CS 61A
Discussion 3

Data Abstraction and Sequences

Raymond Chan
Discussion 121
UC Berkeley

Agenda

- Announcements
- Lists
- List Comprehension
- Data Abstraction
- Quiz

Announcements

- HW 2 released and due next Tuesday 2/16
- Midterm 1!!! next Thursday 2/18
  - Check piazza for logistics
- Lab 3 due Friday 2/12

Midterm 1 Tips

- No need to **re**-watch lecture.
- Do questions!!! Start with extra questions for labs and move on to past midterms (check HKN to TBP websites)
- Practice environment diagrams.
    - Follow the rules and its easy points.
- Start studying early.
- Don’t worry about if midterm questions are hard at first, you will understand them with practice and time.

Sequences

- Ordered collection of values
- Length
- Element Selection

List

- Sequence - order collection of values
- Python list is a type of sequence of whatever values we want.
  - numbers, strings, functions, lists
- Create a list using [] (square brackets).
  - [1, 2, 3, 4, 5]
- Type of a single list's content does not need to be the same.
  - [1, “two”, lambda : 3, 4, True]

List

- We can access, or index, any element with square brackets.
- Lists are **zero-indexed**.
    - First element is at index 0
    - *i*-th element is indexed at *i* -1
- Can have negative index
    - If a list has a length of *n*, we can index from *-n* to *n* -1.

List

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0]
```

List

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0]
1
```

List

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0]
1
>>> L[3]
```

List

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0]
1
>>> L[3]
4
```

List

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0]
1
>>> L[3]
4
>>> L[5]
```

List

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0]
1
>>> L[3]
4
>>> L[5]
Index OutOfBounds Error
```

List

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0]
1
>>> L[3]
4
>>> L[5]
Index OutOfBounds Error
>>> L[-4]
```

List

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0]
1
>>> L[3]
4
>>> L[5]
Index OutOfBounds Error
>>> L[-4]
2
```

List

- With multiple lists, we can concatenated them together using +

```
>>> odds = [1, 3, 5, 7]
>>> evens = [2, 4, 6]
>>> odds + evens
[1, 3, 5, 7, 2, 4, 6]
```

List

- To obtain the length of a sequence, use the **len** built-in function

```
>>> odds = [1, 3, 5, 7]
>>> len(odds)
4
>>> odds[len(odds) - 1]
7
```

List

- Check if an element exists in a list with **in**

```
>>> odds = [1, 3, 5, 7]
>>> 5 in odds
True
>>> 2 in odds
False
```

List Slicing

- We can get a certain part of a list via slicing
- list[<start>:<stop>:<step>]
- Our new list beings at ***start***, takes every ***step***-th element (or jump by ***step***), and ends at index before ***stop.***
- If it cannot reach ***stop***, it will return an empty list.
- By default step is 1
- Slicing will **always** create a **new list.**

List Slicing

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
```

List Slicing

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
```

List Slicing

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
```

List Slicing

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
```

List Slicing

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
```

List Slicing

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
```

List Slicing

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
```

List Slicing

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
['1', 'a', ‘is']
```

List Slicing

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
['1', 'a', ‘is']
>>> lst[-3: -5]
```

List Slicing

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', ‘is']
>>> lst[3:100]
['1', 'a', 'is', 'so', ‘fun']
>>> lst[2:6:2]
['6', ‘a’]
>>> lst[-5: -2]
['1', 'a', ‘is']
>>> lst[-3: -5]
[]
```

List Slicing

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', 'is']
>>> lst[3:100]
['1', 'a', 'is', 'so', 'fun']
>>> lst[2:6:2]
['6', 'a']
>>> lst[-5: -2]
['1', 'a', 'is']
>>> lst[-3: -5]
[]
```

```
>>> lst[4:2]
```

List Slicing

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', 'is']
>>> lst[3:100]
['1', 'a', 'is', 'so', 'fun']
>>> lst[2:6:2]
['6', 'a']
>>> lst[-5: -2]
['1', 'a', 'is']
>>> lst[-3: -5]
[]
```

```
>>> lst[4:2]
[]
```

List Slicing

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', 'is']
>>> lst[3:100]
['1', 'a', 'is', 'so', 'fun']
>>> lst[2:6:2]
['6', 'a']
>>> lst[-5: -2]
['1', 'a', 'is']
>>> lst[-3: -5]
[]
```

```
>>> lst[4:2]
[]
>>> lst[2:7]
```

List Slicing

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', 'is']
>>> lst[3:100]
['1', 'a', 'is', 'so', 'fun']
>>> lst[2:6:2]
['6', 'a']
>>> lst[-5: -2]
['1', 'a', 'is']
>>> lst[-3: -5]
[]
```

```
>>> lst[4:2]
[]
>>> lst[2:7]
['6', '1', 'a', 'is', 'so']
```

List Slicing

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', 'is']
>>> lst[3:100]
['1', 'a', 'is', 'so', 'fun']
>>> lst[2:6:2]
['6', 'a']
>>> lst[-5: -2]
['1', 'a', 'is']
>>> lst[-3: -5]
[]
```

```
>>> lst[4:2]
[]
>>> lst[2:7]
['6', '1', 'a', 'is', 'so']
>>> lst[2:7:-4]
```

List Slicing

```
>>> lst = ['c','s','6','1','a','is', 'so', 'fun']
>>> lst[3:6]
['1', 'a', 'is']
>>> lst[3:100]
['1', 'a', 'is', 'so', 'fun']
>>> lst[2:6:2]
['6', 'a']
>>> lst[-5: -2]
['1', 'a', 'is']
>>> lst[-3: -5]
[]
```

```
>>> lst[4:2]
[]
>>> lst[2:7]
['6', '1', 'a', 'is', 'so']
>>> lst[2:7:-4]
[]
```

List Operators

- list(lst)
  - creates a copy of lst

```
>>> primes = [3, 5, 7, 11, 13]
>>> list(primes)
[3, 5, 7, 11, 13]
```

List Operators

- map(fn, lst)
  - applies fn to each element of lst

```
>>> nums = [1, 2, 3, 4, 5, 6, 7]
>>> list(map(square, nums)
[1, 4, 9, 16, 25, 36, 49]
```

List Operators

- filter(pred, lst)
  - keep elements if calling pred on that element is True

```
>>> nums = [1, 2, 3, 4, 5, 6, 7]
>>> list(filter(is_prime, nums)
[2,3, 5, 7]
```

List Operators

- reduce(accum, lst, zero_value)
    - Iterates elements in list and repeatedly calls the accumulator function
    - The accumulator function takes in 2 arguments
    - Map returns a single value
    - If zero_value is passed in, we call accum(zero_value, lst[0]) first.

List Operators

```
>>> nums = [1, 2, 3, 4, 5, 6, 7]
>>> reduce(add, nums) # 1+2+3+4+5+6+7
28
>>> reduce(add, nums, 55) # 55+1+2+3+4+5+6+7
83
```

List Operators

- The operators do not change the input list
- Map and filter do not return lists; thus we need to call the ‘list’ operator
- Map and filter return <map object> and <filter object> respectively
- From what we have learnt so far, the “new” list is hidden within the abstraction.

List Operators

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

- Think about that filter is able to iterate through the list that is within the abstraction of the <map object>

Discussion Questions

- Q1 page 2
- 1.1 Q1 page 3
- 1.2 Q1 page 4

For Loops

- Another method of iteration
- for <variable> in <sequence>

```
>>> lst [1, 2, 3]
>>> for x in lst:
 ...         print(x)
1
2
3
```

```
>>> for i in range(0, 6):
...         print(i)
0
1
2
3
4
5
```

For Loops

- range(<start>, <stop>,<step>)
- allows a for loop to iterate through a sequence from ***start*** up to and excluding ***stop***, taking every ***step***-th element.
- By default <step> is 1.
- Must have <start> and <stop>

for i in range(0, 5, 2)
for i in range(2, 5)

List Comprehension

- Compact way to create a list
- [<map exp> for <name> in <iter exp> if <filter exp>]
- if clause is optional

```
nums = [1, 2, 3, 4, 5, 6, 7]
lst = []
for x in nums:
    if x % 2 == 0:
        lst += [x+3]

[x + 3 for x in nums if x % 2 == 0]
```

Discussion Questions

- 2.2 Q1, 2 page 5,6

Data Abstraction

- Most of the time we need to work on code that was implemented by someone else.
- Via data abstraction, we don’t need to worry about how the implementation of the data.
- We just need to know how to use the data.
- Why is it useful?

Data Abstraction

- Why is it useful?
- If we were to change the implementation of a ADTs, we only need to change the constructors and selectors.
- Any functions we wrote that used the selectors **do not** need to be changed!

Data Abstraction

- We can treat data as abstract data types
- Constructors create these ADTs
- Selectors are used to retrieve information from ADTs

Data Abstraction

Constructor:

```
def make_city(city, latitude, longitude):
    return [city, latitude, longitude]
```

Selectors:

```
def get_name(city):
    return city[0]
def get_lat(city):
    return city[1]
def get_lon(city):
    return city[2]
```

Data Abstraction Violations

- When we use the direct implementation of an ADT rather than its selectors when writing functions, we are violating data abstraction barriers!
- This is bad because we are making an assumption on how the data is implemented.

Data Abstraction Violations

```
def distance(city1, city2):
    lat_1, lon_1 = get_lat(city1), get_lon(city1)
    lat_2, lon_2 = get_lat(city2), get_lon(city2)
    return sqrt((lat_1 - lat_2)**2 + (lon_1 - lon_2)**2)
```

```
def distance(city1, city2):
    lat_1, lon_1 = city[1], city[2]
    lat_2, lon_2 = city[1], city[2]
    return sqrt((lat_1 - lat_2)**2 + (lon_1 - lon_2)**2)
```

Data Abstraction Violations

GOOD

```
def distance(city1, city2):
    lat_1, lon_1 = get_lat(city1), get_lon(city1)
    lat_2, lon_2 = get_lat(city2), get_lon(city2)
    return sqrt((lat_1 - lat_2)**2 + (lon_1 - lon_2)**2)
```

BAD

```
def distance(city1, city2):
    lat_1, lon_1 = city[1], city[2]
    lat_2, lon_2 = city[1], city[2]
    return sqrt((lat_1 - lat_2)**2 + (lon_1 - lon_2)**2)
```

Discussion Questions

- Qs in 3.1 and 3.2 in data abstraction section

Recap

- Lists contain a sequence of values of which we can access via indexing.
- List slicing creates a new list of a certain portion of the original list.
- For loops are a way to iterate through sequences.
- List comprehension creates a new list in one line.
- Data abstraction is useful in that we do not need to worry about implementation.

Quiz

```
y = 10

def f(fn, x):
    def g(y):
        if x == 1:
            print("It's finally over!")
            return fn(x, y)
        elif y > 7:
            print("T.L.O.P")
            return g(y-2)
        else:
            print("Waves")
            return f(fn, y-1)(x-1)
    return g

print(f(lambda a, b: a + y, 3)(9))
```

Quiz (Solutions)

```
y = 10

def f(fn, x):
    def g(y):
        if x == 1:
            print("It's finally over!")
            return fn(x, y)
        elif y > 7:
            print("T.L.O.P")
            return g(y-2)
        else:
            print("Waves")
            return f(fn, y-1)(x-1)
    return g

print(f(lambda a, b: a + y, 3)(9))
```

```
T.L.O.P
Waves
Waves
It’s finally over!
11
```