DISCUSSION 06

Mutability, Iterators, Generators

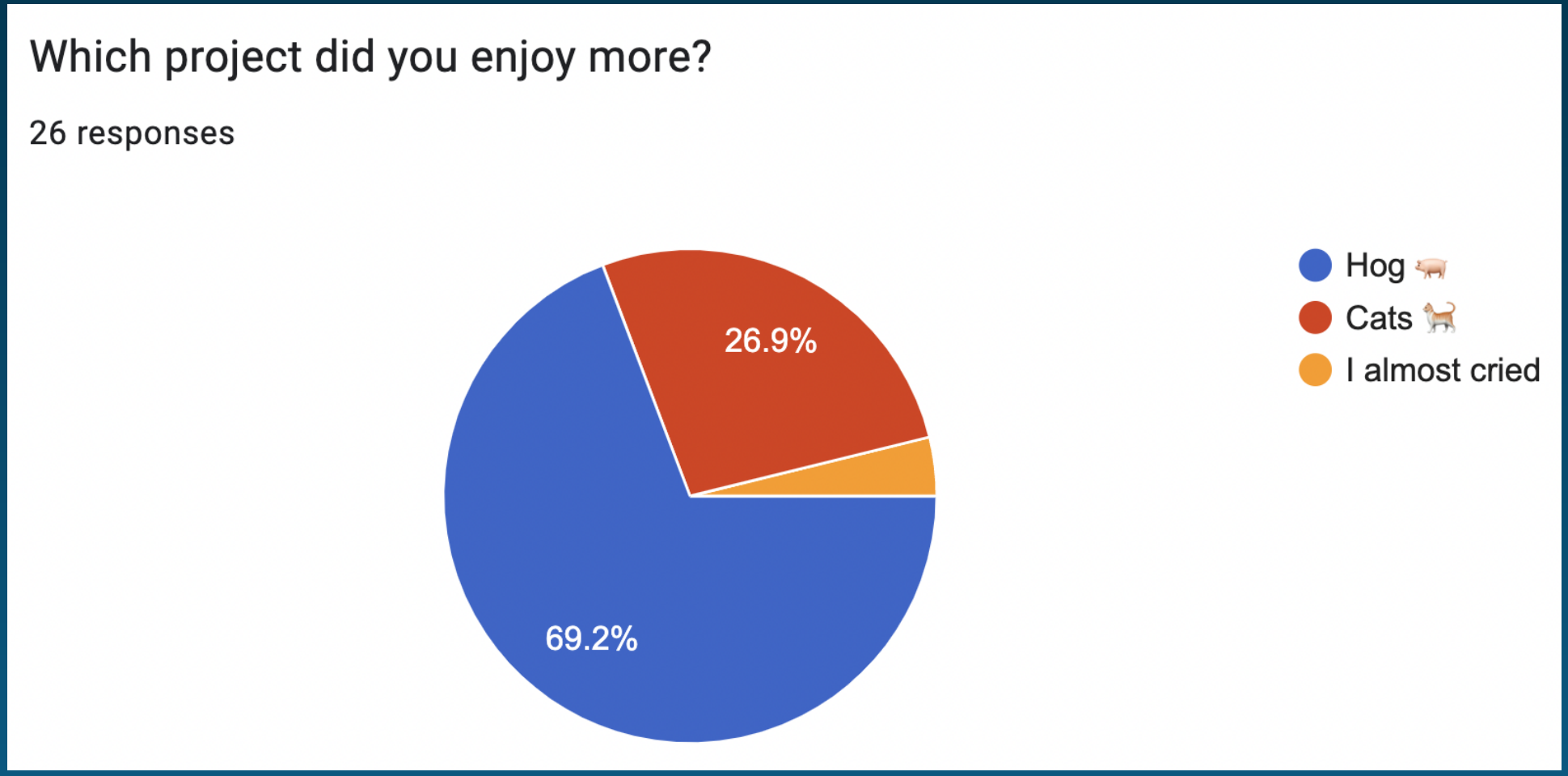
Mingxiao Wei
mingxiaowei@berkeley.edu

Oct 06, 2022

LOGISTICS 

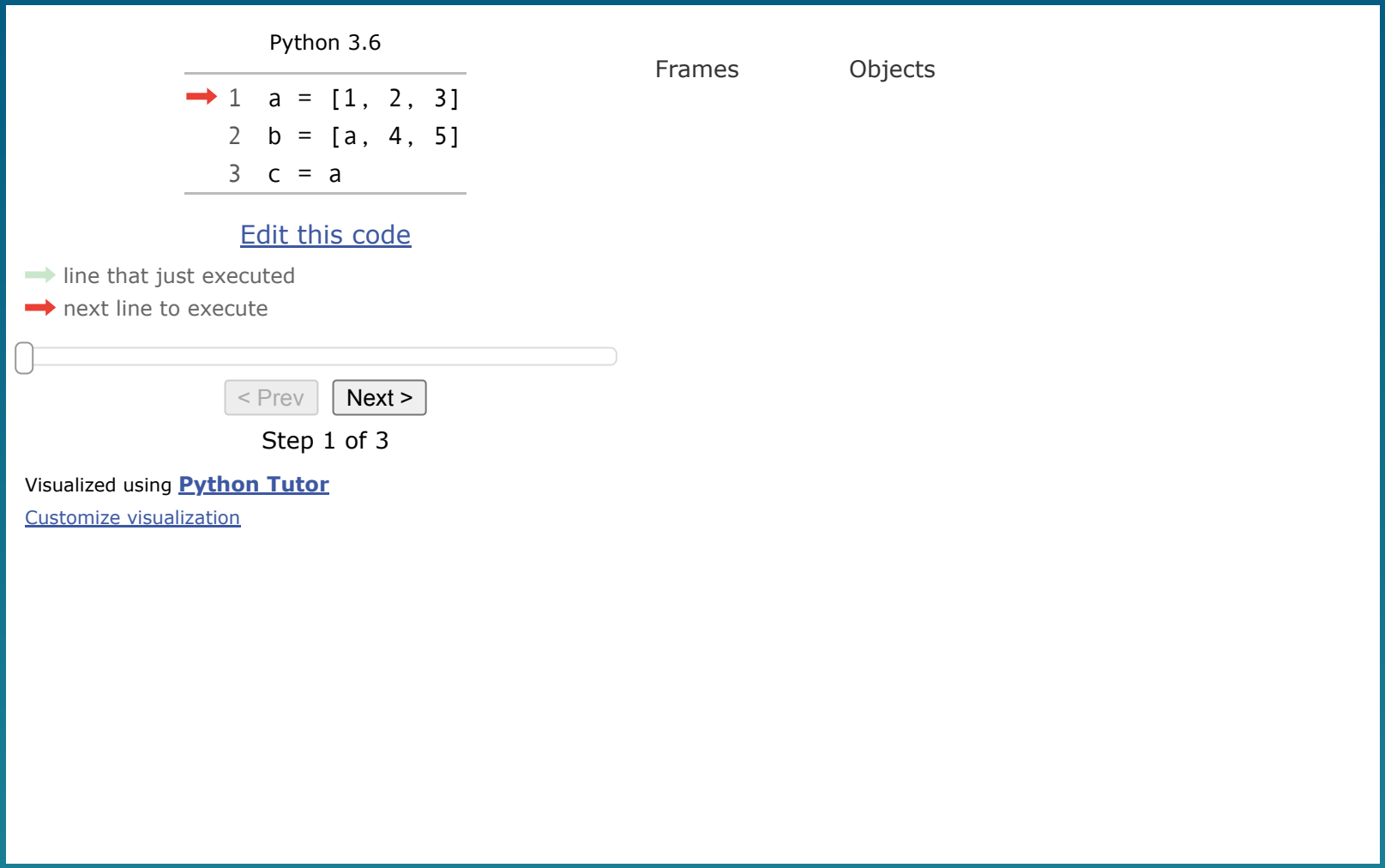
- Homework 04 due today 10/06
- Ants coming soon... 
  - If you want to stick to your old partner or find a new partner, now it's probably a good time to reach out!

FROM LAST TIME... 👀



MUTABILITY 

BOX & POINTER DIAGRAMS



LIST MUTATION METHODS

To call a method on a list, use dot notation: `lst.method(arg)`

- `append(elem)`
  - Add `elem` to the end of the list
  - `elem` can be of any type
  - Append only one element at a time - if `elem` is another list, the resulting list is nested
  - Return `None`
- `extend(lst)`
  - extend the list by making a shallow copy of `lst` and concatenating it with the other list `lst`
  - `lst` must be another list
  - If we need to add one element, `elem` to a list, `lst`, `lst.append(elem)` has the same effect as `lst.extend([elem])`
  - Return `None`

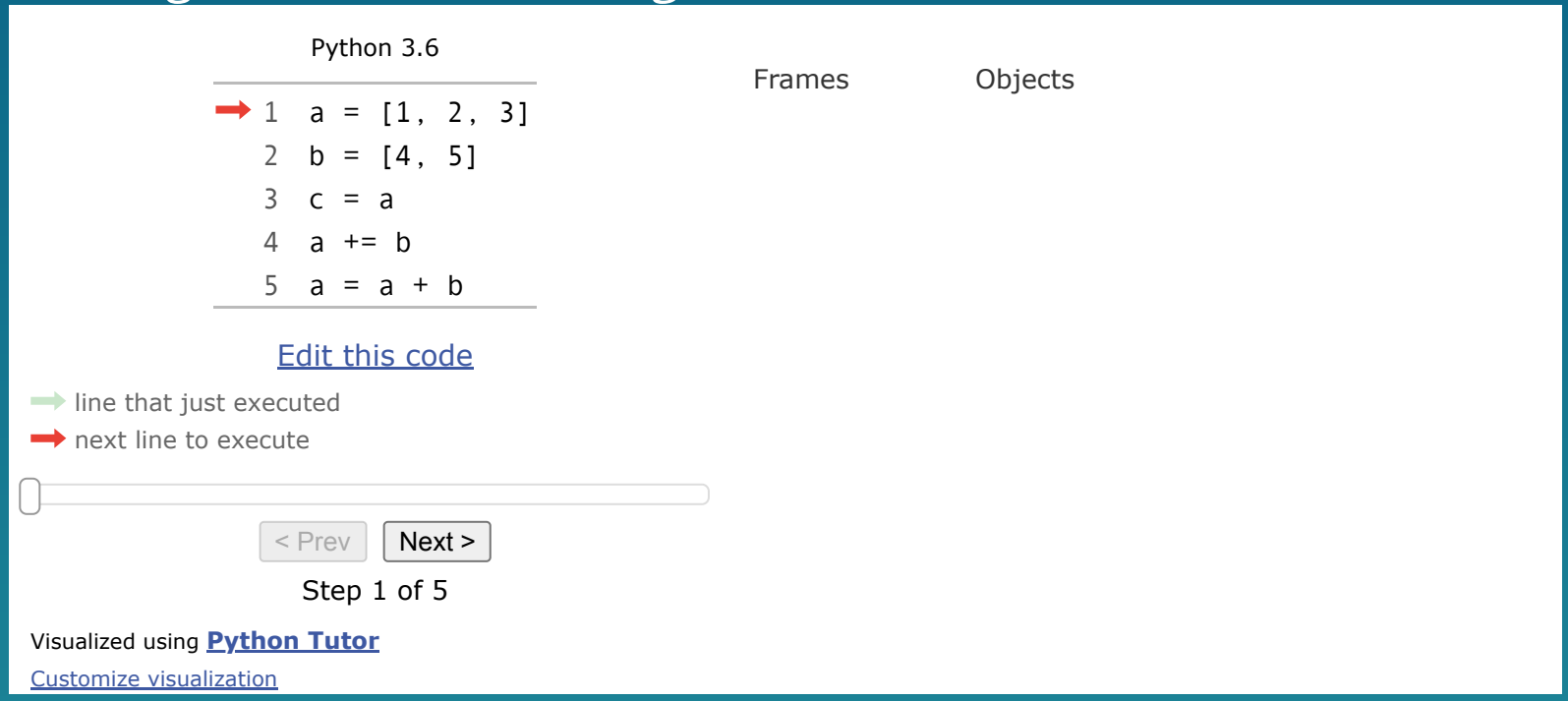
LIST MUTATION METHODS

To call a method on a list, use dot notation: `lst.method(arg)`

- `insert(i, elem)`
  - insert an element `elem` at index `i`
  - This does not replace any existing element - everything after the inserted element is "pushed back" by one element
  - `elem` can be of any type
  - Return `None`
- `remove(elem)`
  - remove the *first* occurence of `elem`
  - If `elem` is not in the list, it errors. Otherwise return `None`.
- `pop(i)`
  - remove and *return* the element at index `i`
  - `i` is optional - if not specified, defaulted to `len(lst)-1`

OTHER WAYS TO MUTATE A LIST

- `lst[i] = elem`
- `lst += another_lst`
  - This is equivalent to `lst.extend(another_lst)`, both of which *mutate* the original list `lst`
  - `lst = lst + another_lst` is different - this creates a *copy* of `lst`, concatenates it with `another_lst`, and returns the new list; the original `lst` is *unchanged*

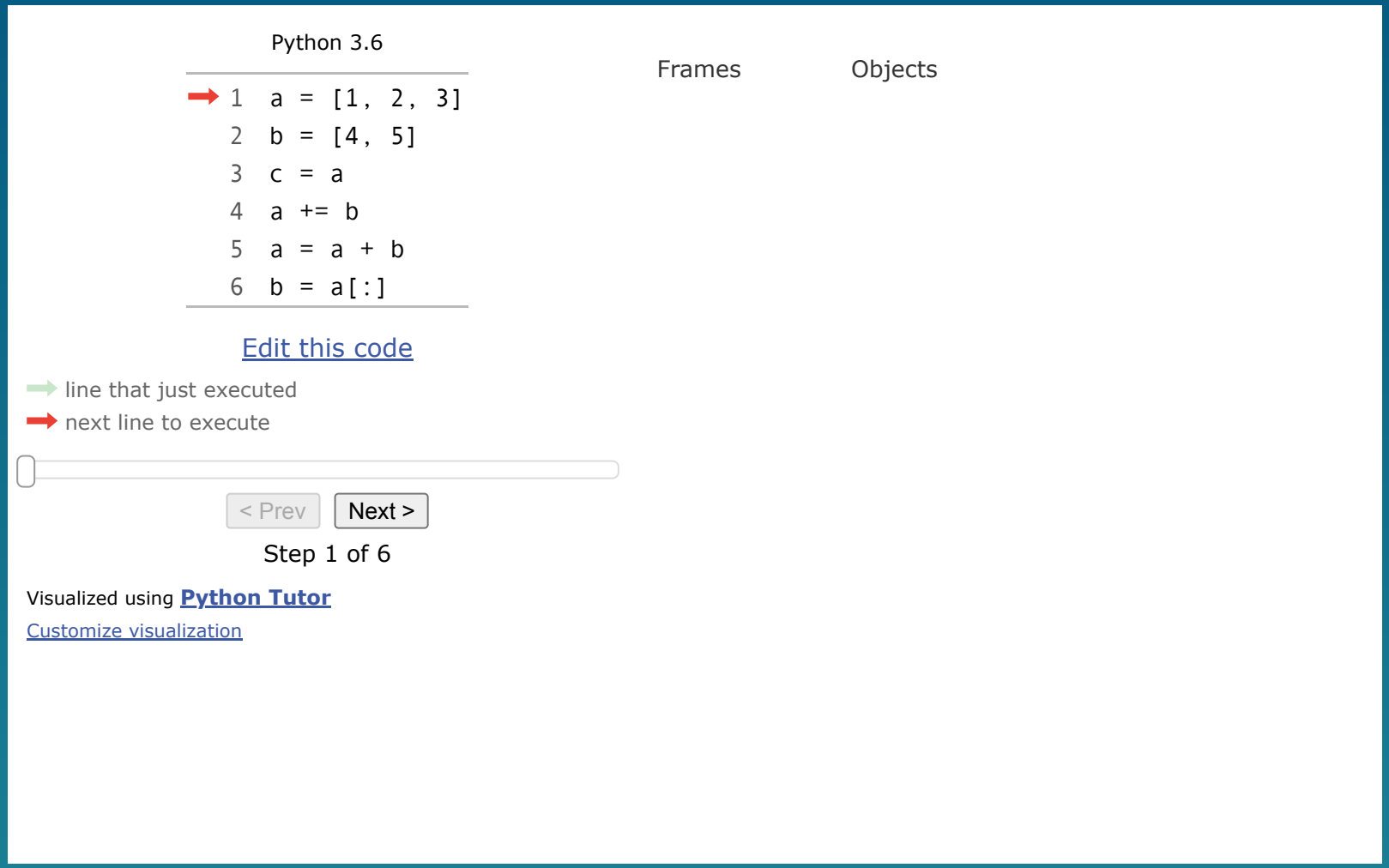


SHALLOW COPY OF LISTS

- Shallow Copy - copy of whatever is in the box
  - If it's a number, copy that number
  - If it's a pointer (e.g., to another list), copy the pointer, so that **the copied pointer points to the same object** in the ED
- When/How to make a shallow copy:
  - Use `+` to concatenate multiple lists together - make a shallow copy of each operand, concatenate them together, and return the new list *
  - `lst[:]` (or any kind of list slicing)
  - `list(lst)`
  - `lst.copy()`

* This is why for lists `a` and `b`, `a += b` is different from `a = a + b`

SHALLOW COPY OF LISTS



IDENTITY VS EQUALITY

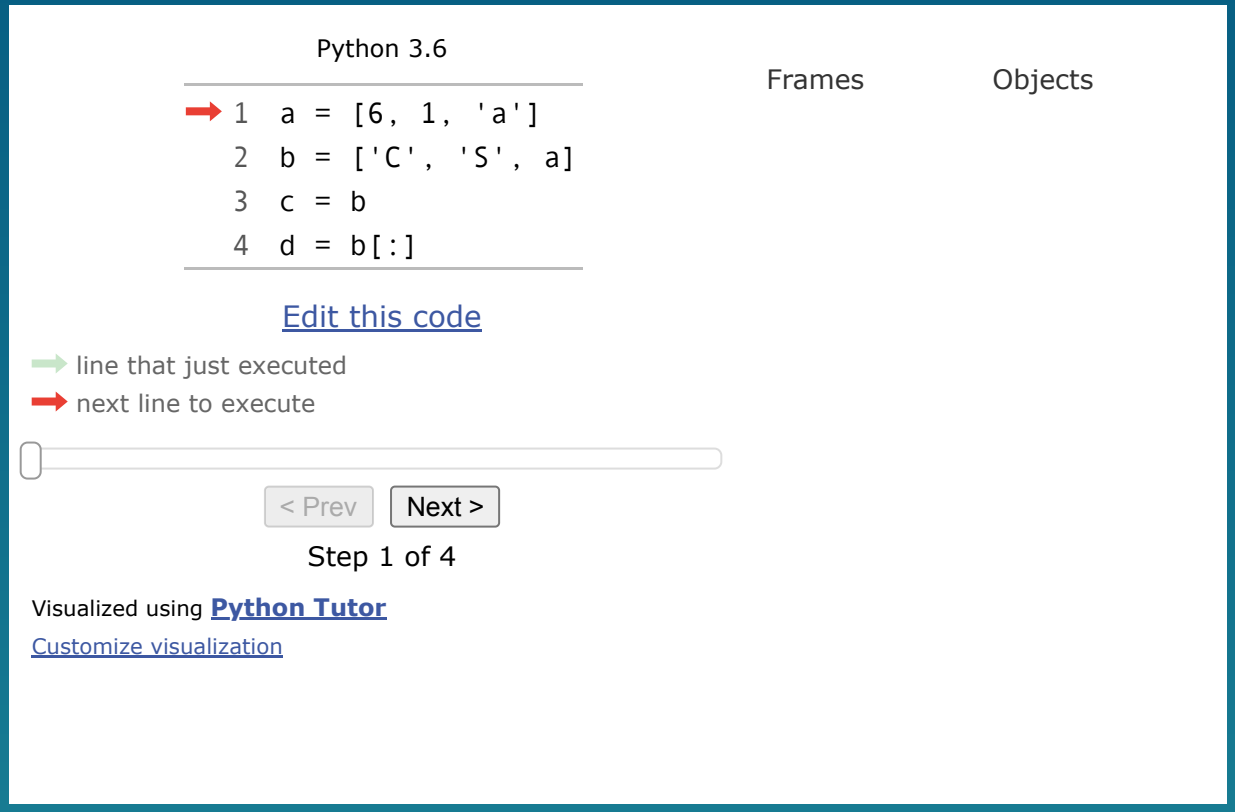
- Identity
  - Use `is` to check
  - Two variables have the same identity if either of the following holds:
    - They are primitive values (numbers, strings, boolean) and have the same value
    - They are non-primitive values (lists, tuples, etc.) and points to the same object in the ED
- Equality
  - Use `==` to check
  - Two variables are equal if they have the same "value"
    - For lists, this means should contain the same elements in the same order.

For primitive values, equality $\Leftrightarrow$ identity.

For non-primitive values, identity $\Rightarrow$ equality, but equality $\not\Rightarrow$ identity.

IDENTITY VS EQUALITY

```
>>> a = [6, 1, 'a']
>>> b = ['C', 'S', a]
>>> c = b
>>> b is c
True
>>> d = b[:]
>>> d is c
False
>>> d == c
True
```



MUTATION - PROBLEM SOLVING STRATEGIES

- Always keep in mind that all list mutation methods, except for `pop()`, return `None`
  - If you need to mutate a list and then return it, do the mutation before the return statement (i.e., be careful not to accidentally return `None`!)
- Do not mutate a list and iterate through it at the same time - this can lead to infinite loop!
  - Instead, iterate through a copy of the list and operate on the original list or iterate through the list using indices and modify the index when appropriate.
- Pay attention to whether the problem requires us to mutate the input list (in which case the return value is likely None), or return a new list

WORKSHEET Q1, 2

ITERATORS 

ITERABLES AND ITERATORS

- Iterables
  - An object that can be iterated through using a for loop:

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

  - Includes, but are not limited to, sequences (lists, tuples, dictionaries, etc.)
  - Calling `iter()` on an iterable returns a new iterator
- Iterators
  - An object that allows us to iterate through an iterable by keeping track of which element is the next in the sequence
  - Calling `next()` on an iterator gives the next element from the iterator - once an element is returned from calling `next()`, we can never go back to that element again by with the same iterator.
    If there's no more remaining element, raise an `StopIteration` error.
  - Calling `iter()` on an iterator returns the iterator *itself*, without resetting the position

ITERABLES AND ITERATORS

Note: since we can call `iter()` on an iterator, which will return the iterator itself, iterators are also iterables. But iterables are not necessarily iterators. You can use iterators wherever you can use iterables, but note that since iterators keep their state, they're only good to be iterated through once.

MORE ON FOR LOOPS

When we use a for loop to iterate over an iterable...

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

Here's what happens behind the scene...

```
iterator = iter(iterable)
# creates a new iterator from the iterable
try:
    while True:
        elem = next(iterator)
        # do something
except StopIteration:
# when there's no more element to iterate through
    pass  # do nothing and exit the loop
```

Using a for loop to iterate through an iterator essentially "uses up" the iterator!

ITERATORS - EXAMPLES

```
>>> lst = [1, 2]
>>> 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(iter(list_iter)) # Calling iter on an iterator returns itself
2
>>> another_iter = iter(list_iter)
>>> another_iter is list_iter
True
>>> next(list_iter)      # No elements left!
StopIteration
>>> lst                  # Original iterable is unaffected
[1, 2]
```

ITERABLE USES

- `map(f, iterable)`: Creates an **iterator** over `f(x)` for `x` in iterable
- `filter(f, iterable)`: Creates an **iterator** over `x` for `x` in iterable if `f(x)`
- `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`
- `zip(iterables*)`: Creates an **iterator** over co-indexed tuples with elements from each of the `iterables` (demo)
- `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**. (demo)

* Though technically iterators are iterables as well, here, by "iterables", we refer to the set of objects that are only iterables, but not iterators

ITERATORS - PROBLEM SOLVING STRATEGIES

- Situations where an iterator gets "used up" implicitly:
  - Using a for loop / list comprehension to iterate through the iterator
  - Calling the list constructor `list()` on an iterator
- Difference between calling `iter()` on an iterable vs. an iterator *:
  - on an iterable - returns a new iterator that starts from the beginning
  - on an iterator - returns the iterator itself, without resetting its position
- Try to avoid calling `next()` on an iterator more than once in a for/while loop - if you need to reuse the element, store it in some variables and use that variable instead!
- Pay attention to how you should iterate through the iterators (how many times you should call `next()` on it) as indicated by the problem statement

* Though technically iterators are iterables as well, here, by "iterables", we refer to the set of objects that are only iterables, but not iterators

WORKSHEET Q3

GENERATORS 

GENERATOR FUNCTIONS

- Generator functions
  - Have at least one `yield` statement
  - When called, return a generator function, which is a special type of iterator, without evaluating the function body
  - Allow us to define customized iterators

GENERATORS

- Generators
  - Returned by a generator function
  - When we call `next()` on a generator - "lazy evaluation"
    - If it's the first time, begin evaluating the body of the generator function until an element is yielded or the function terminates (by `return`, etc.)
    - Otherwise, pick up from where it stopped last time, and keep evaluating until another element is yielded or the function teriminates
  - Because of the lazy evaluation, we can define a generator that yield infinite elements without causing an infinite loop
  - Each time a generator function is called, a **new** generator object is returned
  - As with other iterators, if there are no more elements to be generated, then calling `next()` on the generator will raise a `StopIteration` error.

GENERATORS - EXAMPLES

```
def countdown(n):
    print("Beginning countdown!")
    while n >= 0:
        yield n
        n -= 1
    print("Blastoff!")
```

```
>>> c1, c2 = countdown(2), countdown(2)
>>> c1 is iter(c1)  # a generator is an iterator
True
>>> c1 is c2
False
>>> next(c1)
Beginning countdown!
2
>>> next(c2)
Beginning countdown!
2
```

YIELD VS YIELD FROM

- Both `yield` and `yield from` defines elements in the generator
- We can only `yield` one element at a time
- We can `yield from` any iterables (sequences, iterators, generators, etc.)

```
for elem in iterable:
    yield elem
# is equivalent to
yield from iterable
```

- With for loop and `yield`, we can modify the element before yielding them
- With `yield from`, we can only yield the exact elements from the iterable, without modifying them

GENERATORS - MORE EXAMPLES

```
>>> def gen_list (lst):
...     yield from lst
...
>>> g = gen_list([1, 2])
>>> next (g)
1
>>> next (g)
2
>>> next (g)
StopIteration
```

```
>>> def gen_list_1 (lst):
...     for elem in lst:
...         yield elem + 1
...
>>> g = gen_list([1, 2])
>>> next (g)
2
>>> next (g)
3
>>> next (g)
StopIteration
```

WORKSHEET Q4-7

ATTENDANCE! 🤠

go.cs61a.org/mingxiao-att

- The attendance form and slides are both linked on our section website!
- Once again, please do remember to fill out the form by midnight today!!