CS 61A Data Abstraction & Sequences
Spring 2019 Discussion 4: February 20, 2019 Solutions

1 List Comprehensions

A **list comprehension** is a compact way to create a list whose elements are the results of applying a fixed expression to elements in another sequence.

```
[<map exp> for <name> in <iter exp> if <filter exp>]
```

```
new_lst = []
for <name> in <iter exp>:
    if <filter exp>:
        new_lst += [<map_exp>]
return new_lst
```

It might be helpful to note that we can rewrite a list comprehension as an equivalent for statement. See the example to the right.

Let's break down an example:

```
[x * x - 3 for x in [1, 2, 3, 4, 5] if x % 2 == 1]
```

In this list comprehension, we are creating a new list after performing a series of operations to our initial sequence `[1, 2, 3, 4, 5]`. We only keep the elements that satisfy the filter expression `x % 2 == 1` (1, 3, and 5). For each retained element, we apply the map expression `x*x - 3` before adding it to the new list that we are creating, resulting in the output `[-2, 6, 22]`.

Note: The `if` clause in a list comprehension is optional.

Questions

1.1 What would Python display?

```
>>> [i + 1 for i in [1, 2, 3, 4, 5] if i % 2 == 0]
```

```
[3, 5]
```

Video walkthrough

```
>>> [i * i - i for i in [5, -1, 3, -1, 3] if i > 2]
```

```
[20, 6, 6]
```

Video walkthrough

```
>>> [[y * 2 for y in [x, x + 1]] for x in [1, 2, 3, 4]]
```

```
[[2, 4], [4, 6], [6, 8], [8, 10]]
```

Video walkthrough

2 Data Abstraction

Data abstraction is a powerful concept in computer science that allows programmers to treat code as objects. That way, programmers don't have to worry about *how* code is implemented — they just have to know *what* it does.

Data abstraction mimics how we think about the world. For example, when you want to drive a car, you don't need to know how the engine was built or what kind of material the tires are made of. You just have to know how to turn the wheel and press the gas pedal.

Data abstraction is useful when dealing with compound values, or values that consist of more than one component. An example of such a value is a rational number, or a number than can be written as `x / y`, which consists of a numerator and a denominator.

We can represent these types of values as **abstract data types** (ADTs). An abstract data type consists of two types of functions:

- **Constructors**: functions that build the abstract data type.
- **Selectors**: functions that retrieve information from the data type.

Below are possible function signatures for the constructor and selectors of a rational number ADT:

```
rational(numerator, denominator)
numer(rational)
denom(rational)
```

Here is how we might use this constructor and these selectors:

```
>>> half = rational(1, 2)
>>> numer(half)
1
>>> denom(half)
2
```

The following function multiplies together two rational numbers, returning a new rational number.

```
def mul_rational(x, y):
    return rational(numer(x) * numer(y),
                    denom(x) * denom(y))
```

Questions

The 61A TAs have decided to call upon the power of data abstraction to organize their discussion sections. To do so, they've created a `discussion` abstract data type. A discussion contains three things:

- The name of the TA running the section
- The time the section starts, given as an integer
- A list of students enrolled in the section

Given this, the TAs come up with the following constructor and selectors:

- `make_discussion(ta, time, students)`: Creates and returns a new discussion section.
- `get_ta(disc)`: Returns the TA running the given discussion section.
- `get_time(disc)`: Returns the start time of the given discussion section.
- `get_students(disc)`: Returns the list of students enrolled in the given discussion section.

2.1 Implement `add_student`, which takes in a discussion section and a string representing a student's name, and returns a new discussion with the new student added to the roster. The list of students for the new discussion should be a new list. Remember to use the constructor and selectors!

```
def add_student(disc, student):
    """ Adds a student to this discussion.
    >>> disc = make_discussion("Chris", 4, ["Abhinav", "Brian"])
    >>> new_disc = add_student(disc, "Carl")
    >>> get_students(new_disc)
    ["Abhinav", "Brian", "Carl"]
    >>> get_students(disc)
    ["Abhinav", "Brian"]
    """


    new_students = get_students(disc) + [student]
    ta = get_ta(disc)
    time = get_time(disc)
    return make_discussion(ta, time, new_students)
```

3 Abstraction Violations

Notice how we did not need to know how the constructors and selectors in the previous section were implemented in order to use them. This is what we mean by the *implementation* and *use* of an abstract data type being separate. In fact, you should never assume anything about how the constructors and selectors for an abstract data type are implemented. Doing so is called a **data abstraction violation**.

As an example, here is one implemenation for the `rational` constructor.

```
def rational(n, d):
    return [n, d]
```

Given this constructor, the following would be considered a data abstraction violation:

```
>>> frac1 = rational(3, 4)
>>> frac2 = rational(5, 6)
>>> frac1[0] * frac2[0]
15
```

This is because we assumed rationals were represented as lists instead of accessing their elements using the selectors.

The TAs have decided to reveal the implementation of the discussion section ADT. Use these function definitions to answer the next two questions:

```
def make_discussion(ta, time, students):
    return [name, time, students]

def get_ta(disc):
    return disc[0]

def get_time(disc):
    return disc[1]

def get_students(disc):
    return disc[2]
```

3.1 The TAs have written the following code using the above data abstraction. However, it contains some abstraction violations. Underline each occurence of an abstraction violation. Then, if possible, write the correct line of code to the right.

```
def check_start(disc1, disc2):
    """Checks whether disc1 and disc2 have the same starting time."""
    return disc1[1] == disc2[1]:

def print_students(disc):
    """Prints the name of each student in the discussion."""
    for student in disc[2]:
        print(student)

def print_duplicates(disc1, disc2):
     """Prints each student that attended both disc1 and disc2."""
    students_1, students_2 = get_students(disc1), get_students(disc2)
    for i in range(len(students_1)):
        if students_1[i] in students_2:
            print(students_1[i])
```

Below is the code with all abstraction violations removed. Notice that `print_duplicates` did not contain any data abstraction violations.

```
def check_start(disc1, disc2):
    return get_time(disc1) == get_time(disc2):

def print_students(disc):
    for student in get_students(disc2):
        print(student)

def print_duplicates(disc1, disc2):
    students_1, students_2 = get_students(disc1), get_students(disc2)
    for i in range(len(students_1)):
        if students_1[i] in students_2:
            print(students_1[i])
```

3.2 A disgruntled student makes changes to the discussion data abstraction in an attempt to disrupt the TAs' ability to run section. The new implementation is as follows:

```
def make_discussion(ta, time, students):
    return {"ta" : ta, "time" : time,  "students" : students}
```

```
def get_ta(disc):
    return disc["ta"]

def get_time(disc):
    return disc["time"]

def get_students(disc):
    return disc["students"]
```

Would the code in the previous question, with the corrections you made, still work with these changes? Would the code before removing abstraction violations still work?

After removing the abstraction violations, the code will work correctly. This is because we don't assume anything about the representation of a discussion object, so changing the representation doesn't affect anything.

Before, with abstraction violations, our code will no longer work correctly. When we try to index into a discussion as if it is a list, we will get an error, since it is now implemented as a dictionary.

4 Codewriting With Lists

Questions

4.1 Write a procedure `merge(s1, s2)` which takes two sorted (smallest value first) lists and returns a single list with all of the elements of the two lists, in ascending order. Use recursion.

Hint: If you can figure out which list has the smallest element out of both, then we know that the resulting merged list will have that smallest element, followed by the merge of the two lists with the smallest item removed. Don't forget to handle the case where one list is empty!

```
def merge(s1, s2):
    """ Merges two sorted lists
    >>> merge([1, 3], [2, 4])
    [1, 2, 3, 4]
    >>> merge([1, 2], [])
    [1, 2]
    """


    if len(s1) == 0:
        return s2
```

```
    elif len(s2) == 0:
        return s1
    elif s1[0] < s2[0]:
        return [s1[0]] + merge(s1[1:], s2)
    else:
        return [s2[0]] + merge(s1, s2[1:])
```

4.2 Consider the subset sum problem: you are given a list of integers and a number k. Is there a subset of the list that adds up to k? For example:

```
>>> subset_sum([2, 4, 7, 3], 5)          # 2 + 3 = 5
True
>>> subset_sum([1, 9, 5, 7, 3], 2)
False
>>> subset_sum([1, 1, 5, -1], 3)
False
```

Note: You can use the `in` operator to determine if an element belongs to a list:

```
>>> 3 in [1, 2, 3]
True
>>> 4 in [1, 2, 3]
False
```

```
def subset_sum(seq, k):


    if len(seq) == 0:
            return False
    elif k in seq:
            return True
    else:
            return subset_sum(seq[1:], k - seq[0]) or \
                   subset_sum(seq[1:], k)
```