CSC148 winter 2014

abstraction, idiom, recursion

week 2

Danny Heap (with additions by Dustin Wehr)
heap@cs.toronto.edu (dustin.wehr@utoronto.ca)
BA4270, behind elevators (SF4306D)
http://www.cdf.toronto.edu/~heap/148/W14/

January 16, 2014

Outline

point and property...

abstract data types (ADTs)

implement an ADT with a class

idiomatic python

recursion

that vexing problem with attribute access...

Our old definition of **Point** allowed (possibly bumbling) client code to change **coord** after a point was created. We don't want that! On the other hand, we already have code shipped that uses **coord** directly. What to do?

Python's built-in function **property** intercepts all code that assigns to **coord** and passes that off to **set_coord**.

The client code, as well as code within **Point** continues to assign to, and evaluate **coord** as before, but is intercepted by **property**

common ADTs

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

-
fname, lnam
nancy, davo
erin, bora
tony, rapha
⋮
names.csv
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

ADTs - general form

- Has a name (e.g. List)
- Has descriptions of methods that are fundamental to the intuitive idea of this datatype.
- Each of those methods has a name, a given number or arguments, and optionally types on the arguments and a return type.
  `append(x:List, y:List)` $\rightarrow$ `List`
- The method descriptions should be precise enough that another person can, without knowing anything about how the methods are implemented, write a program that *uses* objects of you ADT.

Computer-readable equivalent of ADT: interface

stack example

The python interpreter (and interpreters for other languages) uses a stack to keep track of nested function calls.

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, . . . , keeping track of the last index used, and which have been popped

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

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

- don't re-invent the wheel (except for academic exercises), e.g. `sum`, `set`
- use comprehensions when you mean to produce a new list (tuple, dictionary, set, . . .)
- use ternary `if` when you want an expression that evalutes in different ways, depending on a condition
  These are equivalent:

```
if y >= 0:                f(y**.5 if y >= 0 else 0)
    x = y**0.5
else:
    x = 0
f(x)
```

example: add (squares of) first 10 natural numbers

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

example: add (squares of) first 10 natural numbers

- You'll be generating a new list from `range(1, 11)`, so use a comprehension
  ```
  [???  for x in range(1,11)]
  ```

example: add (squares of) first 10 natural numbers

- You'll be generating a new list from `range(1, 11)`, so use a comprehension
  `[x**2 for x in range(1,11)]`
- You want to add all the numbers in the resulting list, so use `sum`

example: add (squares of) first 10 natural numbers

- You'll be generating a new list from `range(1, 11)`, so use a comprehension
  ```
  [x**2 for x in range(1,11)]
  ```

- You want to add all the numbers in the resulting list, so use `sum`
  ```
  sum([x**2 for x in range(1,11)])
  ```

list differences, lists without duplicates

- python `lists` allow duplicates, python sets don't
  `set([1,1]) == set([1])`

- python `sets` have a set-difference operator:
  `set((1,2,3,4,5)) - set([2,4,6]) == set([1,3,5])`

- python built-in functions `list()` and `set()` convert types
  `type(list(set([1,2,4]))) == list`

recursion

Write a function sum_list that adds up all the numbers in a nested list.

```
def sum_list(L:list):
    total = 0
    # add to total
    # ...
    return total
```

recursion

Write a function `sum_list` that adds up all the numbers in a nested list.

```
from numbers import Number as Number
def sum_list(L:list) -> Number:
    total = 0
    for x in L:
        # If x is a number then it's clear what to do
        # ``not isinstance(x,list)'' would work too
        if isinstance(x,Number):
            total += x
        else:
            # what do we do if x is not a number?
    return total
```

recursion

Write a function sum_list that adds up all the numbers in a nested list.

```
from numbers import Number as Number
def sum_list(L:list) -> Number:
    total = 0
    for x in L:
        # If x is a number then it's clear what to do
        if isinstance(x,Number):
            total += x
        else:
            # What do we do if x is not a number?
            # Then x is a nested list of numbers,
            # which is what sum_list works on!
            # And x is smaller than L!
    return total
```

recursion

Write a function sum_list that adds up all the numbers in a nested list.

```
from numbers import Number as Number
def sum_list(L:list) -> Number:
    total = 0
    for x in L:
        # If x is a number then it's clear what to do
        if isinstance(x,Number):
            total += x
        else:
            # What do we do if x is not a number?
            # Then x is a nested list of numbers,
            # which is what sum_list works on!
            # And x is smaller than L!
            total += sum_list(x)
    return total
```

re-use and recursion (and list comprehensions)

- a function `sum_list` 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 `sum_lists`
- but wait, some of the list elements are numbers, not lists!

write a more-concise definition of `sum_list`

hey! don't peek!

```
from numbers import Number as Number
def sum_list(L: list) -> Number:
    """
    Return sum of the numbers in possibly nested list L

    >>> sum_list([1, 2, 3])
    6
    >>> sum_list([1, [2, 3, [4]], 5])
    15
    """
    return sum([sum_list(x) if isinstance(x, list) else x for x in L])
```

To understand recursion, trace from simple to complex:

- trace `sum_list([1, 2, 3])`. Remember how the built-in sum works.
- trace `sum_list([1, [2, 3], 4, [5, 6]])`. Immediately replace calls you've already traced (or traced something equivalent) by their value
- trace `sum_list([1, [2, [3, 4], 5], 6 [7, 8]])`. Immediately replace calls you've already traced by their value.

Some useful built-in functions 1

Selected from http://docs.python.org/3/library/functions.html

- `all(iterable_object)` returns true if all the elements of `iterable_object` are true (`iterable_object` can be a `list`, `set`, etc). Useful with list comprehensions. What's this do?

  ```
  all([x > 0 for x in some_list])
  ```

- `any(x)` is like `all(x)` but return true if *some* element of x is true.
- `dir()` returns a list of the names in the current scope. `dir(object or class or module)` returns the attributes (e.g. methods, classes defined in a module, instance variables) attached to the object.

Some useful built-in functions 2

- `int` converts strings to integers.
  `int(‘‘13’’) = 13`
  Warning: also truncates floats:
  `int(1.3) = 1`
- `max, min` - both `max(1,2,3,4)` and `max(some-list-or-set-or-other-iterable)` are accepted.
- `input(prompt:str)` prints `prompt` to the console, then waits for text input (submitted by hitting enter, as usual), and returns the text input as a string.
  Very useful for writing console-based utilities.
  You'll see this in your first assignment. :-)