CSC 148 Winter 2017

Week 3

More inheritance aspects, documentation, abstraction

Bogdan Simion

bogdan@cs.toronto.edu

http://www.cs.toronto.edu/~bogdan

University of Toronto, Department of Computer Science

Outline

- More inheritance aspects
  - Documentation, Special Methods for Inheritance
- Abstract Data Types (ADT)
- Implement ADTs with classes, inheritance

Remember: Inherit, override, or extend?

- Subclasses use three approaches to recycling the code from their superclass, using the same name
  - 1. Methods and attributes that are used as-is from the superclass are inherited – example?
  - 2. Methods and attributes that replace what's in the superclass are overriden – example?
  - 3. Methods that add to what is in the superclass are extended – example?

Avoid Duplicate 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.py and square.py

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]
```

- *expression* evaluates to a value
- *name* refers to each element in the iterable (list, tuple, dict, ...)
- *condition* (optional) evaluates to either True or False

- See [Code like Pythonista]

Exercises

- Construct lists in Python, equivalent to these:
  - D = {2*a | a ∈ (-1; 9] }
  - S = {b^2 | b ∈ [0; 11] }
  - L = {len(c) | c ∈ {“one”, “two”, “three”} }

- Given a list of measurements in inches, convert them to centimetres.
  - I = [23, 45, 17, 14, 59]
  - CM = ?

Pycharm type hinting, redux

- Type hinting is new in the Python world, and to get to 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 ...

Next up ..

- Abstract data types

- But first, reminder: avoid duplicate documentation!
  - 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

Abstraction

- View objects as entities that can store data and operations, useful to solve problems
- Focus on semantics
  - Hides the gory details from the user
  - Freedom to design or update algorithms
  - Independent of programming language
- Examples?

Abstract Data Types (ADTs)

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

Sequences of items; can be added, removed, accessed by position, etc.

Specialized collection of items where we only have access to most recently added item

Collection of items accessed by their associated keys

Stack ADT

- What does it store?
- What are the operations?

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.

- Class design – use what we learnt!
- Be flexible about alternate names and designs for the same class.

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.*

- Class design – use what we learnt!
- Be flexible about alternate names and designs for the same class.

Stack ADT

```
class Stack:

    def __init__(self):
        """ Create new empty Stack self """
        pass

    def add(self, obj):
        """ Add object to top of Stack self """
        pass

    def remove(self):
        """ Remove and return top element of self """
        pass

    def is_empty (self):
        """ Return whether Stack self is empty """
        pass
```

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

Sack (bag) ADT

- Definition:

  A sack contains items of various sorts. New items are added in a random place in the sack, so the order in which 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.

- Once again – use what we learnt!

Sack (bag) ADT

- Definition:

  A sack contains items of various sorts. New items are added in a random place in the sack, so the order in which 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.

- We've identified the key things in the definition

Sack ADT

```
class Sack:

    def __init__(self):
        """ Create new empty Sack self """
        pass

    def add(self, obj):
        """ Add object randomly to Sack self """
        pass

    def remove(self):
        """ Remove and return random element of self """
        pass

    def is_empty (self):
        """ Return whether Sack self is empty """
        pass
```

Notice something?

```python
class Stack:

  def __init__(self):
    """Create new empty Stack"""
    pass

  def add(self, obj):
    """Add object to top of self"""
    pass

  def remove(self):
    """Remove+return top item"""
    pass

  def is_empty (self):
    """Is self empty?"""
    pass
```

```python
class Sack:

  def __init__(self):
    """Create new empty Sack"""
    pass

  def add(self, obj):
    """Add object randomly to self"""
    pass

  def remove(self):
    """Remove+return random item"""
    pass

  def is_empty (self):
    """Is self empty?"""
    pass
```

Very similar structures! Same operations, but different implementation

Generalize Stack, Sack as Container

- Stacks and sacks are very similar in nature

  => It doesn't make sense to have duplicate code

  => Abstract them to a more general *Container class*

- Stacks and sacks can have different implementations: using python lists, dictionaries, etc.

  => It doesn't make sense to have the implementation in a superclass. Why?

- It's nice to have a common API between the two, so we can write client code that works with any stack, sack, or other similar .. Containers ("things" that store data items).

Generalize Stack, Sack as Container

- Suppose L is a list of Containers

```
for c in L:

   for i in range(1000):
      c.add(i)

   while not c.is_empty():
      print(c.remove())
```

- Should work regardless of whether c is a Sack, Stack, etc.
- So we'll make Stack, Sack subclasses of Container!

Container class

- What are the interface commonalities between Stack/Sack?

    s.__init__()

    s.__str__()

    s.__eq__()

    s.add()

    s.remove()

    s.is_empty()

- So, we can abstract the commonalities in a superclass called Container

Container class

- *Important decision:* which methods should be implemented and which ones should be forced to be implemented by subclasses?

    s.__init__()

    s.__str__()

    s.__eq__()

    s.add()

    s.remove()

    s.is_empty()

- Thoughts?

Stack and Sack implementation

- Finally, subclass Stack and Sack from Container

- Remember:
  - Stack and Sack inherit all the Container attributes and methods
  - Stack and Sack may override what's in Container
  - Stack and Sack may extend the superclass Container

- Which methods do we want to override or extend?

Exceptions

- Generic Exception class
- Other predefined exceptions
  - e.g., NotImplementedError, IndexError, etc.
- May define our own custom exceptions
  - Can subclass Exception to define new custom exceptions
- Example ..

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
  - Compose tests before and during implementation

Python unittest

- A framework to setup test cases, run them independently from one another, document them, and reuse them when needed
- Unit == smallest testable part of a program
- Extending unittest.TestCase – same as extending any other class
  - e.g., class myUnbreakableAwesomeSystem(unittest.TestCase): ...
- Override special methods:
  - setUp() and tearDown()
- Follow conventions:
  - test followed by method name: e.g., testIsEmpty, testAdd, etc.
  - assert statement => checks expected outcome
  - see also: assertEquals, assertTrue, assertFalse, etc. (equivalent to assert)
- Remember: tests are independent!

Choosing test cases

- Since you can't test every input, try to think of representative cases:
  - Smallest argument(s): 0, empty list of string, ...
  - Boundary case: moving from 0 to 1, empty to non-empty, ...
  - "Typical" case