GÖTEBORGS UNIVERSITET

Språk-BANKEN

CLT

# Lecture 7: Classes and objects

*Introduction to programming (LT2111)*

Markus Forsberg
Språkbanken
University of Gothenburg

2011-10-18

## The exam

- Exam date: Friday, October 28, 13.30-18.30, Viktoriagatan 30 (V30).
- I will visit the exam an hour into the exam to answer questions about the questions (but not about solutions).

## Classes and objects

- Classes are user-defined types.
- A `class` definition gives a *type specification*.
- An *object* of a class is called an *instance* of that class.
- A class consists of *attributes* (data) and *methods* (functions).
- We have already meet many classes through NLTK, e.g., `Text` and `Synset`.

## Classes and objects

- Programming with a heavy use of classes and objects are called *object-oriented programming*.
- Object-oriented programming is a way of conceptualize problem (if you know the type of a thing, you know what you can do with it).
- Object-oriented programming provides *data encapsulation* (*information hiding*).
- And more, it provides a convenient way for *code reuse* (through inheritance, more later).

GÖTEBORGS UNIVERSITET

Språkbanken

CLT

## Class

- Basic syntax:

```
class NAME:

    def __init__(self,INITPARAMS):
        INITIALIZATION
```

- `self` refers to the object itself.

# Example: Elevator

```
class Elevator:

    def __init__(self,top_floor, bottom_floor=0,
                                 initial_floor=0):
        self.top_floor = top_floor
        self.bottom_floor = bottom_floor
        self.current_floor = initial_floor

    def goto_floor(self, floor):
        if (floor >= self.bottom_floor and
            floor <= self.top_floor):
            self.current_floor = floor
        else:
            raise ValueError, "Floor does not exist."

    def get_floor(self):
        return self.current_floor

>>> e = Elevator(initial_floor=0,
                 bottom_floor=0,top_floor=10)
>>> e.goto_floor(7)
>>> e.get_floor()
7
```

## Note on default values

- Be careful with mutable default values.

```
class C:
    def __init__(self, lst=[]):
        self.lst = lst

    def add(self,e):
        self.lst.append(e)

    def items(self):
        return self.lst

>>> a = C()
>>> b = C()
>>> a.add(5)
>>> b.items()
[5]
```

## Mutable default values: fix

```
class C:
    def __init__(self, lst=None):
        if lst == None:
            self.lst = []
        else:
            self.lst = lst

    def add(self,e):
        self.lst.append(e)

    def items(self):
        return self.lst

>>> a = C()
>>> b = C()
>>> a.add(5)
>>> b.items()
[]
```

GÖTEBORGS UNIVERSITET

Språkbanken

CLT

## Objects are mutable

- Objects are mutable, i.e., we have to be careful to avoid unintentional *aliasing*.
- We can use module `copy` to copy any object (both shallow and deep).

```
>>> a = C()
>>> b = a
>>> a is b
True
>>> import copy
>>> b = copy.copy(a) # shallow
>>> a is b
False
>>> b = copy.deepcopy(a) # deep
```

GÖTEBORGS UNIVERSITET

Språk-BANKEN

CLT

## Public and private

- Methods and attributes are by default *public*, i.e., available with the dot notation.
- Methods and attributes are *private* if they are internal to the class, i.e., help functions.
- Private names are written, by convention, with an initial underscore (e.g., `_internal`).
- Private names signals that it should not be used, but nothing stops you from doing it anyway.

## Inheritance

- *Inheritance* allows us to define a class that is a modified version of another class.
- We say that a *child* class inherents from a *parent* class.
- Inheritance gives us *code reuse*.
- Inheritance also gives us explicit relations between classes.

# Inheritance: silly example

```
class Bird:

    def sound(self):
        print 'quack'

    def can_fly(self):
        return True

class Penguin(Bird):

    def can_fly(self):
        return False

>>> any_bird = Bird()
>>> penguin = Penguin()
>>> any_bird.sound()
quack
>>> penguin.sound()
quack
>>> any_bird.can_fly()
True
>>> penguin.can_fly()
False
```

## String representation of an object

```
class Bird:

    def sound(self):
        print 'quack'

    def can_fly(self):
        return True

    def __str__(self):
        return '<birdy>'

>>> bird = Bird()
>>> print bird
<birdy>
```

## Length of an object

```
class Penguin(Bird):

    def can_fly(self):
        return False

    def __len__(self):
        return 1

>>> penguin = Penguin()
>>> len(penguin)
1
```

# Problem: class Text

- Define a class `Text` that is instantiated with a list of tokens.
- Add a method `concordance` that given a word, gives all its occurrences in the text with a context (4 tokens to the left and right).
- Add a method `count` that given a word, returns the number of times it occurs in the text.
- Add a method `vocab` that returns the vocabulary of the text (no duplication).
- Add indexing, length, and a string representation to `Text`.

## Problem: class Stack

- Define a class `Stack` that implements a stack data structure.
- A stack data structure supports the following methods: `is_empty` (checks if the stack is empty), `push(element)` (pushes the element on the top of the stack), `pop()` (removes and returns the top element of the stack).

GÖTEBORGS UNIVERSITET

Språkbanken

CLT

## Problem: class Deck

- Define a class `Card` to represent a playing card.
- Define a class `Deck` representing a deck of cards, and define a method `draw` for `Deck` that returns and removes a random card in Deck.