Sequences and Containers

Computer Science Mentors 61A

October 3–October 007, 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`

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

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

```
>>> 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))` $\rightarrow$ 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
```

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

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

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

```
>>> b
```

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

```
>>> c
```

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


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


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


>>> c
```

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)
```

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

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

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

   (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]])`.

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 ____________________________:

            ____________________________________

    ________________________________
```

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
```

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
```

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 digraphs 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
```