LAB 06

Iterators, Mutability

Mingxiao Wei
mingxiaowei@berkeley.edu

Oct 03, 2022

LOGISTICS 

- Lab 06 due this Wed 10/05
- Homework 04 due this Thu 10/06
- Reminder - if you want more review on discussion worksheet/exam prep, sign up for tutoring sections! (Ed post #1528)
- Reminder - homework 03 recovery (Ed post #1370)

MUTABILITY 

MUTABILITY - INTRO

- Mutable: contents or state of an object can be changed after the object is created
  - Eg: lists, dictionaries
- Immutable: contents or state of an object CANNOT be changed after the object is created
  - Eg: numeric types, tuples, strings
  - Note: even though we can *reassign a different value to a variable of numeric values*, we cannot change the *value of a number* (1 is always 1)

```
>>> a = (1, 2, 3)
>>> a[1]
2
>>> a[1] = 4
TypeError: 'tuple' object does not
support item assignment
```

```
>>> b = [1, 2, 3]
>>> b[1] = 4
>>> b
[1, 4, 3]
```

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`

```
>>> lst = [1, 2, 3]
>>> lst.append(4)
>>> lst
[1, 2, 3, 4]
>>> print(lst.append([5, 6]))
None
>>> lst
[1, 2, 3, 4, [5, 6]]
```

LIST MUTATION METHODS

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

- `extend(lst)`
  - extend the list by concatenating it with `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`

```
>>> lst = [1, 2, 3]
>>> lst.append(4)
>>> lst
[1, 2, 3, 4]
>>> lst.extend([5, 6])
>>> lst
[1, 2, 3, 4, 5, 6]
```

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`

```
>>> lst = [1, 2, 3]
>>> lst.insert(0, 0)
>>> lst
[0, 1, 2, 3]
>>> lst.insert('hi', 2)
>>> lst
[0, 1, 'hi', 2, 3]
```

LIST MUTATION METHODS

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

- `remove(elem)`
  - remove the *first* occurence of `elem`
  - If `elem` is not in the list, it errors. Otherwise return `None`.

```
>>> lst = [1, 2, 3, 4, 1, 6]
>>> lst.remove(1)
>>> lst
[2, 3, 4, 1, 6]
>>> lst.remove(5)
ValueError: list.remove(x): x not in list
```

LIST MUTATION METHODS

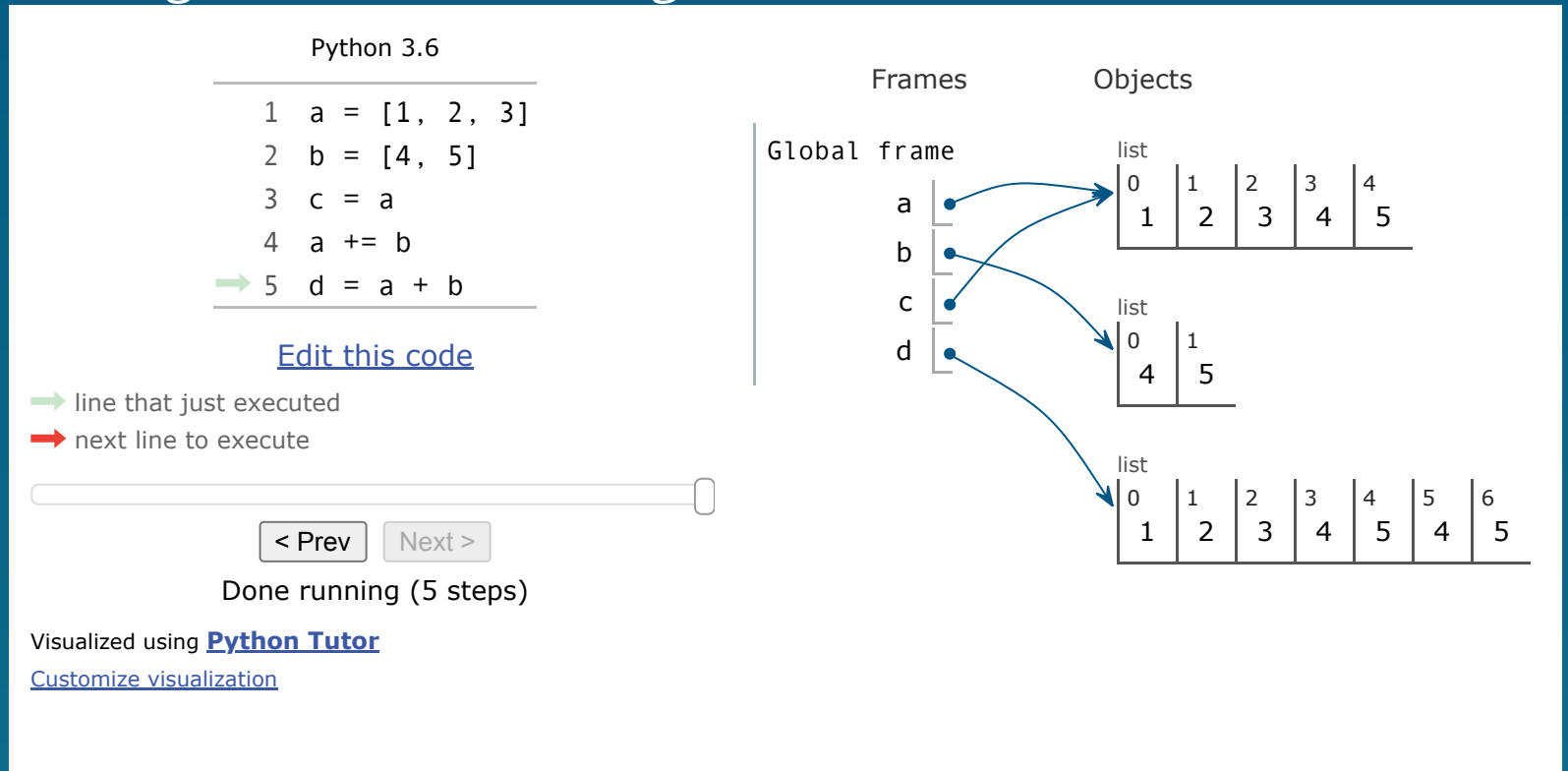
To call a method on a list, use dot notation: `lst.method(arg)`

- `pop(i)`
  - remove and *return* the element at index `i`
  - `i` is optional - if not specified, defaulted to `len(lst)-1`

```
>>> lst = [1, 2, 3, 4]
>>> lst.pop(1)
2
>>> lst
[1, 3, 4]
>>> lst.pop()
4
>>> lst
[1, 3]
```

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*



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.

ITERATORS 

ITERABLES AND ITERATORS - INTRO

- 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 - ANALOGIES

- Iterables
  - Analogous to a book - one can flip through the pages and go back and forth however they like
- Iterators
  - Analogous a one-way (forward-only) bookmark, which saves its current position and can locate the next page, but cannot go backwards
  - There can be multiple bookmarks on a book, all of which are independent of one another

Note: since you 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!

```
>>> 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

- `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

ATTENDANCE! 

go.cs61a.org/mingxiao-att

- The attendance form and slides are both linked on our section website!
- If you finish early, let me or any of the AI's know and we'll check you off
- Once again, please do remember to fill out the form by midnight today!!