Lists, Mutation, Function scope

6.100 LECTURE 3

SPRING 2026

Announcements

- Pset 1 due this Friday 2/13
  - “warm-up” pset, not that long
  - use extra time to practice Python features on your own
- Showed f-strings during recitation last Friday
  - will use them extensively, but won’t be tested
  - reference https://fstring.help/ for syntax
- Expect you to have Python reliably installed by now
  - showed REPL during recitation
  - we gave instructions on setting up VS Code
  - but feel free to use any editor

List Operations

`list` objects

- Lists are just **sequences of references to objects**
  - each reference box is like a variable, but without the name
  - instead, retrieve or assign by **`some_list[index]`**
- `list`s follow many `str`-like operations
  - indexing, slicing
  - concatenation, repetition
  - iteration, **`len()`**
  - comparisons `==` `<`
    - plus implications for `<=` `>=` `>`

`list` membership

- Unlike **`str`**s, **`list`** membership test **`in`** is by element, not by substr/subsequence
  - relies on **`==`** comparison
  - **`min()`** and **`max()`** also accept lists, rely on **`<`** comparison
- Don't confuse **`elt in iterable`** with **`for x in iterable:`**
- **`not elt in iterable`** can be written as **`elt not in iterable`**
  - no parens needed because **`not`** has lower priority than **`in`** and **`not in`** is its own operator
  - https://docs.python.org/3/reference/expressions.html#operator-precedence
    - docs.python.org > Language Reference > Expressions > Operator precedence

`list` mutation

- Update: index assignment
- Grow: **`.append()`, `.extend()`**
- Shrink: **`del`** operator
- Common pattern to build a list
  - ```
    sequence = []
    for x in other_list:
        sequence.append(something with x)
    ```
- Alternate method
  - ```
    sequence = [0] * limit
    for i in range(limit):
        sequence[i] = something with other_list[i]
    ```
- Additional operations
  - https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range
    - docs.python.org > Library Reference > Built-in Types > Sequence Types

Essential list operations that use indices

- Assignment at index
  - `some_list[index] = value`
- Grow/shrink at end
  - `some_list.append(value)`
  - `some_list.extend(other_list)`
    - equivalently: `some_list += other_list`
  - `some_list.pop()`
- Grow/shrink in the middle
  - `some_list.insert(index, value)`
  - `some_list.pop(index)`
- Note syntax: *`object.operation(arg, arg, ...)`*
  - similar syntax for str operations
  - https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str
    - docs.python.org > Library Reference > Built-in Types > Text Sequence Type
  - look and behave like functions, but specialized on provided object
  - technically called methods, will revisit near end of term

Additional list operations for convenience

- Looking up by value
  - `some_list.count(value)`
  - `some_list.index(value)`
    - compare to `some_str.index(substr)`
    - compare to `some_str.find(substr)`
  - `some_list.remove(value)`
- Mutating entire list
  - `some_list.clear()`
  - `some_list.reverse()`
  - `some_list.sort()`
- Copying lists
  - `some_list.copy()` ← most explicit
  - `list(some_list)` ← most versatile, works on any iterable
  - `some_list[:]`

`str` operations involving `lists`

- `some_str.split(separator)`
  - list of substrs surrounding separator occurences
- `some_str.split()`
  - list of substrs surrounding consecutive spaces
- `separator.join([str1, str2, ...])`
  - str with separator interpolated between str elts in list
- https://docs.python.org/3/library/stdtypes.html#text-and-binary-sequence-type-methods-summary
  - docs.python.org > Library Reference > Built-in Types > Text ... Sequence Type Methods Summary

Function Environments

Last time: functions are contained programs

- **Defining a function**
  - accept input through **parameter** variables
  - produce output through a `return` statement
  - body code is indented
    - hence need `pass` if empty

- **Calling a function**
  - **syntax: function name** followed by **( )**
  - **argument** objects go inside parentheses
  - function body runs with parameters bound to arguments
  - **function call evaluates to object returned by body code**

Funtion definition mechanics

- Function definition is straightforward
  - **`def func(param1, param2, ...):`**
    `statement`
    `statement`
    - equivalent effect as *`variable = expression`*
  - create function object in heap memory
    - labeled with **`function`** type
    - stores parameter names in header
    - stores body code in body
    - create **`func`** variable on stack, pointing to object

Funtion call mechanics

- Function call is more involved
  - **`func(arg1, arg2, ...)`**
  - need to run a small program
  - we've seen now that programs are all about:
    - creating objects (and manipulating them with lists)
    - managing variable references to them
  - so need a safe "sandbox" to manage variable names
    - equivalent terminology: **environment, frame, scope**

Funtion call mechanics: overview

1. Retrieve function object
2. Evaluate arguments in order
3. Set up frame for function call
4. Assign parameter names in frame
5. Run body wrt frame until **`return`**
6. Substitute the returned object for the function call

Funtion call mechanics: global frame

- Need to separate **function code's environment** from **where "top-level code" runs**
  - Also need to separate from execution of other functions' code
- **Top-level code runs in global frame**
- All code outside functions we've seen has been in global frame
  - e.g., **all functions are defined in global**

Funtion call mechanics: running body code

- Follow usual rules of execution plus some extra rules
- When encounter a **`return`** statement:
  - evaluate expression into an object
  - tell Python here's what this call evaluates to
  - stop executing body code, remove the frame
  - resume execution in previous frame with object substituted
- If looking up a variable that doesn't exist in current frame, look in **global frame**
  - that's most likely what programmer intended
- If encounter a **function call,** apply the same rules
  - pause execution in current frame
  - evaluate function call, set up new frame on stack, get an object back
  - substitute in object and proceed

Mutation and functions

- When a function is called with a list variable
  - calling frame has a variable pointing to the list
  - function's frame has a parameter/variable pointing to same list
  - function body has potential to mutate that list
  - after function returns, calling frame sees mutated list
- Mutation doesn't have to be bad, can be part of design
  - but need to be clear about expected behavior
- Generally not encouraged to use global variables
  - examples in code only do so to reduce number of frames
  - often, global-level code can be put inside functions

Functions returning None

- Functions with no explicit return actually return **None**
  - a **NoneType** object
  - singleton object: only one instance ever exists in memory
  - comparison with **`is`** or **`is not`**
    - examines object identity
    - in contrast, **==** compares object value
- Typically, mutating operations return None
  - `some_list.append()`
  - `some_list.extend()`
  - `some_list.pop()` → `value`
  - `some_list.insert(index, value)`
  - `some_list.remove(value)`
  - `some_list.clear()`
  - `some_list.reverse()` vs `reversed()` vs `some_list[::-1]`
  - `some_list.sort()` vs `sorted()`
- Be careful about "returning" these calls, often not your intention

Next time

- **Pre-lecture code** will always be released the **day before lecture around noon**
- **One exception: next Tuesday 2/17 classes run on a Monday schedule**
  - to space things out, pre-lecture code will still be released Sunday 2/15 around noon
- **This Wednesday's pre-lecture code:** continuation of function features
  - docstrings
  - keyword arguments
  - default argument values