CSC148 winter 2015

abstraction and idiom

week 2

Danny Heap
heap@cs.toronto.edu
BA4270 (behind elevators)
http://www.cdf.toronto.edu/~csc148h/winter/
416-978-5899

notes:
http://www.cdf.toronto.edu/~csc148h/winter/Notes/148Notes.pdf

January 12, 2015

Computer Science
UNIVERSITY OF TORONTO

Outline

str and repr

point and property...

abstract data types (ADTs)

implement an ADT with a class

idiomatic python

recursion

__str__

Class Point needs a __str__ method as its public face. See point.py

__repr__

Class Point needs a **__repr__** method for exact representation. See point.py

controlling attribute access. . .

So far, our definition of **Point** allows (possibly bumbling) client code to change **coord** after a point was created. We don't want that!

Use Python's built-in function **property** to intercept 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**

protect coord from being set twice

```
def _set_coord(self, coord):
    """ (Point, list-of-floats) -> NoneType

    Set coordinates for self
    """
    if '_coord' in dir(self):
    # has _coord already been set?
        raise Exception('Cannot reset coords')
    else: # if not already set, go ahead!
        self._coord = tuple(coord)
```

make sure coord's public face is a list

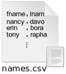
```
def _get_coord(self):
    """ (Point) -> list-of-float

    Return list of coordinates for self
    """
    return list(self._coord)
```

delegating with property

```
# Access to coord is delegated to property,
# so _get_coord and _set_coord
# are called instead
coord = property(_get_coord, _set_coord, None, None)
```

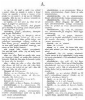
common ADTs

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

- 

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 (ignore the dire warnings...)

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

example: add (squares of) first 10 natural numbers

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

- 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 definition of `sum_list` — don't look at next slide yet!

hey! don't peek!

```
def sum_list(L):
    """ (list) -> float

    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 the elements of list...
               # if x is a sublist, sum_list(x)
               [sum_list(x) if isinstance(x, list)
                            else x # if not list, then number
               for x in L])
```

tracing recursion

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.

sample solutions

- trace `sum_list([1, 2, 3])`. Remember how the built-in `sum` works.
  Solution: `sum([1, 2, 3])` = 6
- trace `sum_list([1, [2, 3], 4, [5, 6]])`. Immediately replace calls you've already traced (or traced something equivalent) by their value
  Solution: `sum([1, 5, 4, 11])` = 21. We already knew what `sum_list` does with a flat list like [2,3] or `[5, 6]`
- trace `sum_list([1, [2, [3, 4], 5], 6 [7, 8]])`. Immediately replace calls you've already traced by their value.
  Solution: `sum([1, 14, 6, 15]) = 36`. We already know what `sum_list` does with nested lists like `[2, [3, 4], 5]`