LISTS AND DATA ABSTRACTION 4

COMPUTER SCIENCE 61A

July 3, 2014

1 Lists

A list is an ordered collection of values. In Python, we can have lists of whatever values we want, be it integers, strings, functions, or even other lists! Furthermore, the types of the list's contents need not be the same; in other words, the list need not be homogeneous.

Lists can be created using square braces, and likewise, their elements can be accessed via square braces (we call this indexing). Lists are zero-indexed; to access their first element, we must index at 0, to access their second element, we must index at 1, and to access their i^{th} element, we must index at $i - 1$. Also handy, we can access the last element at index -1, the second to last at -2, up to the first at $-l$, where l is the length of the list.

Let's try out some indexing basics.

1.1 Basics

1. What would Python output?

```
>>> a = [1, 5, 4, 2, 3]
>>> print(a[0], a[-1])
```

Solution:

```
1 3
```

```
>>> a[2] + a[-2]
```

Solution: 6

```
>>> len(a)
```

Solution: 5

```
>>> 4 in a
```

Solution: True

```
>>> a[3] == a[-2]
```

Solution: True

1.2 Slicing

If we'd like to get back parts of the list, as opposed to single elements, we *slice* the list. Slicing a list *does not change the list itself*, but instead returns a copy containing some subset of the original list. To slice, we use the following syntax:

```
lst[start:end:step]
```

where start, end, and step are integers. The slice includes every other `step` elements starting at the `start` index and up to but not including the `end` index. In other words, it includes elements at indices `start`, `start + step`, `start + 2*step`, `start + 3*step`, and so on up to end. It is legal to omit one or more of `start`, `end`, and `step` - they default to 0, `len(lst)`, and 1, respectively. Like indices, `step` can also be negative - that means you count backwards. In the case when `step` is negative, the default values of `start` and `end` are `-1` and `-(len(lst) + 1)` respectively.

```
>>> a = [0, 1, 2, 3, 4, 5, 6]
>>> a[-6:4]
[1, 2, 3]
>>> a[1:6:2]
[1, 3, 5]
>>> a[:4]
[0, 1, 2, 3]
>>> a[3:]
[3, 4, 5, 6]
```

```
>>> a[1:4:]
[1, 2, 3]
>>> a[-1:]
[6]
```

1. What would Python print?

```
>>> a = [3, 1, 4, 2, 5, 3]
>>> a[:4]
```

Solution: `[3, 1, 4, 2]`

```
>>> a
```

Solution: `[3, 1, 4, 2, 5, 3]`

```
>>> a[1::2]
```

Solution: `[1, 2, 3]`

```
>>> a[:]
```

Solution: `[3, 1, 4, 2, 5, 3]`

```
>>> a[4:2]
```

Solution: `[]`

```
>>> a[1:-2]
```

Solution: `[1, 4, 2]`

```
>>> a[3::-1]
```

Solution: `[2, 4, 1, 3]`

1.3 Adding

If we'd like to combine two lists into one, we *add* the lists. Adding two lists together creates a new list that contains the elements of both lists.

```
>>> a = [0, 1, 2, 3]
>>> b = [5, 6]
>>> a + b
[0, 1, 2, 3, 5, 6]
>>> b + a + a
[5, 6, 0, 1, 2, 3, 0, 1, 2, 3]
```

1.4 For loops

There are two common methods of looping through lists.

- `for el in lst` → loops through the elements in lst
- `for i in range(len(lst))` → loops through the valid, positive indices of lst

If you do not need indices, looping over elements is usually more clear. Let's try this out.

1. Write a function that returns a new list that takes the elements of a given list and multiplies each element by its index.

```
def multiply_index(lst):
    """
    >>> x = [4, 2, 5, 1]
    >>> y = multiply_index(x)
    >>> y
    [0, 2, 10, 3]
    """
```

Solution:

```
    result = []
    for i in range(len(lst)):
        result = result + [i * lst[i]]
    return result
```

2. Write a function that rotates the elements of a list to the left by k. Elements should not "fall off"; they should wrap around the beginning of the list.

```
def rotate(lst, k):
    """ Return a new list, with the same elements
        of lst, rotated to the left by k.
    >>> x = [1, 2, 3, 4, 5]
    >>> rotate(x, 2)
    [3, 4, 5, 1, 2]
    """
```

Solution:

```
    n = len(lst)
    rotated = []
    for i in range(n):
        j = (i + k) % n
        rotated = rotated + [ lst[j] ]
    return rotated
```

or

```
    return lst[k:] + lst[:k]
```

2 Data Abstraction

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

This is especially important when programming with other people: with data abstraction, your group members won't have to read through every line of your code to understand how it works before they use it — they can just assume that it does work.

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.

To facilitate data abstraction, you will need to create two types of functions: constructors and selectors. Constructors are functions that build the abstract data type. Selectors are functions that retrieve information from the data type.

For example, say we have an abstract data type called `city`. This `city` object will hold the `city`'s name, its latitude and longitude, and its population.

To create a `city` object, you'd use a function like

```
city = make_city(name, lat, lon, pop)
```

To extract the information of a `city` object, you would use functions like

```
get_name(city)
get_lat(city)
get_lon(city)
get_pop(city)
```

The following code will compute the distance between two city objects:

```
from math import sqrt
def distance(city1, city2):

    lat_1, lon_1 = get_lat(city_1), get_lon(city_1)
    lat_2, lon_2 = get_lat(city_2), get_lon(city_2)

    return sqrt((lat_1 - lat_2)**2 + (lon_1 - lon_2)**2)
```

Notice that we don't need to know how these functions were implemented. We are assuming that someone else has defined them for us.

It's okay if the end user doesn't know how functions were implemented. However, the functions still have to be defined by someone. We'll look into defining the constructors and selectors later in this discussion.

2.1 Data Abstraction Practice

1. Implement `closer_city`, a function that takes a latitude, longitude, and two cities, and returns the name of the city that is closer.

   You may only use selectors and constructors (introduced above) for this question. You may also use the `distance` function defined above.

   ```
   def closer_city(lat, lon, city1, city2):
   ```

   Solution:

   ```
       new_city = make_city('arb', lat, lon)
       dist1 = distance(city1, new_city)
       dist2 = distance(city2, new_city)
       if dist1 < dist2:
           return get_name(city1)
       return get_name(city2)
   ```

3 Let's be rational!

In lecture, we discussed the `rational` data type, which represents fractions with the following methods: `rational(n, d)` - constructs a rational number with numerator `n`, denominator `d`:

`numer(x)` - returns the numerator of rational number `x`
`denom(x)` - returns the denominator of rational number `x`

We also presented the following methods that perform operations with rational numbers:

```
add_rationals(x, y)
mul_rationals(x, y)
eq_rationals(x, y)
```

You'll soon see that we can do a lot with just these simple methods.

3.1 Rational Number Practice

1. Write a function that returns the given rational number `x` raised to positive power `e`.

```
from math import pow
def rational_pow(x, e):
```

Solution:

```
    return rational(pow(numer(x), e), pow(denom(x), e))
```

2. The irrational number $e \approx 2.718$ can be generated from an infinite series. Let's try calculating it using our rational number data type! The mathematical formula is as follows:

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} \cdots$$

Write a function `approx_e` that returns a rational number approximation of e to `iter` amount of iterations. We've provided a factorial function.

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n - 1)

def approx_e(iter=100):
```

Solution:

```
    k = 0
    e = rational(0, 1)
    while k < iter:
        e = add_rationals(e, rational(1, factorial(k)))
        k += 1
    return e
```

3. Implement the following rational number methods.

```
def inverse_rational(x):
    """Returns the inverse of the given non-zero rational number"""
```

Solution:

```
    return rational(denom(x), numer(x))
```

```
def div_rationals(x, y):
    """Returns x / y for given rational x and non-zero rational y"""
```

Solution:

```
    return mul_rationals(x, inverse_rational(y))
```

4 Abstracting A City

In the earlier `city` example, we worked under the assumption that the city object existed. Now let's try using data abstraction to create the city itself.

4.1 Constructing and Selecting

Remember that a city is defined by its name, latitude, longitude, and population. One way that we have learned to store a collection of values is by using a list.

1. Write a city constructor that stores these values:

```
def make_city(name, lat, lon, pop):
```

Solution:

```
    return [name, lat, lon, pop]
```

2. Now write the selectors which allow us to get information from the `city`:

```
def get_name(city):
```

Solution:

```
    return city[0]
```

```
def get_lat(city):
```

Solution:

```
    return city[1]
```

```
def get_lon(city):
```

Solution:

```
    return city[2]
```

```
def get_pop(city):
```

Solution:

```
    return city[3]
```

4.2 Data Abstraction Violations

Data abstraction violations happen when we assume we know something about how our data is represented. For example, if we assume `city` stores its data as a list and directly index rather than using a selector:

```
>>> berkeley = make_city("Berkeley", 38, 122, 115403)
>>> print("I attend UC", berkeley[0])
I attend UC Berkeley
```

In this example, we assume that `berkeley` is represented as a list and that we can use square bracket indexing. However, this violates data abstraction, and we should have instead used the selector `get_name`. Why is this bad?

Suppose that you decided later that you wanted to put the name of the city after latitude and longitude. You change the constructor:

```
def make_city(name, lat, lon):
    return [lat, lon, name, pop]
```

and change the corresponding selectors:

```
def get_name(city):
    return city[2]

def get_lat(city):
    return city[0]

def get_lon(city):
    return city[1]
```

Now, the print statement will print out "I attend UC 38", which is not what we were expecting. If we had instead used the selector, we would not have faced this issue. This may not seem like a problem for simple data structures, but it becomes much more important when you have to use complex data structures written and maintained by other people.

4.3 Higher Order Functions

Another way of storing a collection of data is with higher order functions. Recall that a `pair` is a way to store two items of data using higher-order functions. The constructor which makes a pair is given by

```
def pair(a, b):
    """
    Constructs a pair of a and b.
    """
    return lambda m: m(a, b)
```

The selectors which allow you to get the first and second element are

```
def car(pair):
    """
    Returns the first element of the pair.
    """
    return p(lambda x, y: x)

def cdr(pair):
    """
    Returns the second element of the pair.
    """
    return p(lambda x, y: y)
```

1. Using `pair`, write `quartet`, a data structure which uses higher-order functions to store four elements.

```
def quartet(a, b, c, d):
    """
    A data structure which behaves like a four-element list.
    """
```

Solution:

```
    return pair(pair(a, b), pair(c, d))
```

2. Define the selector get_item.

```
def get_item(quartet, i):
    """
    Returns the item at index i (0, 1, 2, or 3).
    """
```

Solution:

```
    if i == 0:
        return car(car(quartet))
    elif i == 1:
        return cdr(car(quartet))
    elif i == 2:
        return car(cdr(quartet))
    else:
        return cdr(cdr(quartet))
```

3. Going back to our city object we created earlier, write a constructor for a city and a selector for its name using higher-order functions this time.

```
def make_city_hof(name, lat, lon, pop):
```

Solution:

```
    return quartet(name, lat, lon, pop)
```

```
def get_name_hof(city):
```

Solution:

```
    return get_item(city, 0)
```

Data abstractions are extremely useful when the underlying implementation of the abstraction changes.

For example, after writing a program using lists as a way of storing data, suddenly someone switches the implementation to higher order functions. If we correctly use constructors and selectors, our program should still work perfectly.

4.4 Challenge Problem: A Slice of Fun

The beauty of data abstraction is that we can treat complex data in a very simple way. Although we've only been dealing with storing primitive data types, we can store complex data in the same way. An example is nesting lists inside of each other. Let's say for example we wanted to make a `car` data type, which stores a `make`, a `model`, a `year`, and a list of `values`, which is a list representing the price of the car at different years. The first element is the price at purchase, the second element the price one year later, and so on.

```
>>> def make_car(make, model, year, values):
        return [make, model, year, values]
>>> def get_value_list(car):
        return car[3]
>>> bumblebee = make_car("Chevy", "Camaro", 1977, [75,000, 45,000, \
                             30,000, 100,000])
```

1. Write a method `get_values` that takes in a car object, a start year, and an end year, and returns a list of prices over that time using the original 4 selectors. (Note that now the `get_value_list` selector returns the entire list of the prices.)

```
def get_values(car, start, end):
    """
    >>> get_values(bumblebee, 1978, 1979)
    [45,000, 30,000]
    """
```

Solution:

```
    start_index = start     get_year(car)
    end_index = end     get_year(car)
    return get_value_list(car)[start_index:end_index+1]
```

2. Later, you realize that it is wasteful to have a selector that gets the entire list - normally you don't need the entire list but just a few elements in it. So instead you decide to make `get_values` a selector itself and remove the `get_value_list` selector. Write the new version of `get_values`.

```
def get_values(car, start, end):
```

Solution:

```
    start_index = start     get_year(car)
    end_index = end     get_year(car)
    return car[3][start_index:end_index+1]
```

What is the difference between these two implementations? It depends entirely on how you want to design your object. There is no right answer, as both forms can be advantageous depending on what suits the needs of your program.