concatenating it with lst
- returns None
- lst has to be a list
- extend(lst) = append(element) if only adding one element when lst = [element]

```
>>> lst = [7, 8]
>>> lst.extend([100])
>>> lst
[7, 8, 100]
>>> print(lst.extend([5, 6]))
None
>>> lst
>>> [7, 8, 100, 5, 6]
```

COMPETITORS

insert(i, element)

Definition: insert element at index i

- returns None
- element can be of any type
- Doesn’t replace any elements → shifts index of everything after inserted element by one

```
>>> lst = [5, 6, 7, 8, 9, 10]
>>> lst.insert(3, ‘cs’)
>>> lst
[5, 6, 7, ‘cs’, 8, 9, 10]
```

remove(element)

Definition: removes first occurrence of element in list

- Errors if element is not in the list
- returns None

```
>>> lst = [10, 8, 7, 8, 9, 10]
>>> lst.remove(8)
>>> lst
[10, 7, 8, 9, 10]
>>> lst.remove(2)
ValueError: list.remove(x): x
not in list
```

pop(i) / pop()

Definition: remove and return element at index i

- i is optional → if no arguments passed in, will automatically remove and return element at index len(lst) - 1

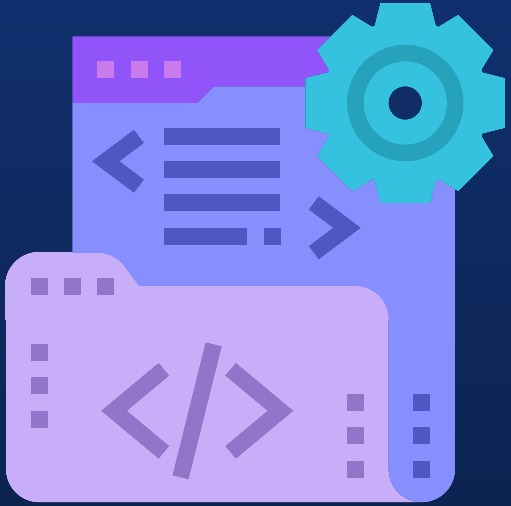
```
>>> lst = [5, 8, 7, 8, 9, 10]
>>> lst.pop(2)
7
>>> lst
[5, 8, 8, 9, 10]
>>> lst.pop()
10
```

Important Notes

- Can also mutate list through lst[index] = element
- lst += second_lst
  - Same as lst.extend(second_lst) → mutates lst
  - *Note:* different than lst = lst + second_lst → this creates a copy of lst, attaches second_lst to the copy, and returns the copied list
    - Doesn't change lst
- Don't iterate through a list and then mutate the list during the iteration
  - To fix this:
    - try creating a copy of the list and then iterate through the copy
    - Iterate through the list indices → change the indexes when appropriate
- All methods except for pop return None
  - If need to return a mutated list, make sure to mutate the list and then return the mutated list → don't return with the mutation method, it will return none!!

02

Iterators



Iterable vs Iterator

Iterable

- Any object that can be iterated through
- for loops work on any object that is iterable
- An object on which calling iter function returns an iterator

```
for elem in iterable:
    #do something
```

Iterator

- An object that allows us to iterate through an iterable
- Keeps track of which element is next in the sequence

```
iterator = iter(iterable)
try:
   while True:
       elem = next(iterator)
       # do something
except StopIteration:
   pass
```

Built in iter function called on iterable to create iterator

Gets next element in sequence

Occurs when next is called but there's no more elements in the iterator

Analogy

Books

- An iterable is like a book
- An iterator is like a bookmark
- Calling iter on a book gives you a new bookmark
- Calling iter on a bookmark gives you the bookmark itself with no changes
- Calling next on the bookmark (iterator) moves it to the next page, but doesn't change the pages/contents in the book (iterable)
  - Calling next on the book (iterable) wouldn't make sense
- It's possible to have multiple bookmarks that are independent of each other

Methods

- Calling iter() on an iterable creates and returns a corresponding iterator
    - Calling iter() on an iterable multiple times returns a new iterator each time with distinct states
- Calling next() on an iterator gets the next element from the iterator
    - Will error if you call next on the iterable directly
- Calling iter() on an iterator returns the same iterator without any change
- Note: all iterators are iterables, but not all iterables are iterators

```
>>> lst = [1, 2, 3, 4]
>>> next(lst)                # Calling next on an iterable
TypeError: 'list' object is not an iterator
>>> list_iter = iter(lst) # Creates an iterator for the list
>>> list_iter
<list_iterator object ...>
>>> next(list_iter)          # Calling next on an iterator
1
>>> next(list_iter)          # Calling next on the same iterator
2
>>> next(iter(list_iter)) # Calling iter on an iterator returns itself
3
>>> list_iter2 = iter(lst)
>>> next(list_iter2)         # Second iterator has new state
1
>>> next(list_iter)          # First iterator is unaffected by second iterator
4
>>> next(list_iter)          # No elements left!
StopIteration
>>> lst                      # Original iterable is unaffected
[1, 2, 3, 4]
```

Iterable Uses

- range(start, end): creates an iterable of ascending integers from start (inclusive) to end (exclusive)
- Built-in functions that take in iterables and return useful results:
  - map(f, iterable) - Creates an iterator over f(x) for x in iterable
  - filter(f, iterable) - Creates an iterator over x for each x in iterable if f(x)
  - zip(iterables*) - Creates an iterator over co-indexed tuples with elements from each of the iterables
  - reversed(iterable) - Creates an iterator over all the elements in the input iterable in reverse order
  - list(iterable) - Creates a list containing all the elements in the input iterable
  - tuple(iterable) - Creates a tuple containing all the elements in the input iterable
  - sorted(iterable) - Creates a sorted list containing all the elements in the input iterable
  - reduce(f, iterable) - Must be imported with functools. Apply function of two arguments f cumulatively to the items of iterable, from left to right, so as to reduce the sequence to a single value.

Note: Call next() on the returned iterables to access the values

Thank you!

Attendance form:
go.cs61a.org/mingxiao-att