

CSC148 fall 2013

abstraction and idiom

week 2

Danny Heap

heap@cs.toronto.edu

BA4270 (behind elevators)

<http://www.cdf.toronto.edu/~heap/148/F13/>

416-978-5899

September 18, 2013

Outline

abstract data types (ADTs)

implement an ADT with a class

idiomatic python

common ADTs

In CS we recycle our intuition about the outside world as ADTs. We abstract the data and operations



- ▶ sequences of items; can be added, removed, accessed by position



- ▶ specialized list where we only have access to most recently added item



- ▶ collection of items accessed by their associated keys

stack example

visit this [visualization of code](#) and step through it

The calls to `first` and `second` are stored on a stack that defies gravity by growing downward

stack class design

We'll use this real-world description of a stack for our design:

A stack contains items of various sorts. New items are pushed on to the top of the stack, items may only be popped from the top of the stack. It's a mistake to try to remove an item from an empty stack. We can tell how big a stack is, and what the top item is.

Take a few minutes to identify the main noun, verb, and attributes of the main noun, to guide our class design.

Remember to be flexible about alternate names and designs for the same class

implementation possibilities

The public interface of our Stack ADT should be constant, but inside we could implement it in various ways

- ▶ Use a python list, which already has a pop method and an append method
- ▶ Use a python list, but push and pop from position 0
- ▶ Use a python dictionary with integer keys $0, 1, \dots$, keeping track of the last index used

testing

Use your docstring for testing as you develop, but use **unit testing** to make sure that your particular implementation remains consistent with your ADT's interface. Be sure to:

- ▶ import the module `unittest`
- ▶ subclass `unittest.TestCase` for your tests, and begin each method that carries out a test with the string `test`
- ▶ compose **tests** before and during implementation

going with the (pep) tide

list comp?

```
L = []  
for i in L2:  
    L.append(2*i)
```

Python is more flexible than the community you are coding in.
Try to figure out what the **python way** is

```
sum([1, 2, 3, 7])
```

- ▶ don't re-invent the wheel (except for academic exercises),
e.g. sum, set([1, 2, 3, 7, "gosh!"])



- ▶ use comprehensions when you mean to produce a new list
(tuple, dictionary, set, ...) $[expr \text{ for } i \text{ in } iterable]$
 $[2 * i \text{ for } i \text{ in } [1, 2, 3]]$
- ▶ use ternary if when you want an expression that evaluates
in different ways, depending on a condition
 $exp1 \text{ if } condition \text{ else } exp2$



example: add (squares of) first 10 natural numbers

- ▶ You'll be generating a new list from `range(1, 11)`, so use a comprehension

$num^2 \times 2$
`[0 for num in range(1, 11)]`

- ▶ You want to add all the numbers in the resulting list, so use `sum`

list differences, lists without duplicates

- ▶ python lists allow duplicates, python sets don't
- ▶ python sets have a set-difference operator
- ▶ python built-in functions `list()` and `set()` convert types

re-use and recursion — take one!

list_sum (

[1, [2, 3], 4]

- ▶ a function `list_sum` that adds all the numbers in a nested list shouldn't ignore built-in `sum`

... sum([- .])

- ▶ ...except `sum` wouldn't work properly on the nested lists, so make a list-comprehension of their `list_sums`
- ▶ but wait, some of the list elements are numbers, not lists!

↑ base case

write a definition of `list_sum`



re-use and recursion — take one!

- ▶ a function `list_sum` that adds all the numbers in a nested list shouldn't ignore built-in `sum`
- ▶ ...except `sum` wouldn't work properly on the nested lists, so make a list-comprehension of their `list_sums`
- ▶ but wait, some of the list elements are numbers, not lists!

write a definition of `list_sum`

re-use and recursion — take one!

- ▶ a function `list_sum` that adds all the numbers in a nested list shouldn't ignore built-in `sum`
- ▶ ...except `sum` wouldn't work properly on the nested lists, so make a list-comprehension of their `list_sums`
- ▶ but wait, some of the list elements are numbers, not lists!

write a definition of `list_sum`