CS 1
Introduction to Computer Programming

Lecture 20: November 25, 2012

Inheritance

Last time

- Regular expressions

Today

- Class inheritance
  - subclass, superclass
  - polymorphism
  - demos
- Exceptions and classes
  - Exception objects
  - the `raise` statement
  - the `pass` statement
  - the `Exception` base class

The `Square` class

- Previously we defined a **`Square`** class that represented squares on a **`Tkinter`** canvas object
- Using **`Square`**s and methods made it easy to do a lot of things that are more complicated without them:
  - setting size (**`setSize`**), color (**`setColor`**)
  - moving the square (**`move`** and **`moveTo`**)
  - deleting the square (**`delete`** and **`__del__`**)
  - changing the stacking order (**`lift`**)

Other shapes

- Notice that none of the **`Square`** methods are really specific to **`Square`**s
- If we had a **`Circle`** or a **`Triangle`** class, we would still want to do the following:
  - set the size (**`setSize`**) and color (**`setColor`**)
  - move the shape (**`move`** and **`moveTo`**)
  - delete the shape (**`delete`** and **`__del__`**)
  - change the stacking order (**`lift`**)

Other shapes

- Furthermore, some of the code for these methods would probably be similar or identical:

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

- There is nothing specific to **`Square`**s here
- If this was in a **`Circle`** class, we would define the exact same method!

Other shapes

- In fact, for all these `Square` methods:
  - `move, moveTo`
  - `lift`
  - `setColor, setSize`
  - `scale`
  - `delete, __del__`
- the code would be *exactly* the same for `Circle` methods!
- The only change would be in the `Circle` constructor (`Circle.__init__`)

D.R.Y. again

- We have been telling you to obey the *D.R.Y.* principle in programming:
  - D.R.Y. = Don't Repeat Yourself
- But here, we have two classes (`Square` and `Circle`) which repeat nearly all of their code
  - (everything but the constructor)
- This is highly undesirable
- This kind of situation comes up quite frequently when writing classes

Sharing code

- We would like to have a way to share the code for the methods in the `Square` and `Circle` classes that are identical in those classes
- There is a way to do this: *class inheritance*
- Before we explain this, we need to explain one other thing first

Missing methods

- What happens when we try to call a method on an object when the method doesn't exist?
- For instance, let's say we have a **`Square`** called **`s`** and we think there is a method called **`rotate`** that rotates the **`Square`** by 45 degrees:

```
>>> s.rotate()
AttributeError: Square instance has no
  attribute 'rotate'
```

AttributeError

- When a method or a field of an object does not exist, Python raises an **`AttributeError`** exception
- *Recall*: both fields and methods of objects are considered to be "attributes"
- (Methods are simply attributes which happen to be functions)

Inheritance

- However, it is possible to tell Python that a particular class (call it class **`B`**) *inherits* from another class (call it class **`A`**)
- Meaning: if a method is called on an instance of class **`B`**, and it doesn't exist in class **`B`**, then Python looks for it in class **`A`**
  - if it's found, Python executes it using the **`B`** instance as the **`self`** argument
  - if it's not found, it's an **`AttributeError`** unless class **`A`** also inherits from another class

Inheritance

- Syntax:

```
class A:
    # whatever

class B(A):  # class B inherits from class A
    # whatever
```

- Terminology:
  - class **A** is the *superclass* of class **B**
  - class **B** is the *subclass* of class **A**

Inheritance

- Inheritance is mainly useful when you have different classes that are slight variations of each other
- In our example, the **`Square`** class and the **`Circle`** class share most of their code
- We could make one of them inherit from the other, but...
- It's more conceptually correct to define a "generic" **`Shape`** class that both **`Square`** and **`Circle`** inherit from

The `Shape` class

- The **`Shape`** class will contain all the methods we previously defined for the **`Square`** class
- We will *not* define a constructor (**`__init__`** method) for **`Shape`**s
- There is no such thing as a "generic **`Shape`**" so a constructor wouldn't be useful
- Therefore, we will not be creating **`Shape`** objects
  - We are only using the **`Shape`** class as something for **`Square`** and **`Circle`** to inherit from
  - This is called an "abstract base class" in OOP lingo

The Shape class

```
class Shape:
    '''Generic shape class.  More specific shape
    classes should inherit from this class.'''
    def move(self, x, y):
        # code from earlier lecture
    def moveTo(self, x, y)
        # code from earlier lecture
    def setColor(self, color):
        # code from earlier lecture
    def setSize(self, size):
        # code from earlier lecture
    # etc.
```

The `Shape` class

- We moved all the method definitions (except `__init__`) from `Square` into the `Shape` class
- How do we now define the `Square` class?

The `Square` class

```
class Square(Shape):
    '''Objects of this class represent squares
    on a Tkinter canvas.'''

    def __init__(self, canvas, center, size, color):
        # code from earlier lecture
```

- No other method definitions!
- The notation

```
class Square(Shape): ...
```

- means that class **`Square`** inherits from class **`Shape`**

The `Square` class

- So far, all we have done is split what used to be the `Square` class between the `Shape` class (the methods) and the `Square` class (the constructor)
- This works because any method requested on `Square` instances which aren't there (`move`, `setSize` etc.) are looked for in the `Shape` class
- However, we haven't saved ourselves any work yet
- Let's define a `Circle` class

The `Circle` class

- A `Circle` class would need a different constructor
- `Canvas` objects use the `create_oval` method to create circles, not the `create_rectangle` method
- Let's write the `Circle` class constructor

The `Circle` class

```
def __init__(self, canvas, center, size, color):
    '''Create a new Circle object.'''
    (x, y) = center
    x1 = x - size / 2
    y1 = y - size / 2
    x2 = x + size / 2
    y2 = y + size / 2
    self.handle = canvas.create_oval(x1, y1,
        x2, y2, fill=color, outline=color)
    self.canvas = canvas
    self.center = center
    self.size   = size
    self.color  = color
```

The `Circle` class

- Now that we have the constructor, what's the rest of the class?

```
class Circle(Shape):
    def __init__(self, canvas,
                 center, size, color):
        # as before
```

- This is the *entire* class definition!
- We have saved ourself a *ton* of work!

Inheritance recap

- When an object (an instance of some class) tries to access a method and that method is not present in that object (*i.e.* in the class of which the object is an instance)...
- ...then the object looks for the method in the class that its class inherited from (if any); this "parent class" is called its *superclass*
  - also sometimes called a "base class" *e.g.* in the expression "abstract base class"

Inheritance recap

- If the object's class doesn't have a superclass, an `AttributeError` is raised
- If the object's class has a superclass, look for the method in the superclass definition
- If the method is found, call it with the original object as the `self` argument

Inheritance recap

- If the superclass *also* doesn't have the method, then it looks in *its* superclass for the method, etc., until one of two things happen:
  1. the method is found, so execute that method with the original object as the **`self`** argument, or
  2. a class is reached with no superclass, so raise an **`AttributeError`** exception

Overriding a method

- If the superclass does have a particular method, it's still OK for the subclass to have a method with the same name
- In this case, the subclass method will be called when the method is called
- We say that the subclass method *overrides* the superclass method
- We'll see an example of this shortly

The `Triangle` class

- Let's continue, using inheritance to define a **`Triangle`** class
- Most of the methods will be inherited from the **`Shape`** class
- We will need to define only two methods
  - a constructor (**`__init__`**)
  - a **`setSize`** method (overriding the **`Shape`** **`setSize`** method)

Constructor

- The `Triangle` class will use the canvas `create_polygon` method
- This creates a polygon given the (x, y) coordinates of all the vertices (corners) of the polygon
- It also takes `fill` and `outline` keyword arguments as usual

Constructor

```
def __init__(self, canvas, center, size, color):
    '''Create a new Triangle instance.'''
    (x, y) = center
    # Square bounding box of the triangle
    x1 = x - size / 2
    y1 = y - size / 2
    x2 = x + size / 2
    y2 = y + size / 2
    # Continued on next slide...
```

Constructor

```
def __init__(self, canvas, center, size, color):
    # <code from last slide omitted>
    # Construct vertices of the triangle
    p1 = ((x1 + x2) / 2, y1)
    p2 = (x1, y2)
    p3 = (x2, y2)
    self.handle = \
      canvas.create_polygon(p1[0], p1[1],
        p2[0], p2[1], p3[0], p3[1],
        fill=color, outline=color)
    # continued on next slide
```

Constructor

```
def __init__(self, canvas, center, size, color):
    # <code from last two slides omitted>
    # Set fields of object.
    self.canvas = canvas
    self.center = center
    self.size   = size
    self.color  = color
```

setSize method

```
def setSize(self, size):
    '''Change this object's size.'''
    (cx, cy) = self.center
    x1 = cx - size / 2
    y1 = cy - size / 2
    x2 = cx + size / 2
    y2 = cy + size / 2
    p1 = ((x1 + x2) / 2, y1)
    p2 = (x1, y2)
    p3 = (x2, y2)
    self.canvas.coords(self.handle, p1[0], p1[1],
      p2[0], p2[1], p3[0], p3[1])
    self.size = size
```

`Triangle` class

```
class Triangle(Shape):
    def __init__(self, canvas,
                 center, size, color):
        # as before
    def setSize(self, size):
        # as before
```

- This is the *entire* class definition!
- Everything else is inherited from the **`Shape`** class

`setSize` on `Triangles`

- Consider:

```
t = Triangle(canvas, (100, 100), 50, 'red')
t.setSize(100)
```

- There are two possible **`setSize`** methods that a **`Triangle`** could use:
  - the **`setSize`** method defined in the **`Triangle`** class
  - the **`setSize`** method defined in the **`Shape`** class
- The "most specific" one (the **`Triangle setSize`** method) will be used (overriding the **`Shape setSize`** method)

Working with `Shapes`

- Now we have defined the **`Square`**, **`Circle`** and **`Triangle`** classes
- All of them inherit methods from the **`Shape`** class (their superclass)
- That means that we can have a list of objects which are all **`Shape`**s
  - some **`Square`**s, some **`Circle`**s, some **`Triangle`**s
- and tell all of them to do something
  - *e.g.* change size, move

Working with **Shapes**

```
shapes = [...]  # list of various shapes
for shape in shapes:
    shape.move(50, -40)
```

- This will move every shape 50 pixels to the right and 40 pixels up
- It doesn't matter what kind of shapes are in the list, because *all* shapes have a **`move`** method that works the same way

Polymorphism

- This ability to do something "generic" on a group of shapes, without worrying about what the class of a specific shape is, is *incredibly* useful
  - This is the *most important idea* in object-oriented programming!
- It has a technical name: *polymorphism*
- Here, it means that any object that "knows how to move itself" given `x` and `y` offsets can be moved

Polymorphism

- Inheritance is one easy way to get polymorphism
- All subclasses get access to the same methods, so can use those methods from any subclass

Duck Typing

- Inheritance is not the *only* way to get polymorphism!
- If you have a bunch of objects of unrelated classes (no inheritance) which just happen to define compatible `move()` methods, you could use them in the previous example
- This is often called *"Duck Typing"*
  - "If it walks like a duck, and quacks like a duck, it *is* a duck!"

Demos

- Some examples of what we can do with groups of **`Shape`**s (some **`Square`**s, some **`Circle`**s, some **`Triangle`**s)

Exceptions and classes

- Python functions raise exceptions when an "exceptional" situation arises
  - often when an error occurs (*e.g.* trying to open a non-existent file)
- Can catch these exceptions using a **`try`**/ **`except`** construct in Python
  - in the function where the exception was raised
  - in another function below it in the runtime stack

Exceptions and classes

- However, we haven't talked about two vital topics:
  - raising your own exceptions
  - creating your own exception types
- Without this knowledge, exceptions are not very useful
- With it, can use exceptions to make code much easier to manage in error situations

raise

- If you want to raise your own exception, use a **raise** statement:

```
>>> raise ValueError
```

- This will result in:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError
```

- (We won't show the stack tracebacks from now on)

raise

- You can add an error message to a **raise** statement with most built-in exceptions:

```
>>> raise ValueError('Invalid value')
```

- This will result in:

```
ValueError: Invalid value
```

- This can be used to give more detail about what specific error happened

Example

- Consider writing a function to compute factorials of positive integers (assignment 2)
- factorial(n) = n * (n-1) * (n-2) * ... * 1
- If n < 0, this is invalid
- Can just "hope for the best"
  - or can raise a **`ValueError`** exception if n < 0

Example

```
def factorial(n):
    '''Computes the factorial of n.
    n must be >= 0.'''
    if n < 0:
        raise ValueError('n must be >= 0')
    # ... rest of code for factorial ...
```

- Notice:
  - `raise` statement doesn't have to be in a `try`/`except` statement

Example

- Using the **`factorial`** function:

```
val = int(raw_input('Enter a number: '))
try:
    f = factorial(val)
    print 'The factorial of %d is %d' % (val, f)
except ValueError, e:
    # e is the ValueError object that was raised
    print e  # prints error message
```

Syntax

- The statement:

```
try:
   # code that may raise a ValueError
except ValueError, e:
   # code that uses the ValueError instance
```

- has an **`except`** block where the exception object is given a name (**`e`**)
- The code in the block can use the exception object as it sees fit (*e.g.* by printing it)

Example

```
val = int(raw_input('Enter a number: '))
try:
    f = factorial(val)
    print 'The factorial of %d is %d' % (val, f)
except ValueError, e:
    # e is the ValueError object that was raised
    print e  # prints error message
```

- **`ValueError`** exception was raised in the **`factorial`** function
- It wasn't caught in that function, but in the code that *called* the **`factorial`** function

Slightly different example

- Can handle the exception directly in the **`factorial`** function:

```
def factorial(n):
    '''Compute the factorial of n.'''
    try:
        if n < 0:
            raise ValueError('n must be >= 0')
        # code to compute factorial of n
    except ValueError, e:
        print e
```

Slightly different example

- Handling an exception in the function that raised it is occasionally useful, but more often is *not* the right thing to do
- The question that must be asked is this:
- *Whose responsibility is it if this function fails?*

Slightly different example

- For **`factorial`**, whatever code called **`factorial`** was responsible for giving **`factorial`** a valid input
- If that doesn't happen, it's not **`factorial`**'s problem
  - just raise the exception and let other code handle it
- We say that factorial has a *precondition* that its input is >= 0
- If a precondition is violated, just inform the caller by raising an exception

Slightly different example

- Code that called `factorial` has many options to handle the error:
  - print error message and quit
  - ask user for a different option
  - choose another value (default value)
- Not handling the exception in the function that raised it makes the code more flexible
  - more options for caller to choose from

Exceptions are objects

- An exception is an object
- This is legal:

```
>>> v = ValueError('You messed up!')
>>> raise v
ValueError: You messed up!
```

- `v` is an instance of the `ValueError` class

Exception classes

- Any instance of a class can be an exception
- If you create a class, you can **`raise`** its instances
- Here is the simplest possible class:

```
class MyException:
    pass
```

`pass`

- **`pass`** is a new Python keyword
- It can be used anywhere an indented block is expected
  - in a **`try`**, **`except`**, **`for`**, **`while`** block
  - in a function definition
  - in a class definition
- It means: *do nothing*
- Useful as a "place-holder" when Python requires a block of code, but nothing needs to be done

Exception classes

```
class MyException:
    pass
```

- This is an empty class
  - no constructor, no methods, no nothing!
- Can still make instances of this class:

```
>>> e = MyException()
```

- (though they don't do anything)

Exception classes

- Empty classes like `MyException` are still useful because you can `raise` them

```
>>> raise MyException()
```

- This creates a new `MyException` instance and raises it

Exception classes

- You can catch exceptions created from your own exception classes:

```
try:
    # if something goes wrong:
    raise MyException()
except MyException:
    print 'oops!'
```

- (Normally, would catch this exception in a different function than the one that raised it)

Exception classes

- Trivial exception classes like **`MyException`** cannot take error messages as arguments when creating them

```
>>> raise MyException('You messed up!')
TypeError: this constructor takes no
  arguments
```

- We would like our exception objects to at least be able to contain an error message
- Can solve by adding a constructor

Exception classes

```
class MyException:
    def __init__(self, msg):
        '''Create a MyException object with error
        message 'msg'.'''
        self.msg = msg
```

- The last line stores an error message in the **`MyException`** object

Exception classes

```
class MyException:
    def __init__(self, msg):
        '''Create a MyException object with error
        message 'msg'.'''
        self.msg = msg
```

- Now we can use this as follows:

```
>>> raise MyException('You messed up!')
```

Exceptions and inheritance

- There is a "standard" exception class called **`Exception`** that has several nice features:
  - Can take error messages as argument, or nothing
  - Can print itself out nicely
- Example:

```
>>> raise Exception('You messed up!')
Exception: You messed up!
>>> raise Exception()
Exception
```

Exceptions and inheritance

- You could use **`Exception`** whenever you needed to raise one of your own exceptions
  - adding an error message to give more details
- Much better: create a *subclass* of **`Exception`** for your own exception classes:

```
class MyException(Exception):
    pass
```

- Very simple, but very useful!

Exceptions and inheritance

- Now:

```
>>> raise MyException('You messed up!')
__main__.MyException: You messed up!
>>> raise MyException()
__main__.MyException
```

- Can pass error message to **`MyException`** constructor, or leave it out

Exceptions and inheritance

- Why is using **`MyException`** better than just using **`Exception`**?
- Answer: it's *more specific*
- Most exception classes (**`ValueError`**, **`IOError`**, etc.) are subclasses of **`Exception`**
- Catching **`Exceptions`** catches **`ValueError`**s, **`IOError`**s, **`MyException`**s and almost any other kind of exception

Exceptions and inheritance

```
try:
    raise ValueError('I am a ValueError')
except Exception, e:
    print e
```

- gives:

```
I am a ValueError!
```

Exceptions and inheritance

```
try:
    raise IOError('I am an IOError')
except Exception, e:
    print e
```

- gives:

```
I am an IOError!
```

Exceptions and inheritance

```
try:
    raise MyException('I am a MyException')
except Exception, e:
    print e
```

- gives:

```
I am a MyException!
```

Exceptions and inheritance

```
try:
    raise ValueError('I am a ValueError')
except MyException, e:
    print e
```

- gives a traceback (the `ValueError` exception wasn't caught)
- Why?
- Because a `ValueError` instance is not an instance of `MyException` or an instance of a subclass of `MyException`

Exceptions and inheritance

- When using exceptions, want to handle the *most specific* exception possible
- **`MyException`** exceptions could only have been raised by *your* code
  - not by *e.g.* Python built-in modules
- **`Exception`** exceptions could have come from *any* function
  - no idea where a particular exception came from
  - though traceback will tell you this too

Exceptions and inheritance

- Rules of thumb:
- When catching exceptions, catch whatever exceptions that could have been raised by anything in the `try` block
  - as specifically as possible
- When raising exceptions, *usually* want to define your own exception class or classes to make sure that you can catch *only* your own exceptions if you want to

Next time

- Odds and Ends, part 2
  - Minor topics we haven't had time to discuss yet