DATA ABSTRACTION AND SEQUENCES 3

COMPUTER SCIENCE 61A

February 11, 2016

1 Sequences

A *sequence* is an ordered collection of values. It has two fundamental properties: length and element selection. In this discussion, we'll explore one of Python's data types, the *list*, which implements this abstraction.

In Python, we can have lists of whatever values we want, be it numbers, 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 homogenous.

Lists can be created using square braces. Their elements can be accessed (or *indexed*) with square braces. Lists are zero-indexed: to access the first element, we must index at 0; to access the ith element, we must index at $i - 1$.

We can also index with negative numbers. These begin indexing at the end of the list, so the index -1 is equivalent to the index `len(list) - 1` and index -2 is the same as `len(list) - 2`.

Let's try out some indexing:

```
>>> fantasy_team = ['frank gore', 'calvin johnson']
>>> print(fantasy_team)
['frank gore', 'calvin johnson']
>>> fantasy_team[0]
'frank gore'
>>> fantasy_team[len(fantasy_team) - 1]
'calvin johnson'
>>> fantasy_team[-1]
'calvin johnson'
```

If we have two lists, we can use the + operator to create a new list with the values of the original two lists, concatenated together.

```
>>> mouse_names = ['Picasso', 'Fred']
>>> dog_names = ['Spot', 'Rusty']
>>> pet_names = mouse_names + dog_names
>>> pet_names
['Picasso', 'Fred', 'Spot', 'Rusty']
```

Sequences also have a notion of length, the number of items stored in the sequence. In Python,we can check how long a sequence is with the `len` built-in function.

We can also check if an item exists within a list with the `in` statement.

```
>>> poke_list = ['Meowth', 'Mewtwo']
>>> len(poke_list)
2
>>> 'Meowth' in poke_list
True
>>> 'Pheobe' in poke_list
False
```

1. What would Python print?

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

   >>> len(a)

   >>> 2 in a

   >>> 4 in a

   >>> a[3][0]
   ```

1.1 Slicing

If we want to access more than one element of a list at a time, we can use a *slice*. Slicing a sequence is very similar to indexing. We specify a starting index and an ending index, separated by a colon. Python creates a new list with the elements from the starting index up to (but not including) the ending index.

We can also specify a step size, which tells Python how to collect values for us. For example, if we set step size to 2, the returned list will include every **other** value, from the starting index until the ending index. A negative step size indicates that we are stepping backwards through a list when collecting values.

If the step size is left out, the default step size is 1. If either the start or end indices are left out, the slice starts at the beginning and ends at the end of the list. When the step size is negative, the slice starts at the end and ends at the beginning of the list.

Thus, `lst[:]` creates a list that is identical to `lst` (a copy of `lst`). `lst[::-1]` creates a list that has the same elements of `lst`, but reversed. Those rules still apply if more than just the step size is specified e.g. `lst[3::-1]`.

```
>>> pet_list = ['Mochi', 'Picasso', 'Rusty', 'Pheobe']
>>> pet_list[:2]
['Mochi', 'Picasso']
>>> pet_list[1:3]
['Picasso', 'Rusty']
>>> pet_list[1:]
['Picasso', 'Rusty', 'Pheobe']
>>> pet_list[0:4:2]
['Mochi', 'Rusty']
>>> pet_list[::-1]
['Pheobe', 'Rusty', 'Picasso', 'Mochi']
```

1. What would Python print?

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

>>> a[:]

>>> a[4:2]

>>> a[1:-2]

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

1.2 List Operators

Many times, we wish an operation to be applied to all elements of a list. Python has built-in methods to help us with these tasks. (*Note:* reduce was hidden in Python 3.)

- `map(fn, lst)` applies fn to each element in lst
- `filter(pred, lst)` keeps those elements in lst that satisfy the predicate
- `reduce(accum, lst, zero_value)` repeatedly calls the accumulator function, which takes in two arguments and returns a single value, on elements of lst.

An important thing to note is that these operators do not change the input list, and they also do not return lists. You must call the `list` operator on them to get a list back.

1. What would Python print?

```
>>> fn1, fn2 = lambda x: 3*x + 1, lambda x: x % 2 == 0
>>> list(filter(fn2, map(fn1, [1, 2, 3, 4])))
```

2 List Comprehensions and For Loops

2.1 For Loops

There are two common methods of looping through lists. If you don't need indices, looping over elements is usually more clear.

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

2.2 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>]
```

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.

1. What would Python print?

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

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

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

2. Define a function `foo` that takes in a list `lst` and returns a new list that keeps only the even-indexed elements of `lst` and multiplies each of those elements by the corresponding index.

```
def foo(lst):
    """
    >>> x = [1, 2, 3, 4, 5, 6]
    >>> foo(x)
    [0, 6, 20]
    """

    return [_____________________________________________________________]
```

3 My Life for Abstraction

So far, we've only *used* data abstractions. Now let's try *creating* some! In the next section, we'll be looking at two ways of implementing abstract data types: lists and functions.

3.1 Lists, or Zerg Rush!

One way to implement abstract data types is with the Python list construct.

```
>>> nums = [1, 2]
>>> nums[0]
1
>>> nums[1]
2
```

We use the square bracket notation to access the data we stored in `nums`. The data is *zero indexed*: we access the first element with `nums[0]` and the second with `nums[1]`.

Let's now use data abstractions to recreate the popular video game Starcraft: Brood War. In Starcraft, the three races, Zerg, Protoss, and Terran, create "units" that they send to attack each other.

1. Implement the constructors and selectors for the unit data abstraction using lists. Each unit will have a string catchphrase and an integer amount of damage.

```
def make_unit(catchphrase, damage):



def get_catchphrase(unit):



def get_damage(unit):
```

3.2 Data Abstraction Violations, or, I Long For Combat!

Data abstraction violations happen when we assume we know something about how our data is represented. For example, if we use pairs and we forget to use a selector and instead use the index.

```
>>> raynor = make_unit('This is Jimmy.', 18)
>>> print(raynor[0]) # violation!!!!
```

```
This is Jimmy.
```

In this example, we assume that `raynor` is represented as a list because we use the square bracket indexing. However, we should have used the selector `get_catchphrase`. This is a data abstraction violation.

1. Let's simulate a battle between units! In a battle, each unit yells its respective catchphrase, then the unit with more damage wins. Implement `battle`, which prints the catchphrases of the first and second unit in that order, then returns the unit that does more damage. The first unit wins ties. Don't violate any data abstractions!

```
def battle(first, second):
    """Simulates a battle between the first and second unit
    >>> zealot = make_unit('My life for Aiur!', 16)
    >>> zergling = make_unit('GRAAHHH!', 5)
    >>> winner = battle(zergling, zealot)
    GRAAHHH!
    My life for Aiur!
    >>> winner is zealot
    True
    """
```

3.3 Functional Pairs, or, We Require More Minerals

The second way of constructing abstract data types is with higher order functions. We can implement the functions `pair` and `select` to achieve the same goal.

```
>>> def pair(x, y):
        """Return a function that represents a pair of data."""
        def get(index):
            if index == 0:
                return x
            elif index == 1:
                return y
```

```
        return get
>>> def select(p, i):
        """Return the element at index i of pair p"""
        return p(i)
>>> nums = pair(1, 2)
>>> select(nums, 0)
1
>>> select(nums, 1)
2
```

Note how although using functional pairs is different syntactically from lists, it accomplishes the exact same thing.

We can tie this in with our continuing Starcraft example. Units require resources to create, and in Starcraft, these resources are called ”minerals” and ”gas.”

1. Write constructors and selectors for a data abstraction that combines an integer amount of minerals and gas together into a bundle. Use functional pairs.

```
def make_resource_bundle(minerals, gas):


def get_minerals(bundle):


def get_gas(bundle):
```

3.4 Putting It All Together

1. Let’s make a `building` pair that is constructed with a unit data type and a resource bundle data type. This time take your choice of lists or functional pairs in representing a building. Make sure not to violate any data abstractions.

```
def make_building(unit, bundle):


def get_unit(building):


def get_bundle(building):
```

2. 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 pairs, suddenly someone switches the implementation to functional pairs. If we correctly use constructors and selectors, our program should still work perfectly.

   Reimplement the `resource` abstraction to use lists instead of functional pairs. Then verify that all the code that use the `resource` still works.

```
def make_resource_bundle(minerals, gas):


def get_minerals(bundle):


def get_gas(bundle):
```