Python Lists, Dictionaries, and Data Abstraction Solutions

CS 61A

July 7, 2021

1 Lists

Lists Introduction:

Lists are a type of sequence, which is to say they're ordered collections of values that have both a length and the ability to select elements.

```
>>> lst = [1, False, [2, 3], 4] # a list can contain anything
>>> len(lst)
4
>>> lst[0] # Indexing starts at 0
1
>>> lst[4] # Indexing ends at len(lst) - 1
Error: list index out of range
```

We can iterate over lists using their index, or iterate over elements directly

```
for index in range(len(lst)):
        # do things
for item in lst:
        # do things
```

List comprehensions are a useful way to iterate over lists when your desired result is a list.

```
new_list2 = [<expression> for <element> in <sequence> if <
   condition>]
```

We can use **list slicing** to create a copy of a certain portion or all of a list.

```
new_list = lst[<starting index>:<ending index>]
copy = lst[:]
```

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

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

>>> a[2]
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. Draw the environment diagram that results from running the code.

```
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

3. Write a function that takes in a list `nums` and returns a new list with only the primes from `nums`. Assume that `is_prime(n)` is defined. You may use a `while` loop, a `for` loop, or a list comprehension.

```
def all_primes(nums):

    result = []
    for i in nums:
        if is_prime(i):
            result = result + [i]
    return result

    List comprehension:
    return [x for x in nums if is_prime(x)]
```

4. 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)]
```

2 Dictionaries

Dictionaries are data structures that map keys to values. In Python, the key-value pairs in a dictionary are unordered.

1. Write a function `replace_all` that replaces all occurences of x as a value (not a key) in d with y.

```
def replace_all(d, x, y):
    """Replace all occurrences of x as a value (not a key) in
       d with y.
    >>> d = {3: '3', 'foo': 2, 'bar': 3, 'garply': 3, 'xyzzy':
        99}
    >>> replace_all(d, 3, 'poof')
    >>> d == {3: '3', 'foo': 2, 'bar': 'poof', 'garply': 'poof
       ', 'xyzzy': 99}
    True
    """

    for key in d:
        if d[key] == x:
            d[key] = y
```

2. Write a function `counter` that takes in an input string, `message`, and returns a dictionary that maps each unique word in `message` to the number of times it appears.

```
def counter(message):
    """ Returns a dictionary of each word in message mapped
    to the number of times it appears in the input string.
    >>> x = counter('to be or not to be')
    >>> x['to']
    2
    >>> x['be']
    2
    >>> x['not']
    1
    >>> y = counter('run forrest run')
    >>> y['run']
    2
    >>> y['forrest']
    1
    """
    word_list = message.split() # .split() returns a list of
       the words in the string. Try printing it!

    result_dict = {}
    for word in word_list:
        if word in result_dict:
            result_dict[word] += 1
        else:
            result_dict[word] = 1
    return result_dict
```

3 Abstraction

Data Abstraction Overview:

Abstraction allows us to create and access different types of data through a controlled, restricted programming interface, hiding implementation details and encouraging programmers to focus on how data is used, rather than how data is organized. The two fundamental components of a programming interface are a constructor and selectors.

1. Constructor: The interface that creates a piece of data; e.g. calling `c = car("Tesla")` creates a new car object and assigns it to the variable `c`.
2. Selectors: The interface by which we access attributes of a piece of data; e.g. calling `get_brand(c)` should return `"Tesla"`.

Through constructors and selectors, a data type can hide its implementation, and a programmer doesn't need to *know* its implementation to *use* it.

1. The following is an **Abstract Data Type (ADT)** for elephants. Each elephant keeps track of its name, age, and whether or not it can fly. Given our provided constructor, fill out the selectors:

```
def elephant(name, age, can_fly):
    """
    Takes in a string name, an int age, and a boolean can_fly.
    Constructs an elephant with these attributes.
    >>> dumbo = elephant("Dumbo", 10, True)
    >>> elephant_name(dumbo)
    "Dumbo"
    >>> elephant_age(dumbo)
    10
    >>> elephant_can_fly(dumbo)
    True
    """
    return [name, age, can_fly]
def elephant_name(e):

    return e[0]
def elephant_age(e):

    return e[1]
```

```
def elephant_can_fly(e):

    return e[2]
```

2. This function returns the correct result, but there's something wrong about its implementation. How do we fix it?

```
def elephant_roster(elephants):
    """
    Takes in a list of elephants and returns a list of their
       names.
    """
    return [elephant[0] for elephant in elephants]
```

`elephant[0]` is a Data Abstraction Violation (DAV). We should use a selector instead. The corrected function looks like:

```
def elephant_roster(elephants):
    return [elephant_name(elephant) for elephant in elephants]
```

3. Fill out the following constructor for the given selectors.

```
def elephant(name, age, can_fly):

    return [[name, age], can_fly]


def elephant_name(e):
    return e[0][0]


def elephant_age(e):
    return e[0][1]


def elephant_can_fly(e):
    return e[1]
```

4. How can we write the fixed `elephant_roster` function for the constructors and selectors in the previous question?

No change is necessary to fix `elephant_roster` since using the `elephant` selectors "protects" the roster from constructor definition changes.

5. **(Optional)** Fill out the following constructor for the given selectors.

```
def elephant(name, age, can_fly):
    """
    >>> alex = elephant("Alex Kassil", 22, False)
    >>> elephant_name(alex)
    "Alex Kassil"
    >>> elephant_age(alex)
    22
    >>> elephant_can_fly(alex)
    False
    >> alex("size")
    "Breaking abstraction barrier!"
    """
    def select(command):

        if command == "name":
            return name
        elif command == "age":
            return age
        elif command == "can_fly":
            return can_fly
        return "Breaking abstraction barrier!"


    return select

def elephant_name(e):
    return e("name")

def elephant_age(e):
    return e("age")

def elephant_can_fly(e):
    return e("can_fly")
```