CS 61A Discussion 3 Sequences and Data Abstraction

Announcements

- Guerrilla section this Saturday from 12-3pm in 247 Cory (environment diagrams, HOF, recursion)

- Midterm 2/17 from 7-9pm; fill out alternate form by Sunday

- Homework 3 has been released, due 2/14 (party 6:30-8:30pm in 247 Cory)

You know what it is

(a quiz)

Sequences

Sequences are ordered collections of items.
Every sequence must have a **length** and allow for **element selection** (indexing).

Examples of Sequences

Lists

are sequences.

```
>>> len([1, 2])
2
>>> [1, 2][1]
2
```

Tuples

are sequences.

```
>>> len((1, 2))
2
>>> (1, 2)[1]
2
```

Strings

are sequences.

```
>>> len('12')
2
>>> '12'[1]
'2'
```

Sets

are **not** sequences.

```
>>> len({1, 2})
2
>>> {1, 2}[1]
Error
```

Dictionaries

are **not** sequences.

```
>>> len({1: 2, 2: 1})
2
>>> {1: 2, 2: 1}[0]
Error
```

List Manipulation

Creation

`[7, 8]` OR `list((7, 8))`

DO NOT call `list` on a non-iterable! `list(7, 8)` *WILL* error.

Population

You can put anything you want into a list.

```
[1, [1], 'one', None,
True, (1,), 1.0, {1: 1}]
```

Concatenation

Glue multiple lists together with the + operator.

```
>>> [1, 2, 3] + ['four', {5}, (6,)]
[1, 2, 3, 'four', {5}, (6,)]
```

More List Manipulation

Existence Checking

Use the `in` operator.

```
>>> your_grades = [‘a+’, ‘a-’, ‘a’, ‘a+’]
>>> ‘f’ in your_grades
False
>>> ‘a+’ in your_grades
True
```

Length Practice

```
>>> len(([4, 5], 6, ‘7’))

>>> len(([1, 2, 3]))

>>> len(‘abc’)
```

Length Practice (Solutions)

```
>>> len([[4, 5], 6, '7'])
3
>>> len([1, 2, 3])
3
>>> len('abc')
3
```

Indexing Practice

```
>>> naturals = list(range(5))
>>> naturals[1] = list(range(5))
>>> naturals
[0, [0, 1, 2, 3, 4], 2, 3, 4]
>>> naturals[-5] + naturals[4]

>>> naturals[1][1]

>>> naturals[-3][3]

>>> naturals[naturals[-4][-4]][-4]
```

Indexing Practice (Solutions)

```
>>> naturals = list(range(5))
>>> naturals[1] = list(range(5))
>>> naturals
[0, [0, 1, 2, 3, 4], 2, 3, 4]
>>> naturals[-5] + naturals[4]
4
>>> naturals[1][1]
1
>>> naturals[-3][3]
Error
>>> naturals[naturals[-4][-4]][-4]
1
```

List Slicing

Get a **new list** whose elements are some subset of the original list. Slicing involves three arguments, all of which are optional:

```
lst[start index : end index ± 1 : step size]
```

- If `step size` is omitted, it defaults to 1.
- If `start index` is omitted, it defaults to 0 if `step size` > 0, and `len(lst) - 1` if `step size` < 0.
- If `end index` is omitted, it defaults to `len(lst) - 1` if `step size` > 0, and 0 if `step size` < 0.
- It'll be `end index + 1` if `step size` > 0, and `end index - 1` if `step size` < 0.

Slicing Practice

```
>>> naturals = [list(range(4)), 4, 5]
>>> orig = naturals[:]
>>> naturals[0][2], naturals[-2] = 50, 6
>>> naturals
[[0, 1, 50, 3], 6, 5]
>>> orig

>>> naturals[1::-1]

>>> naturals[:2]

>>> naturals[:-3:-1]

>>> naturals[:2:3]

>>> naturals[0][::-1][1:5:2]
```

Slicing Practice (Solutions)

```
>>> naturals = [list(range(4)), 4, 5]
>>> orig = naturals[:]
>>> naturals[0][2], naturals[-2] = 50, 6
>>> naturals
[[0, 1, 50, 3], 6, 5]
>>> orig
[[0, 1, 50, 3], 4, 5]
>>> naturals[1::-1]
[6, [0, 1, 50, 3]]
>>> naturals[:2]
[[0, 1, 50, 3], 6]
>>> naturals[:-3:-1]
[5, 6]
>>> naturals[:2:3]
[[0, 1, 50, 3]]
>>> naturals[0][::-1][1:5:2]
[50, 0]
```

List Processing Functions

- **`map(fn, lst)`**
  - Returns an iterator over the elements of `lst`, where `fn` has been applied to all of them.
  - `list(map(lambda x: x * 2, [1, 2, 3])) → [2, 4, 6]`
- **`filter(pred, lst)`**
  - Returns an iterator over the elements of `lst` for which `pred(<elt>)` is a true value.
  - `list(filter(lambda x: x % 2, [1, 2, 3])) → [1, 3]`
- **`reduce(accum, lst, zero_value)`**
  - Repeatedly combines elements of `lst` into one value (using the `accum` function), starting with the base value `zero_value`.
  - In Python 3, you'll need to import `reduce` from `functools`.
  - `reduce(lambda x, y: x + y, [1, 2, 3]) → 6`

List Comprehensions

A more concise way to create a new list.

```
lst = [<expr> for x in <iterable> if <cond expr>]
```

- is equivalent to -

```
lst = []
for x in <iterable>:
    if <cond expr>:
        lst += [<expr>]
```

(I write x in the code above, but any name will do!)

Abstract Data Types

“I don’t care how it’s implemented. I just want to know what properties and/or behavior the data has” - Abraham Lincoln

*Only the **constructors** and the **selectors** should know how the data is really represented!*

Everything else should just reference the constructors / selectors themselves.

(Then, if you change the constructor / selector implementation, nothing else should break.)

That's it for today!

See you around. :)