CS 61A Summer 2018

Mutation & Data Abstraction

Discussion 4: July 3, 2018

1 Mutable Lists

Let's imagine you order a mushroom and cheese pizza from La Val's, and that they represent your order as a list:

```
>>> pizza = ['cheese', mushrooms']
```

A couple minutes later, you realize that you really want onions on the pizza. Based on what we know so far, La Val's would have to build an entirely new list to add onions:

```
>>> pizza = ['cheese', mushrooms']
>>> new_pizza = pizza + ['onions'] # creates a new python list
>>> new_pizza
['cheese', mushrooms', 'onions']
>>> pizza # the original list is unmodified
['cheese', 'mushrooms']
```

This is silly, considering that all La Val's had to do was add onions on top of `pizza` instead of making an entirely new pizza.

We can fix this issue with **list mutation**. In Python, some objects, such as lists and dictionaries, are **mutable**, meaning that their contents or state can be changed over the course of program execution. Therefore, instead of building a new pizza, we can just mutate `pizza` to add some onions!

```
>>> pizza.append('onions')
>>> pizza
['cheese', 'mushrooms', 'onions']
```

`append` is what's known as a method, or a function that belongs to an object, so we have to call it using dot notation. Don't worry too much about the details of methods; we will talk more about them later on in the course. For now, here's a list of useful list mutation methods:

1. `append(el)`: Adds `el` to the end of the list
2. `extend(lst)`: Extends the list by concatenating it with `lst`
3. `insert(i, el)`: Insert `el` at index `i` (does not replace element but adds a new one)

4. `remove(el)`: Removes the first occurrence of `el` in list, otherwise errors
5. `pop(i)`: Removes and returns the element at index `i`

We can also use the familiar indexing operator with an assignment statement to change an existing element in a list. For example, let's say you want to replace mushrooms on your pizza with tomatoes. We can index into the list at index 1 and reassign it to `'tomatoes'` like so:

```
>>> pizza[1] = 'tomatoes'
>>> pizza
['cheese', 'tomatoes', 'onions']
```

Although lists and dictionaries are mutable, many other objects, such as numeric types, tuples, and strings, are *immutable*, meaning they cannot be changed once they are created.

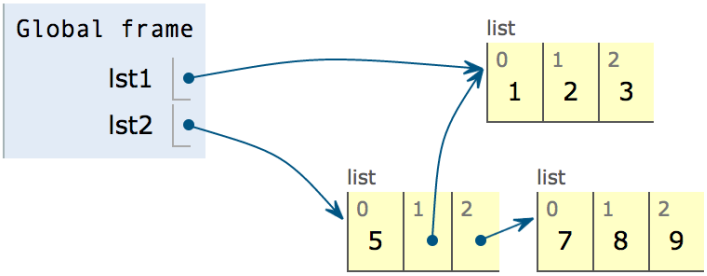
Box-and-Pointer Diagrams

So far, we've been working with fairly simple lists whose contents we can visualize in our heads. With the introduction of list mutation, programs containing multiple list objects, especially nested lists, become very difficult to keep track of.

To help us better visualize such programs, we can draw **box-and-pointer diagrams** for lists. In a box-and-pointer diagram, each element in the list goes into a box. The boxes are chained together to create a sequence. Primitive values, like numbers, are written directly into the box. Non-primitive values, such as other lists, are drawn outside of the box and are pointed to with an arrow. You can also label each box with its index to make it easier to read.

Below is a diagram for the following code:

```
lst1 = [1, 2, 3]
lst2 = [5, lst1, 6]
lst2[2] = [7, 8, 9]
```



Questions

1.1 What would Python display? It may be helpful to draw the box and pointer diagrams to the right in order to keep track of the state.

```
>>> lst1 = [1, 2, 3]
>>> lst2 = lst1
>>> lst1 is lst2


>>> lst2.extend([5, 6])
>>> lst1[4]


>>> lst1.append([-1, 0, 1])
>>> -1 in lst2


>>> lst2[5]


>>> lst3 = lst2[:]
>>> lst3.insert(3, lst2.pop(3))
>>> len(lst1)


>>> lst1[4] is lst3[6]


>>> lst3[lst2[4][1]]


>>> lst1[:3] is lst2[:3]


>>> lst1[:3] == lst3[:3]
```

1.2 Write a function that takes in a value `x`, a value `el`, and a list and adds as many `el`'s to the end of the list as there are `x`'s. **Make sure to modify the original list using list mutation techniques.**

```
def add_this_many(x, el, lst):
    """ Adds el to the end of lst the number of times x occurs
    in lst.
    >>> lst = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5]
    >>> add_this_many(2, 2, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5, 2, 2]
    """
```

1.3 Write a function that takes in a list and reverses it *in place*, i.e. mutate the given list itself, instead of returning a new list.

```
def reverse(lst):
    """ Reverses lst in place.
    >>> x = [3, 2, 4, 5, 1]
    >>> reverse(x)
    >>> x
    [1, 5, 4, 2, 3]
    """
```

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, ["Alice", "Bob"])
    >>> new_disc = add_student(disc, "Carol")
    >>> get_students(new_disc)
    ["Alice", "Bob", "Carol"]
    >>> get_students(disc)
    ["Alice", "Bob"]
    """
```

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.

Questions

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]
```

2.2 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.

```
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])
```

2.3 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?