Sequences and Containers Meta

Computer Science Mentors 61A

September 26–September 30, 2022

Recommended Timeline

- Lists Minilecture: 10 minutes
- Lists WWPD: 5 minutes
- Lists Environment Diagram: 15 minutes
- Comprehensions: 12 minutes
- Duplicate List: 7 minutes
- Dictionaries Minilecture/example: 5 minutes/0 minutes
- Snapshot: 4 minutes
- Digraph: 10 minutes

As a reminder, these times do not add up to 50 minutes because no one is expected to get through all questions in a section. This is especially true this week, because this worksheet is rather long. You should use the worksheet as a problem bank around which you can structure your section to best accommodate the needs of your students. Both before and during section, consider which questions would be most instructive and how you should budget your time.

This week, we're providing slides! Check them out in the content team folder. Feel free to use these while you mini-lecture, and make a copy and modify them if you'd like.

Teaching sequences can be tricky because there's a lot of material to cover, but it tends to be pretty boring "this is how this works"-type content instead of deeper conceptual, information. So please, for the love of God, do not do a detailed mini-lecture on every aspect of sequences. This will take up far too much time and will probably not be a valuable experience for your students. Before mini-lecturing, it's valuable to ask your students what they would like you to go over specifically so that you're not repeating a bunch of information they already know.

Vibha Tantry and Jordan Schwartz, with
Gabe Classon, Aditya Balasubramanian, Oliver Yu, Hanze Tan, Gene Pan, Carolyn Wang, Raman Varma, Noor Haq, Angela Zhang, Abigail Brooks, Ted Kim, Aiden Joshua Vehemente, Marie Chorpita, Ashley Chiu, Bill Hu, Omar Yu, Angel Alberto Aldaco, Andrew Park, Chase Clements, Grace Yi, Preshtha Garg, Laksith Venkatesh Prabu, Aditya Murali, Irene Yang, Andy Chen, and Hailey Park

1 Sequences

Sequences are ordered data structures that have length and support element selection. Here are some common types of sequences you'll be dealing with in this class:

- Lists: `[1, [2], 'a', lambda x: 5]`
- Tuples: `(1, (2,), 'a', lambda x: 5)`
- Strings: `'Hello World!'`

While each type of sequence is different, they all share a common interface for manipulating and accessing their data:

- **Item selection**: Use square brackets to select an element at an index:

  `(3, 1, 2)[0]` → `"3"`, `"Hello"[-1]` → `"o"`

- **Length**: The built-in **`len`** function returns the length of a sequence:

  `len``((1,2))` → `2`

- **Concatenation**: Sequences can be concatenated with the + operator, which returns a *new* sequence:

  `[1, 2] + [3,4]` → `[1,2,3,4]`

- **Membership**: The **`in`** operator tests for sequence membership:

  `1` **`in`** `(1, 2, 3)` → `True`, `5` **`not in`** `(1, 2, 3)` → `True`, `"apple"` **`in`** `"snapple"` → `True`

- **Looping**: Sequences can be looped through with **`for`** loops:

```
>>> for x in [1, 2, 3]:
...     print(x)
1
2
3
```

- **Aggregation**: Common built-in functions—including **`sum`**, **`min`**, and **`max`**—can take sequences and aggregate them into a single value:

  `max``((3, 4, 5))` → `5`

- **Slicing**: Slicing is a way to create a copy of all or part of a sequence. The general syntax for slicing a sequence `seq` is as follows:

  `seq[<start index>:<end index>:<step size>]`

This evaluates to a new sequence that includes every element starting at `<start index>` and up to and *excluding* `<end index>` in `seq`, taking steps of size `<step size>`.

If we do not supply `<start index>` or `<end index>`, it will start at the beginning of the sequence and include every element up to and including the end of the sequence.

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[2:]
[3, 4, 5]
>>> lst[:3]
[1, 2, 3]
>>> lst[::-1]
[5, 4, 3, 2, 1]
>>> lst[1::2]
[2, 4]
```

List comprehensions, which only apply to lists, are a concise and powerful method to create a new list from another sequence. The syntax for a list comprehension is

```
[<expression> for <element> in <sequence> if <condition>]
```

We could equivalently write the following:

```
lst = []
for <element> in <sequence>:
    if <condition>:
        lst = lst + [<expression>]
```

The **`if`** `<condition>` filter statement is optional. The following list comprehension doubles each odd element of [1, 2, 3, 4]:

```
>>> [i * 2 for i in [1, 2, 3, 4] if i % 2 != 0]
[2, 6]
```

Equivalent in **`for`** loop syntax:

```
lst = []
for i in [1, 2, 3, 4]:
    if i % 2 != 0:
        lst = lst + [i * 2]
```

A lot of information in this guide is not the full and complete picture of how Python works, but students don't need to know that. Often a little misdirection is necessary to

improve initial knowledge acquisition before a more complete picture is painted later on. I would just be sure to mention the nature of sequences: a loose umbrella (not a strict definition like a class) that encompasses many different data types.

If students are confused consider bringing up specific examples of sequences. Reading rules on a page is often not actually that instructive.

1. What would Python display? Draw box-and-pointer diagrams for the following:

```
>>> a = [1, 2, 3]
>>> a
```

```
[1, 2, 3]
```

```
>>> a[2]
```

```
3
```

```
>>> a[-1]
```

```
3
```

```
>>> b = a
>>> a = a + [4, [5, 6]]
>>> a
```

```
[1, 2, 3, 4, [5, 6]]
```

```
>>> b
```

```
[1, 2, 3]
```

```
>>> c = a
>>> a = [4, 5]
>>> a
```

```
[4, 5]
```

```
>>> c
```

```
[1, 2, 3, 4, [5, 6]]
```

```
>>> d = c[3:5]
>>> c[3] = 9
>>> d

[4, [5, 6]]

>>> c[4][0] = 7
>>> d

[4, [7, 6]]

>>> c[4] = 10
>>> d

[4, [7, 6]]

>>> c

[1, 2, 3, 9, 10]
```

Teaching Tips

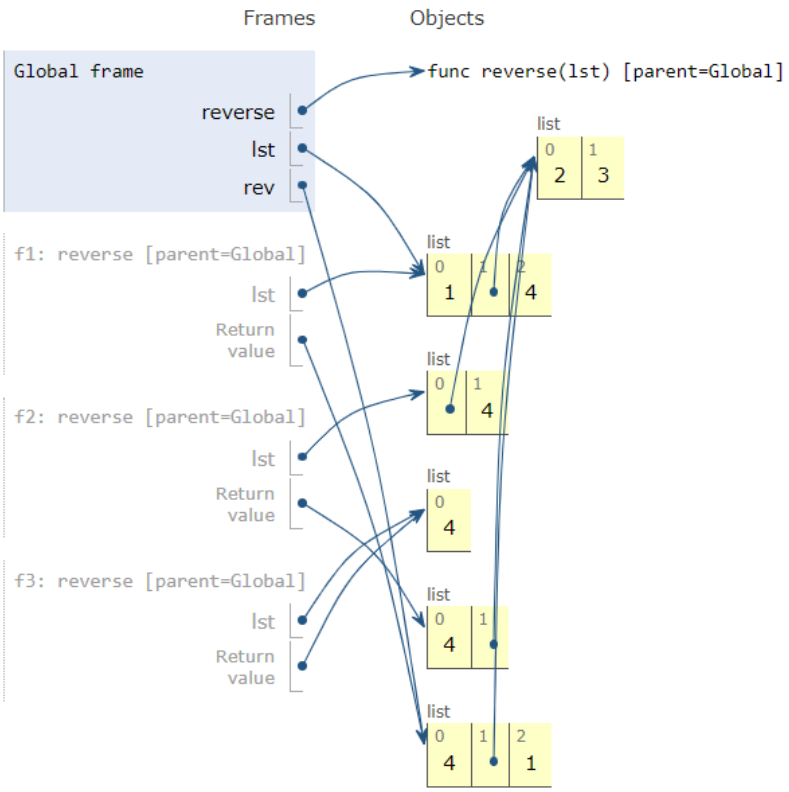
1. Refer to above notes on box and pointer diagrams! When going through this one, draw the box and pointer diagrams on the board
2. Encourage students to draw box and pointer diagrams if they seem stuck
3. It can be helpful to visually go through indexing and list slicing in the box and pointer diagram
4. Python is confusing when adding on to a list with `lst += [element]` and `lst = lst + element`. The former mutates and the latter creates a new list
5. Make sure you clearly state when you're making a new list object (i.e. at `a = a + [4, [5, 6]]` and list slicing like `d = c[3:5]`)
6. Be sure to touch on negative indices and reiterate the difference between shallow and deep copies

An alternative to doing a super detailed mini-lecture is going over these problems with your students and "learning by doing."

2. Draw the environment diagram that results from running the code below.

```
def reverse(lst):
    if len(lst) <= 1:
        return lst
    return reverse(lst[1:]) + [lst[0]]

lst = [1, [2, 3], 4]
rev = reverse(lst)
```



https://goo.gl/6vPeX9

Teaching Tips

- We call `reverse` recursively 3 times and open 3 new frames, each time passing through a shallow copy of the list without the first element
- We then return the list `[4]` after hitting the base case of a length 1 list.
- At each level, we take the list returned from the smaller recursive call and append `lst[0]` to the end of returned list.

- When you pass in a list as an argument to a function, the new frame's argument points to the same list passed in (it does not create a new list!)
- List slices create shallow copies (a new list is created but any pointers in the new list still point to the same thing)
- Keep in mind that the return value of the reverse function is a new list because of the `+ [lst[0]]`

This problem apparently is very tricky and takes a lot of time. So beware of this information while doing it with your students.

3. Write a list comprehension that accomplishes each of the following tasks.

   (a) Square all the elements of a given list, `lst`.

   ```
   [x ** 2 for x in lst]
   ```

   (b) Compute the dot product of two lists `lst1` and `lst2`. *Hint*: The dot product is defined as $\text{lst1}[0] \cdot \text{lst2}[0] + \text{lst1}[1] \cdot \text{lst2}[1] + \ldots + \text{lst1}[n] \cdot \text{lst2}[n]$. The Python **`zip`** function may be useful here.

   ```
   sum([x * y for x, y in zip(lst1, lst2)])
   ```

   (c) Return a list of lists such that `a = [[0], [0, 1], [0, 1, 2], [0, 1, 2, 3], [0, 1, 2, 3, 4]]`.

   ```
   a = [[x for x in range(y)] for y in range(1, 6)]
   ```

   (d) Return the same list as above, except now excluding every instance of the number 2: `b = [[0], [0, 1], [0, 1], [0, 1, 3], [0, 1, 3, 4]])`.

   ```
   b = [[x for x in range(y) if x != 2] for y in range(1, 6)]
   ```

Teaching Tips

1. It may be helpful to start with the basic list comprehension template of `[<expr> for <item> in <iterable> if <condition>]`
2. The list of list questions are tricky, so try nudging your students in the right direction by reminding them that it is completely possible to nest list comprehensions.
3. Make sure you also understand what the Python `zip` function does!

I think it's probably not necessary to go over all of these with your students, just do enough to where they're comfortable with things.

4. Write a function `duplicate_list`, which takes in a list of positive integers and returns a new list with each element `x` in the original list duplicated `x` times.

```
def duplicate_list(lst):
    """
    >>> duplicate_list([1, 2, 3])
    [1, 2, 2, 3, 3, 3]
    >>> duplicate_list([5])
    [5, 5, 5, 5, 5]
    """
    ____________________________________

    for ____________________________________:

        for ____________________________________:

            ____________________________________

    ____________________________________
```

```
new_list = []
for x in lst:
    for i in range(x):
        new_list = new_list + [x]
return new_list
```

Teaching Tips

1. If students have trouble arriving at the solution, walk through the intuition of nested for loops and discuss what each loop represents. For example, the first loop represents iterating over each element of the list and the second one represents repeating that element.
2. This is a good problem to emphasize how we can format our logic and approach to problems based on the skeleton code

2 Dictionaries

Dictionaries are another useful Python data structure that store a collection of items. However, instead of assigning each item a numerical index, each **value** in a dictionary is mapped to by some **key**.

Dictionaries are denoted with curly braces and use much of the syntax—including item selection with square brackets, membership testing with **`in`**, and length checking with **`len`**—is the same as that of sequences. Consider the following "Big" example:

```
>>> big_game_wins = {"Cal": 48, "Stanford": 65}
>>> big_game_wins
{"Cal": 48, "Stanford": 65}
>>> big_game_wins["Stanford"]
65
>>> big_game_wins["Cal"]
48
>>> big_game_wins["Cal"] += 1
>>> big_game_wins["Cal"]
49


>>> list(big_game_wins.keys())
["Cal", "Stanford"]
>>> list(big_game_wins.values())
[49, 65]

>>> "Cal" in big_game_wins
True
>>> "Tie" in big_game_wins
False
>>> 65 in big_game_wins
False


>>> big_game_wins["Tie"]
KeyError: Tie
>>> big_game_wins["Tie"] = 11
>>> big_game_wins["Tie"]
11
```

Here, I decided to not list out everything a dictionary can do but rather teach by giving a long example. I find that students tend to glaze over when they're asked to look at

something that long, so I really recommend walking through the whole process.

From a technical standpoint, dictionaries are ordered in Python. But your students don't need to know that.

1. Complete the function `snapshot`, which takes a single-argument function `f` and a list `inputs` and returns a "snapshot" of `f` on `inputs`. A "snapshot" is a dictionary where the keys are the provided `inputs` and the values are the corresponding outputs of `f` on each input.

```
def snapshot(f, inputs):
    """
    >>> snapshot(lambda x: x**2, [1, 2, 3])
    {1: 1, 2: 4, 3: 9}
    """

    snap = ____________________________________________________

    ____________________________________________________:

        ____________________________________________________

    return snap
```

```
def snapshot(f, inputs):
    snap = {}
    for input in inputs:
        snap[input] = f(input)
    return snap
```

One way to think of a dictionary is as a function (in the mathematical sense) with a finite domain: you provide it an input and it gives your some output. The idea behind this problem is to exercise that connection by having students convert between functions defined by rules that often have unlimited domains (e.g. $f(x) = x^2$) and finite functions that are defined by directly spelling out the outputs of a function. That's why this problem is called "snapshot"—it's a small snapshot of a function's behavior over a limited domain.

This problem apparently is rather easy, so you can probably skip over it if your students are strapped for time or if you think they probably don't need it.

2. A *digraph* is any pair of immediately adjacent letters; for example, "otto" contains three digraphs: "ot", "tt", and "to". Write a function `count_digraphs`, which takes a piece of `text` and a list of letters `alphabet` and analyzes the frequency of diagraphs in `text`. Specifically, `count_digraphs` returns a dictionary whose keys are the valid digraphs of `text` and whose values are the number of times each digraph occurred. (A digraph is valid if it is formed out of letters from the specified `alphabet`.)

```
def count_digraphs(text, alphabet):
    """
    >>> count_digraphs("otto", ['o', 't'])
    {'ot': 1, 'tt': 1, 'to': 1}
    >>> count_digraphs("otto", ['t'])
    {'tt': 1}
    >>> count_digraphs("6161 6", ['6', '1'])
    {'61': 2, '16': 1}
    """

    freq = {}

    _____________________________________________________________:

        if _____________________________________________________:

            digraph = _____________________________________

            ___________________________________________________

                ___________________________________________________

            ___________________________________________________

                ___________________________________________________

    return freq
```

```
def count_digraphs(text, alphabet):
    freq = {}
    for i in range(len(text) - 1):
        if text[i] in alphabet and text[i + 1] in alphabet:
            digraph = text[i] + text[i + 1]
```

```
            if digraph in freq:
                freq[digraph] += 1
            else:
                freq[digraph] = 1
    return freq
```

What's the point of this question? Well, analyzing the frequency of digraphs can be valuable in all sorts of situations, including cryptanalysis. But in a more general sense, using dictionaries to count things is very common. So hopefully this problem will give students some guidance on that front.

Students will probably need to look at the doc tests to fully understand the problem, including what we mean by 'valid' and whether spaces should count or not.

If your students are completely lost, walk through the doctests with them, and ask them how they would find digraphs by hand, and try and lead them to understand what two elements they should be checking in each iteration.

Students will likely find a significant amount of trouble in differentiating what to do when the digraph is present in the dictionary and when it is not. If they are stuck on this,, here are some leading questions you could ask:

- What happens when we encounter a digraph we haven't seen before?
- What happens when we try to add a digraph to the dictionary that's not already in there?
- What do we need to know is true before we can add onto the digraph that's already in the dictionary? How can we do that?