SEQUENCES, CONTAINERS, AND DATA ABSTRACTION Solutions

COMPUTER SCIENCE MENTORS 61A

February 20–February 24, 2022

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`

  Membership in Strings vs. Lists and Tuples: As a short aside, while the `in` operator works the same for lists and tuples, checking if an element is contained within the list/tuple container, the `in` operator instead for strings checks for direct substrings rather than the existence of distinct elements within the string.

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

Created by Alyssa Smith, Esther Shen, Maya Romero, Natalie Wei, Sandhya Ganesan, Kaelyn Huang, Cecilia Aiko, Aurelia Wang

```
>>> 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]
```

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]
```

2. What would Python display? Draw box-and-pointer diagrams to find out.

   (a)
   ```
   L = [1, 2, 3]
   B = L
   B
   ```

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

   (b)
   ```
   A = L[1:3]
   L[0] = A
   L = L + A
   B
   ```

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

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)]
   ```

4. Fill in the methods below according to the doctests.

```
def gen_list(n):
    """
    Returns a nested list structure of n elements where the
    ith element is a list from 0 (inclusive) to i (exclusive).
    >>> gen_list(3)
    [[0], [0, 1], [0, 1, 2]]
    >>> gen_list(5)
    [[0], [0, 1], [0, 1, 2], [0, 1, 2, 3], [0, 1, 2, 3, 4]]
    """
    return ________________________________________________
```

For an additional challenge, try out the following:

```
def gen_increasing(n):
    """
    Returns a nested list structure of n elements where the
    ith element of each list is one more than the previous
    element (even if the previous is in a prior sublist).
    >>> gen_increasing(3)
    [[0], [1, 2], [3, 4, 5]]
    >>> gen_increasing(5)
    [[0], [1, 2], [3, 4, 5], [6, 7, 8, 9], [10, 11, 12, 13,
    14]]
    """
    return ________________________________________________
```

Hint: You can sum ranges. E.g. `sum(range(3))` gives us $0 + 1 + 2 = 3$.

```
def gen_list(n):
    return [[i for i in range(j+1)] for j in range(n)]

def gen_increasing(n):
        return [[i for i in range(sum(range(j+1)), sum(range(j+1)) + j+1)]
            for j in range(n)]

\textit{An alternate solution for gen_increasing is:}
def gen_increasing(n):
    return [[i + sum(range(j + 1)) for i in range(j + 1)] for j in
        range(n)]
```

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 same syntax as sequences—including item selection with square brackets, membership testing with **in**, and length checking with **len**. 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
```

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

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

    snap = ________________________________________

    ________________________________________:

        ________________________________________

    return snap
```

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

2. Write a function `count_t`, which takes in a dictionary `d` and a string `word`. The function should count the instances of the letter "t" in `word` and add a key-value pair to the dictionary. The key will be `word` and the value will be the number of "t"s in `word`

```
def count_t(d, word):
    """
    >>> words = {}
    >>> count_t(words, "tatter")
    >>> words["tatter"]
    3
    >>> count_t(words, "tree")
    >>> words
    {'tatter': 3, 'tree': 1}
    """
    ______________________________

    for ___________________________:

        if ___________________________:

            _________________________________

    ______________________________
```

```
count = 0
for c in word:
    if c == 't':
        count += 1
d[word] = count
```

3. 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
```