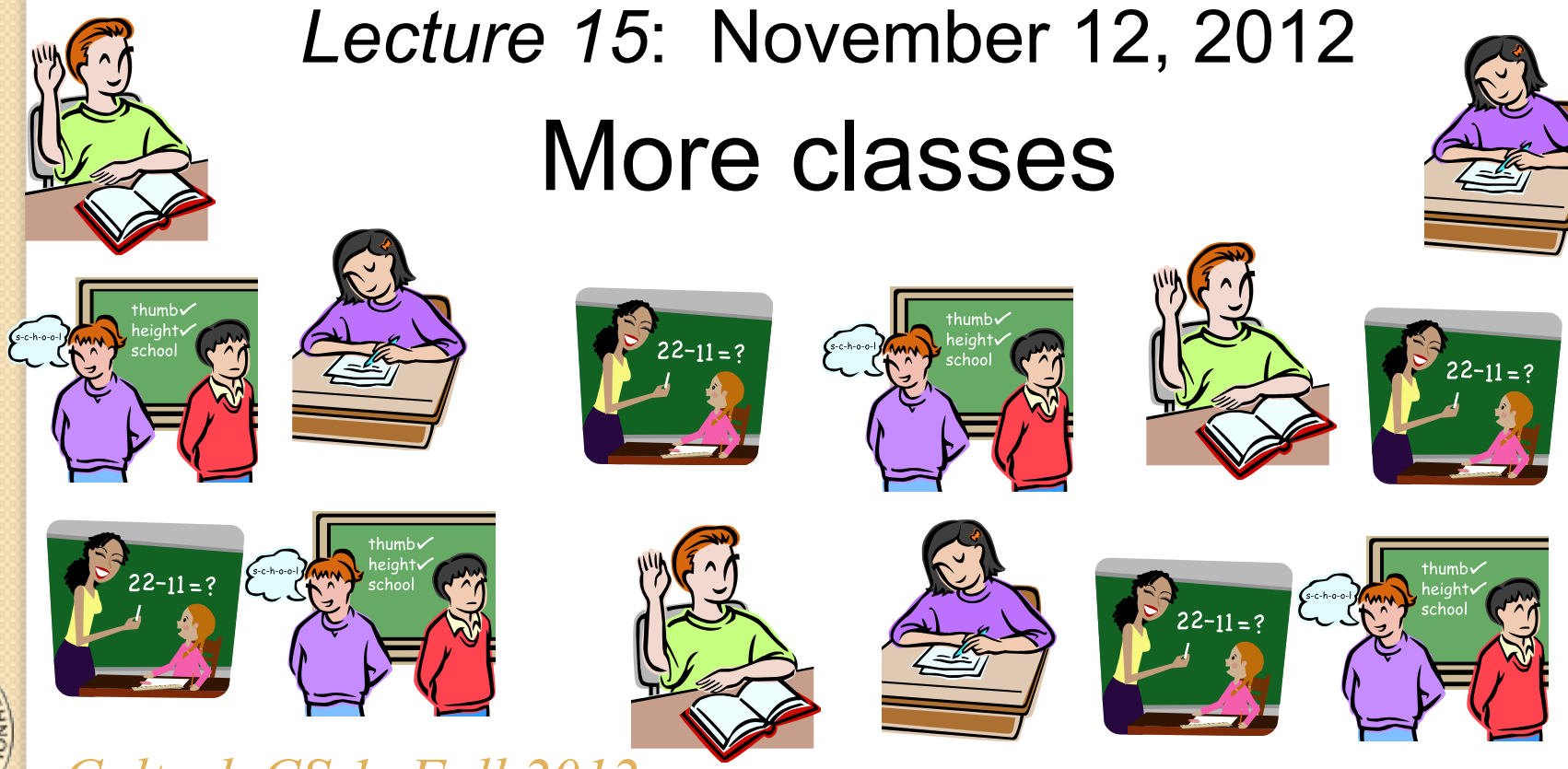
CS 1
Introduction to Computer Programming

Lecture 15: November 12, 2012

More classes



Last time

- Classes and object-oriented programming, part 1
- The `class` statement
  - creating your own objects
- Constructors
- Defining new methods

Today

- We'll use classes to develop a realistic example

Classes and objects

- Recall from last time:
- An *object* is a data value that contains
  - *attributes* (data items)
  - *methods* (functions that act on the object)
- A *class* is a description of what an object is
  - contains the definition of all methods for an object
- Every object is an *instance* of some class

Terminology

- Technically, methods are also attributes; they are attributes which happen to be functions
- We will usually refer to something as an "attribute" to mean a non-function attribute (data attribute) as opposed to a method
- Data attributes are also often called *fields* for short

Classes and objects

- Every object of the same class contains the *exact same* methods
- Every object of the same class contains the same (data) attribute *names*, but the *values* for these attributes are generally different
- We can think of an object as a "bag of data attributes with methods"

Classes and objects

- Every class has a *constructor method* (or *constructor* for short) called `__init__`
- The constructor is called when an object is created
- The data attributes of an object are normally initialized in the constructor
  - though the values can be changed later

Methods

- Method definitions look like regular Python function definitions
  - except they are located inside a `class` statement
- Method definitions have a first argument called `self` which represents the object being acted on
- Inside a method, can access the attributes of the current object through the `self` argument

Methods

- When methods are called, the method call is missing the first (**`self`**) argument
  - that argument goes before the dot in the dot syntax
- So if you have a class **`Thingy`** with an instance **`t`** and a method called **`getValue`**:

```
t.getValue()
```

- is the same as:

```
Thingy.getValue(t)   # t → self
```

Graphics example

- There's more to know about classes and objects, but that's enough for now
- Let's recap the graphics example we discussed last time and extend it

Graphics example

- We want to create objects to represent things we can draw on canvases
  - squares, circles
- We'll start with a `Square` object

Square object

```
class Square:
    '''Objects of this class represent a square
    on a Tkinter canvas.'''
    def __init__(self, canvas,
                 center, size, color):
        ...
    <other methods>
```

Square constructor

```
def __init__(self, canvas, center, size, color):
    (x, y) = center
    x1 = x - size/2  # upper left
    y1 = y - size/2  # coordinates
    x2 = x + size/2  # lower right
    y2 = y + size/2  # coordinates
    self.handle = \
      canvas.create_rectangle(x1, y1, x2, y2,
        fill=color, outline=color)
```

Square constructor

- Creating squares:

```
s1 = Square(canvas, (100, 100), 50,  'red')
s2 = Square(canvas, (230, 110), 100, 'blue')
s3 = Square(canvas, (50, 240),  75,  'green')
```

- This puts three squares on a canvas
- Each square knows what its handle is
- However, we need to add some more attributes to the object

Square constructor

```
def __init__(self, canvas, center, size, color):
    (x, y) = center
    x1 = x - size/2  # upper left
    y1 = y - size/2  # coordinates
    x2 = x + size/2  # lower right
    y2 = y + size/2  # coordinates
    self.handle = \
      canvas.create_rectangle(x1, y1, x2, y2,
        fill=color, outline=color)
    self.canvas = canvas
    self.center = center
    self.size   = size
    self.color  = color
```

Square constructor

```
def __init__(self, canvas, center, size, color):
    # <some code omitted>
    self.canvas = canvas
    self.center = center
    self.size   = size
    self.color  = color
```

- What we're doing here is saving the arguments of the constructor as fields (attributes) of the **Square** objects we create

`Square` constructor

```
def __init__(self, canvas, center, size, color):
    # <some code omitted>
    self.canvas = canvas
    self.center = center
    self.size   = size
    self.color  = color
```

- This allows us to do two things:
  - We can look up/change these values in any `Square` object
  - We can use these values inside `Square` methods

Creating Squares again

```
>>> s1 = Square(canvas, (100, 100), 50,  'red')
```

- With the new fields we can look up values in the square **s1** at any time:

```
>>> s1.center
(100, 100)
>>> s1.size
50
>>> s1.color
'red'
```

Square constructor

```
def __init__(self, canvas, center, size, color):
    # <some code omitted>
    self.canvas = canvas
    self.center = center
    self.size   = size
    self.color  = color
```

- Without these lines, the values of the arguments to the constructor would be gone after the constructor executes
- We store them inside the object so that we can use them in the object's methods

`Square` module

- Let's build up our **`Square`** object method-by-method to make our **`Square`**s more powerful
- We'll put this code in a module called **`Square`** (*i.e.* in a file called **`Square.py`**)
- At the bottom of the file we'll have code to create the root object, the canvas, and one or more **`Square`**s

Square module

```
from Tkinter import *

class Square:
    # Constructor omitted

if __name__ == '__main__':
    root = Tk()
    root.geometry('800x800')
    canvas = Canvas(root, height=800, width=800)
    canvas.pack()
    s1 = Square(canvas, (100, 100), 50, 'red')
```

python -i

- Our `Square` class only has a constructor so far
- We can run our program and test it interactively by typing

```
% python -i Square.py
```

- at the terminal prompt (`%`)
- The `-i` tells Python that you want to run the module *interactively*
- Just like `python Square.py`, except that after it's done you go back into the Python shell

Square module

```
% python -i Square.py
```

- The program runs, displays a canvas with a red square, and you get the prompt:

```
>>>
```

- At this point, you can play around with the objects you've created:

```
>>> s1
<__main__.Square instance at 0x76ddf0>
```

`Square` module

- Let's look at the `Square` object's fields:

```
>>> s1.center
(100, 100)
>>> s1.size
50
>>> s1.color
'red'
```

- The `Square` object stores these values, so every method of Square will be able to access them through the `self` argument

Changing color

- What if we want to change the color?

```
>>> s1.color = 'blue'
>>> s1.color
'blue'
```

- We've successfully changed the color *field* of the square
- But the square has not changed color!
- Why?

Changing color

- To change the color of an object on a canvas, we have to call a canvas method called *itemconfig*:

```
>>> canvas.itemconfig(s1.handle,
        fill='blue', outline='blue')
```

- This is tedious:
  - have to get the handle from square **`s1`**
  - have to separately specify fill and outline color
- Let's write a method on **`Square`**s to do this

setColor method

```
class Square:
    # leave out previously-defined code

    def setColor(self, color):
        '''Changes this object's color to
           'color'.'''
        self.canvas.itemconfig(self.handle,
            fill=color, outline=color)
        self.color = color
```

- Change the color of the canvas object

`setColor` method

```
class Square:
    # leave out previously-defined code

    def setColor(self, color):
        '''Changes this object's color to
           'color'.'''
        self.canvas.itemconfig(self.handle,
            fill=color, outline=color)
        self.color = color
```

- Store the new color into the **`Square`** object

`setColor` method

- Now we can do this:

```
>>> s1.setColor('blue')
```

- and the color of the **`Square`** object **`s1`** on the canvas turns to blue!
- And also:

```
>>> s1.color
'blue'
```

- So the new color is recorded in the **`Square`** object **`s1`**

Changing size

- What if we want to change the size?

```
>>> s1.size
50
>>> s1.size = 100
>>> s1.size
100
```

- By now, you shouldn't be surprised that this doesn't change the size of the square on the canvas

Changing size

- To change the size of an object on a canvas, we have to call a canvas method called **`coords`**:

```
>>> canvas.coords(s1.handle,
                  x1, y1, x2, y2)
```

- **`(x1, y1)`** are the (new) coordinates of the upper-left-hand corner
- **`(x2, y2)`** are the (new) coordinates of the lower-right-hand corner

Changing size

- We need to compute `x1`, `y1`, `x2`, `y2` given the square's `center` coordinates and the new `size` value
- We saw how to do this in the constructor (the `__init__` method):

```
(x, y) = center
x1 = x - size/2
y1 = y - size/2
x2 = x + size/2
y2 = y + size/2
```

`setSize` method

- Let's write a `setSize` method to change the size of a `Square`:

```
def setSize(self, size):
    '''Change the size of this square.'''
    (x, y) = self.center
    x1 = x - size/2
    y1 = y - size/2
    x2 = x + size/2
    y2 = y + size/2
    self.canvas.coords(self.handle,
        x1, y1, x2, y2)
    self.size = size
```

`setSize` method

- Calling this method on square **`s1`**:

```
>>> s1.setSize(120)
```

- Now **`s1`** is **`120`** pixels on each side

`scale` method

- Let's also write a **`scale`** method to change the size of a **`Square`** by some scaling factor:

```
def scale(self, scale_factor):
    '''Scale the size of this square by a
    scaling factor.'''
    self.setSize(self.size * scale_factor)
```

- Now, if we do this to square **`s1`**:

```
>>> s1.scale(2.0)
```

- the square will grow to twice its original size

Interlude

- Computer history!

More methods!

- Let's add more methods!
- Want to be able to
  - move objects relative to their current position
  - move objects to some absolute position
  - lift objects to the top of the stacking order
  - delete objects

Moving squares

- Canvases have a `move` method which can move objects
- We would like to be able to move our squares
- Let's define a `move` method which calls the canvas `move` method
- Using the `move` method on squares will be much less tedious than using the `move` method on canvasses

Moving squares

```
def move(self, x, y):
    '''Move this square by (x, y).'''
    self.canvas.move(self.handle, x, y)
```

- This will work, but it's not enough
- The square needs to know its location
  - stored in the `center` field of the object
- We've changed the location
  - must also update `center`

Moving squares

```
def move(self, x, y):
    '''Move this square by (x, y).'''
    self.canvas.move(self.handle, x, y)
    (cx, cy) = self.center
    self.center = (cx + x, cy + y)
```

- This is the complete method

Moving squares

- Calling this method on square **`s1`**:

```
>>> s1.move(50, 150)
```

- and the square will move 50 pixels in the **`x`** (horizontal) direction and 150 pixels in the **`y`** (vertical) direction

Move to absolute location

- The `move` method will only move a square relative to its current location
- Might want to move a square to an absolute location on the canvas
  - *e.g.* to point `(350, 450)`
- Let's define a method to do that
  - We'll call it `moveTo`

Move to absolute location

- The trick: if we knew the difference between our current location and the target location, we could call the `move` method with those numbers (different in `x` position, difference in `y` position)
- *Larger point*: when you have some basic methods defined, can often define other methods in terms of them
  - need to write less code overall

Move to absolute location

```
def moveTo(self, x, y):
    '''Move this square to the location (x, y)
    on its canvas.'''
    ...
```

- Let's fill this in

Move to absolute location

```
def moveTo(self, x, y):
    '''Move this square to the location (x, y)
    on its canvas.'''
    (cx, cy) = self.center
    dx = x - cx
    dy = y - cy
    self.canvas.move(self.handle, dx, dy)
```

- This works, but...
- What's missing?

Move to absolute location

```
def moveTo(self, x, y):
    '''Move this square to the location (x, y)
    on its canvas.'''
    (cx, cy) = self.center
    dx = x - cx
    dy = y - cy
    self.canvas.move(self.handle, dx, dy)
    self.center = (x, y)
```

- Need to record the new **`(x, y)`** position in the **`center`** field of the square object

Move to absolute location

- Call this method:

```
>>> s1.moveTo(200, 200)
```

- and the square `s1` is now centered on the point `(200, 200)` of the canvas

D.R.Y. again

- We can simplify this code by calling the **`move`** method instead of **`self.canvas.move`**:

```
def moveTo(self, x, y):
    '''Move this square to the location (x, y)
    on its canvas.'''
    (cx, cy) = self.center
    dx = x - cx
    dy = y - cy
    self.move(dx, dy)
```

Change stacking order

- When two squares overlap, only one can be visible in the overlapping region
- The `Tkinter` canvas object keeps track of "what is above what"
- This is called the *stacking order* of the canvas
- Objects on top of the stacking order cannot be covered up by other objects

Change stacking order

- Every object on a canvas has a place in the stacking order
  - even objects that have no other objects overlapping with them
- Sometimes, want to manually lift an object to the top of the stacking order
- The canvas object defines a `lift` method to do this
- Let's define such a method for our squares

Change stacking order

```
def lift(self):
    '''Lift this square to the top of the canvas
    stacking order.'''
    self.canvas.lift(self.handle)
```

- That's all it takes!
- Now we can do (for square **`s1`**):

```
>>> s1.lift()
```

- and **`s1`** will pop up above any other squares that overlap it

Deleting squares

- Let's finish with an easy method: `delete`
- This method will remove an object from a canvas
- BUT:
  - It will not remove it from the Python program!
  - (See how to do that later)

Deleting squares

- Here's the method:

```
def delete(self):
    '''Remove the square from the canvas.'''
    self.canvas.delete(self.handle)
```

- That's it!
- Now, when this method is called, the square disappears from the canvas

Deleting squares

- Example:

```
>>> s1.delete()
[square disappears from canvas]
>>> s1
<__main__.Square instance at 0x102e030>
```

- The **`Square`** *object* is still there!
- How do we make the *object* **`s1`** go away?

Deleting squares

- To delete **`s1`** entirely we need to use the **`del`** operator

```
>>> del s1
>>> s1
NameError: name 's1' is not defined
```

- It would be nice if we could combine these two kinds of "deletions" into one method

Deleting squares

- Python allows objects to define a special method called `__del__`
- It is called whenever `del` is applied to an object (*i.e.* `del s1` for the square `s1`)
- It allows us to do additional clean-up before the object is deleted
- In the object-oriented programming world, this kind of method is called a *finalizer*
- Let's define it

Deleting squares

```
def __del__(self):
    '''Delete this square.'''
    self.delete()
```

- That's it!
- Some interesting points about this method:
  - We are calling one method (**`delete`**) from inside another method (**`__del__`**)
  - This method doesn't *replace* what **`del`** used to do; it gets run just *before* **`del`** destroys the object

Next time

- New topic: Exception handling
  - How to recover from errors in your program in a controllable way