Lists and Mutation

6.1000 LECTURE 6

FALL 2025

Announcements

- Pset 1 checkoff available through Wednesday 9/24
- Pset 2 out, due next Monday 9/29
  - redownload for updated test.py as of Saturday 9/20 at 3 pm
- Midterm 1 in two weeks 10/6
  - covers lectures 1–9, psets 1–2
  - study lecture code, finger exercises, psets, checkoffs
  - lectures 7–9 will be tested less heavily
- Pset 3 to be released Wednesday 10/8 after midterm
  - uses lectures 8–10 material

List mutation operations

- Review Python’s documentation
  - **non-mutating sequence ops**
    - applies to **`list`**, **`str`**, **`range`**
    - https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range
  - **mutating sequence ops**
    - https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types
  - **list-specific mutating ops**
    - https://docs.python.org/3/library/stdtypes.html#lists

List mutation operations

- Index assignment
  - *list*`[idx] = val`

- Grow or shrink by element
  - *list*`.append(val)`
  - *list*`.extend(vals),` *list* `+= vals`
  - *list*`.remove(val)`

- Grow or shrink by index
  - *list*`.insert(idx, val)`
  - *list*`.pop(idx), del` *list*`[idx]`

List mutation operations

- Sort and reverse
  - `list.sort() vs sorted(list)`
  - `list.reverse() vs reversed(list)`
- Clearing
  - `list.clear()`

Why mutation?

- Lists can get arbitrarily long
  - to change a small amount of content, would be wasteful to create an entirely new list
- So why aren't **`str`**s mutable? They can get quite long as well.
  - language design tradeoff: immutable objects have advantages
  - will discuss more next lecture

Meaning of "dot" notation

- E.g., ***list*.append(val)** or ***str*.index(char)**
- These are actual functions, but they work only on sequences
  - as if you were calling something like:
    - **append(*list*, val)**
    - **index(*str*, char)**
- Doesn't make sense to call ***int*.append(val)**
- Mechanism will become more clear by Lecture 14
  - classes and custom types

Naming suggestions

- Don’t name your lists **`list`**
- Avoid naming them a single character **`L`**
- Call them what they represent
  - **`seq`**, **`sequence`**, **`numbers`**, **`names`**, **`x_vals`**
- Conventions is lowercase with underscores
- Start reading PEP 8
  - https://peps.python.org/pep-0008/
  - https://pep8.org/

Aliasing vs copying

- Aliasing is when there are two or more references to the same object

  -
    ```
    your_list = [
        ["peanut butter", "jelly"], "toast“
    ]
    ```
  - `my_list = your_list`

- Copying is when an object’s contents are duplicated in a separate but equivalent object
  - `my_list = your_list.copy()`
  - equivalent forms:
    - `my_list = list(your_list)`
    - `my_list = your_list[:]`
  - note that `my_list[0]` and `your_list[0]` are still aliases
    - refer to the same list `["peanut butter", "jelly"]`

Shallow vs deep copying

- From previous slide

  -
    ```
    your_list = [
          ["peanut butter", "jelly"], "toast"
    ]
    ```
- Default copies only work on single object's contents, not objects referenced by those contents
  - `my_list = your_list.copy()`
- Deep copying traverses any nested compound structure to arbitrary depth
  - `import copy`
  - `my_list = copy.deepcopy(your_list)`
  - now `my_list[0]` and `your_list[0]` are no longer aliases
- Deep copying is rarely truly needed
  - `copy.deepcopy()` is fairly complex, needs to work for many types to arbitraty nesting depth
  - most applications don't involve arbitrary depth

Aliasing in functions

- Aliasing happens all the time
  - e.g., function parameters are aliases of references in the calling frame
  - inconsequential for immutable objects
  - can be useful and/or dangerous
    - **useful:** saves memory, different names in different contexts
    - **dangerous:** code in another context may not be aware contents of list or nested lists are changing
- Good practice
  - don’t mutate objects accessible from arguments unless docstring/spec says to
  - keep function parameters assigned to original inputs

Mutating examples

- Study code
- Often more than one way to apply mutating operations to achieve final result
    - Try coming up with alternate solutions, or explaining why they wouldn’t work
- Be careful when mutating what you’re looping over
    - indices can shift
    - end of the list can shift
    - consider iterating over a separate sequence that’s not being mutated

Takeaways

- Lists are mutable sequences of object references
  - no objects “within” the list object
- Aliasing happens everywhere
- Know when you need a copy instead
- Understand how mutation interacts with `for` loop mechanism