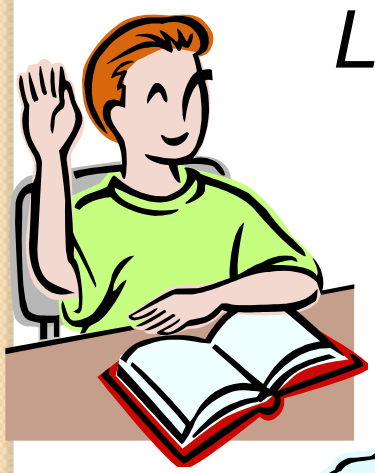
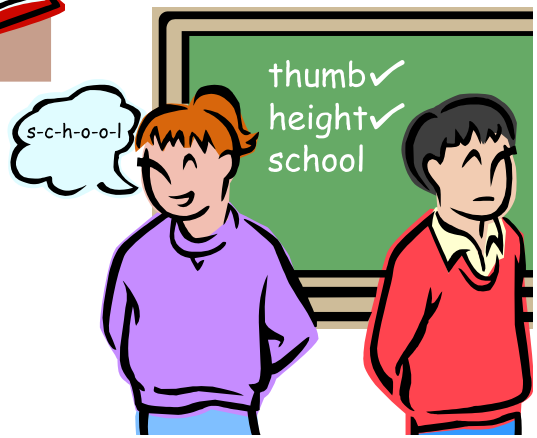
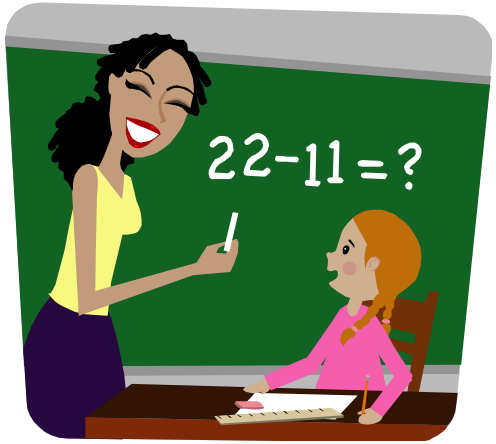
CS 1
Introduction to Computer Programming

Lecture 14: November 4, 2015

Classes







Last time

- Event handling in graphical programs
  - Key events
  - Mouse button events
  - Callback functions
- Global variables

Today

- Classes and object-oriented programming
  - The `class` statement
  - Creating your own objects
  - Constructors
  - Defining new methods

Objects

- Since lecture 2, have been working with *objects*
  - strings
  - lists
  - dictionaries
  - tuples
  - files
  - canvases in **`Tkinter`**
- All the kinds of objects we've been working with have been built in to Python

What is an object?

- We have used an operational definition of what an "object" is
- It's some kind of thing that
  - has some kind of internal data (called its *state*)
  - is something we can call *methods* on
- A method is like a function, but it acts on an object (and may take other arguments)
- Methods use the *"dot syntax"* :

```
object.method(arg1, arg2, ...)
```

What is an object?

- Example: the **`append`** method on lists:

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.append(42)
```

- Here, **`lst`** is the *object* (a list)
- **`append`** is the name of the *method*
- **`42`** is the *argument* to the method

Methods vs. functions

- We might ask: why not just have **`append`** be a regular Python function?
- Then we could write this:

```
>>> lst = [1, 2, 3, 4, 5]
>>> append(lst, 42)
```

- What's wrong with this?

Methods vs. functions

```
>>> lst = [1, 2, 3, 4, 5]
>>> append(lst, 42)
```

- *Problem 1:*
- If **`append`** was a function, it would have to change the list argument passed to it
- Usually, we don't like to do this
  - functions are easier to manage if we don't change the arguments to the function

Methods vs. functions

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.append(42)
```

- Having **`append`** as a method on a list object suggests that it's OK for **`append`** to change the list that it's acting on
- You're "appending to *this* list (**`lst`**)"
- **`42`** is "the thing you're appending"

Methods vs. functions

```
>>> lst = [1, 2, 3, 4, 5]
>>> append(lst, 42)
```

- *Problem 2:*
- **`append`** as a function can only work on *one* kind of data
- We might have some other thing (say, a **`Thingy`**) that has a different way of appending
- Would need to use a different function with a different name

Methods vs. functions

```
>>> t = make_a_Thingy()
>>> append_Thingy(t, 42)
```

- It's annoying to have to come up with a completely new name for a similar kind of action (`append_Thingy` instead of just `append`)
- If you have a lot of different types of data with similar behaviors, this can make programs cluttered and hard to read

Methods vs. functions

```
>>> lst = [1, 2, 3, 4, 5]
>>> append(lst, 42)
```

- *Problem 3:*
- This will only work if the internal state of the objects (lists here) can be changed by any function
- May want to allow only *particular* functions to change the internal state
  - *e.g.* only methods can change internal state

Methods vs. functions

- So methods have some (conceptual, readability, convenience, safety) advantages over functions in some cases
- Functions have one big advantage so far:
  - We know how to define them!
- Methods are inextricably tied to objects
  - so in order to talk about defining methods, we have to talk about defining new kinds of objects

Objects and internal data

- Objects can store internal data
  - (We saw this with `Tkinter` event objects and their `x` and `y` attributes)
- Sometimes we would like to create special kinds of objects to store particular kinds of data
- And define new methods to interact with that data in object-specific ways
- Creating new object types will allow us to do that

Running example

- We have discussed the **`Tkinter`** graphics package a lot in class
- **`Tkinter`** is partially object-oriented
  - Some things in **`Tkinter`** (root windows, canvases) are objects and have methods
  - Some things (shapes on canvases) are not objects in the Python sense
  - so must be handled indirectly using "handles" and methods on canvases

Running example

- This distinction in `Tkinter` between
  - true objects, and
  - things that seem like they should be objects but aren't
- is really ugly
- Not the only source of ugliness in `Tkinter`!
- Event handling forced us to rely heavily on global variables to do a lot of things
- Global variables are dangerous
  - would like to get rid of them if possible

The plan

- Good news, everyone!
- Defining our own objects will allow us to fix *both* of these problems
  - and will also allow us to do other interesting/powerful things
- We will make object versions of squares and circles that we have been drawing on `Tkinter` canvases
- First, though, we'll try a very simple example
- But first...

Interlude

- TV clip!
  - The Meaning of Everything!

Classes

- Objects in Python are all *instances* of some *class*
- A *class* describes what an object is
- A *class definition* includes the definition of all the methods that objects of that class can do
- A class is also a *type* (a kind of data)
- Instances of a class can contain internal data
  - Examples: a list has elements, a dictionary has key/value pairs, etc.

The `class` statement

- Classes in Python are defined using the **`class`** statement
- A **`class`** statement is a complete description of how instances of that class (objects of that class) behave
- Let's look at a template for a **`class`** statement now

The `class` statement

```
class <name of class>:
    '''<docstring>'''
    def __init__(self, arg1, ...):
        ...
    def <method1>(self, arg1, ...):
        ...
    def <method2>(self, arg1, ...):
        ...
```

- Let's fill in the blanks for a trivial class

The `class` statement

```
class Thingy:
    '''Instances of this class store a
    single value.'''
    def __init__(self, value):
        self.value = value
    def getValue(self):
        return self.value
    def setValue(self, newValue):
        self.value = newValue
```

The `class` statement

- There's a lot going on in that last slide!
- Let's cut it down and talk about the pieces separately

The `class` statement

```
class Thingy:
    '''Instances of this class store a
    single value.'''

    def __init__(self, value):
        self.value = value
```

The `class` statement

```
class Thingy:
    '''Instances of this class store a
    single value.'''

    def __init__(self, value):
        self.value = value
```

- Classes are introduced using the **`class`** keyword

The `class` statement

```
class Thingy:
    '''Instances of this class store a
    single value.'''

    def __init__(self, value):
        self.value = value
```

- The name of the class follows the `class` keyword, followed by a colon (`:`)
- So this defines class `Thingy`

The `class` statement

```
class Thingy:
    '''Instances of this class store a
    single value.'''

    def __init__(self, value):
        self.value = value
```

- Everything inside the `class` statement is indented (as with most Python statements)

The `class` statement

```
class Thingy:
    '''Instances of this class store a
single value.'''

    def __init__(self, value):
        self.value = value
```

- The class docstring is the first thing inside the `class` statement
- Can be left out, but much better to have it

The `class` statement

```
class Thingy:
    '''Instances of this class store a
    single value.'''

    def __init__(self, value):
        self.value = value
```

- After the docstring come one or more *method definitions*

Method definitions

```
def __init__(self, value):
    self.value = value
```

- Python method definitions use **`def`** as their keyword like regular functions
- For the most part, these are the same as regular function definitions
  - but they behave differently when called

self

```
def __init__(self, value):
    self.value = value
```

- The first argument to a Python method represents *the object being acted on*
- By convention, this is called **`self`**
  - (though it doesn't have to be)

self

```
def __init__(self, value):
    self.value = value
```

- The **self** object is kind of like a dictionary
- It uses the dot syntax instead of dictionary syntax to add/modify values associated with names (or to get values associated with names)
- These names are called the object *attributes*

self

```
def __init__(self, value):
    self.value = value
```

- This method makes **value** an attribute of the object represented by **self**
- The attribute **self.value** is assigned to be the same as the method argument called **value**
- So this method just stores the **value** argument as **self.value**

__init__

```
def __init__(self, value):
    self.value = value
```

- This method has a funny name: `__init__`
- Recall that names in Python that have two initial underscores and two terminal underscores are "special"
- They have a special meaning to Python
  - *e.g.* `__name__` is the current module's name

Constructor

```
def __init__(self, value):
    self.value = value
```

- The **`__init__`** method is what is called the *constructor method* for the class (or just the *constructor* for short)
- This method is called when an instance of the class is being created
- It is responsible for **`init`**ializing the object in whatever way is required

Constructor

```
def __init__(self, value):
    self.value = value
```

- The `__init__` method returns the object that has been constructed (even though there is no `return` statement)
- It's as if it were written:

```
def __init__(self, value):
    self.value = value
    return self  # not actually necessary
```

Constructor

- If you actually write it like this:

```
def __init__(self, value):
    self.value = value
    return self
```

- It will result in an error once the constructor is called
- Constructors must not *explicitly* return anything!

Constructor

```
def __init__(self, value):
    self.value = value
```

- The constructor is normally where the attributes of the object are defined and given their initial values
- Here, the object will have one attribute: **`self.value`**

More methods

```
def getValue(self):
    return self.value
def setValue(self, newValue):
    self.value = newValue
```

- This class contains two more method definitions
- They look like regular function definitions
- Both have **`self`** as their first argument
  - meaning "this object"

More methods

```
def getValue(self):
    return self.value
def setValue(self, newValue):
    self.value = newValue
```

- The **`getValue`** method takes the object as its argument and returns the **`value`** attribute of the object

More methods

```
def getValue(self):
    return self.value
def setValue(self, newValue):
    self.value = newValue
```

- The **`setValue`** method takes the object and a new value as arguments and changes the **`value`** attribute of the object to **`newValue`**

Using objects

- That's the end of the definition of class **`Thingy`**
- Now we need to know how to use it to create objects and call their methods

Creating objects

- To create a new **`Thingy`** object, just use **`Thingy`** as if it were a function name:

```
>>> t = Thingy(42)
```

Creating objects

```
>>> t = Thingy(42)
```

- What does this mean?
- `Thingy` is a class name, not a function!

Creating objects

```
>>> t = Thingy(42)
```

- When you use a class name as if it were a function, what you are doing is calling the `__init__` method of that class
- However, the `__init__` method took two arguments (`self` and `value`), and this only takes one (`42`) – what's going on?

Creating objects

```
>>> t = Thingy(42)
```

- Python translates this into something like

```
t = makeEmptyObject()
Thingy.__init__(t, 42)
```

- In other words, Python:
  - creates a new object with no attributes
  - passes it and the argument `42` to the `__init__` method of the class `Thingy`

Creating objects

```
>>> t = Thingy(42)
```

- The `__init__` method itself isn't responsible for creating the (initially empty) object
  - Python does that just before `__init__` is called
- Instead, `__init__` is responsible for creating and initializing the *attributes* of the object
  - and whatever other initialization might be necessary

Using objects

```
>>> t = Thingy(42)
```

- Now that we've created the object `t` (which is a `Thingy`), we can call its methods

```
>>> t.getValue()
42
>>> t.setValue(101)
>>> t.getValue()
101
```

Using objects

```
>>> t.getValue()
```

- Again, something weird is happening
- Recall the method definition:

```
def getValue(self):
    return self.value
```

- `getValue` was defined to take one argument
- We called it with no arguments
- Why does this work?

Using objects

```
def getValue(self):
    return self.value
```

- When we write

```
t.getValue()
```

- Python interprets this as if it were:

```
Thingy.getValue(t)
```

- So the object (`t` in this case) is always the first argument to every method

Using objects

```
def setValue(self, value):
    self.value = value
```

- When we write

```
t.setValue(101)
```

- Python interprets this as:

```
Thingy.setValue(t, 101)
```

- Same idea!

Methods

- So methods are really just functions with
  - a special syntax (dot syntax)
  - an object as the *implicit first argument*
- The first argument (usually called **`self`**) is the object before the dot in the method call
- **`t.getValue()`** means that the **`getValue`** method is called on the **`Thingy`** object **`t`**

Attributes

- Our `Thingy` object has one attribute: `value`
- We can directly access it if we want to:

```
>>> t.value
101
```

- We can also change its value:

```
>>> t.value = 999
>>> t.value
999
```

Attributes

- Directly accessing attributes usually isn't a good idea
- Directly changing attribute values is usually a *very* bad idea
- Attributes should be considered to be the *private* state of an object
  - "private" means "used only by the object's methods"
- There are ways to restrict access to attributes (will see later)

Graphics example

- We want to create objects to represent things we can draw on canvases
  - squares, circles
- We'll see that classes make this easy
- We'll start with a class for squares

`Square` class

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

- The **`Square`** constructor has arguments:
  - **`canvas`** (the **`Tkinter`** canvas it's drawn on)
  - **`center`** (a tuple of **`(x, y)`** coordinates)
  - **`size`** (integer)
  - **`color`** (a color string)

Square class

```
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` class

- With nothing but this, it's now easy to create squares (**`Square`** objects):

```
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
  - can use this in other methods
- 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 `Square`s 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

Next time

- Continue with the example
  - Add more methods to the `Square` class
  - Create a `Circle` class