CSC148 winter 2016

documentation, idiom, abstraction
week 3

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 31, 2016

Outline

avoid duplicating documentation

don't maintain documentation in two places, e.g. superclass and subclass, unless there's no other choice:

- inherited methods, attributes — no need to document again
- extended methods — document that they are extended and how
- overridden methods, attributes — document that they are overridden and how

see Shape and Square

Pycharm type hinting, redux

type hinting is new in the Python world, and to get the benefit of Pycharm's inspector, some fussing may be needed...

@type doesn't play well with text describing an attribute, so I have switched to @param...

special methods for Shape

Class Shape needs __**str**__ and __**eq**__, and so do all its subclasses.

Although we could override this in each subclass, a bit of research shows another way.

new lists from old

suppose **L** is a list of the first hundred natural numbers:

```
L = list(range(100))
```

if I want a new list with the squares of all the elements of **L** I *could*

```
new_list = []
for x in L:
    new_list.append(x * x)
```

or I could use the equivalent **list comprehension**

```
new_list = [x * x for x in L]
```

filtering with [...]

I can make sure my new list only uses specific elements of the old list...

```
L = ["one", "two", "three", "four", "five", "six"]
```

by adding a condition...

```
new_list = [s * 3
            for s in L
            if s <= "one"]
```

notice that a comprehension can span several lines, if that makes it easier to understand

general comprehension pattern

[expression **for** name in iterable **if** condition]

Python expressions evaluate to values, **name** refers to each element of **iterable** (list, tuple, dictionary, ...) in turn, and a **condition** evaluates to either **True** or **False**

see Code like Pythonista

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

try the python visualizer

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 added on to the top of the stack, items may only be removed from the top of the stack. It's a mistake to try to remove an item from an empty stack, so we need to know if it is empty. We can tell how big a stack 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 add and remove from position 0
- Use a python dictionary with integer keys 0, 1, . . . , keeping track of the last index used, and which have been removed

bag ADT

Here's a description of a **sack**, which has similar features to a stack:

> *A sack contains items of various sorts. New items are added on to a random place in the sack, so the order items are removed from the sack is completely unpredictable. It's a mistake to try to remove an item from an empty sack, so we need to know if it is empty. We can tell how big a sack 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

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

chosing test cases

since you can't test every input, try to think of **representative** cases:

- smallest argument(s): 0, empty list or string, ...
- boundary case: moving from 0 to 1, empty to non-empty, ...
- "typical" case

isolate units

- test classes separately
- test (related) methods separately

why?

generalize stack, sack as Container

stacks and sacks can have different implementations: using python lists, dictionaries, ... so it doesn't make sense to have the implementation in a superclass. However, it is nice to have a common API between the two, so we can write client code that works with any stack, sack, or other... Containers

```
# suppose L is list[Container]

for c in L:
    for i in range(1000):
        c.add(i)
    while not c.is_empty():
        print(c.remove())
```

... so we'll make Stack, Sack subclasses of Container!