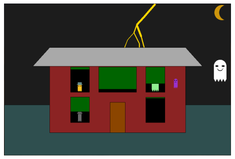
Animation

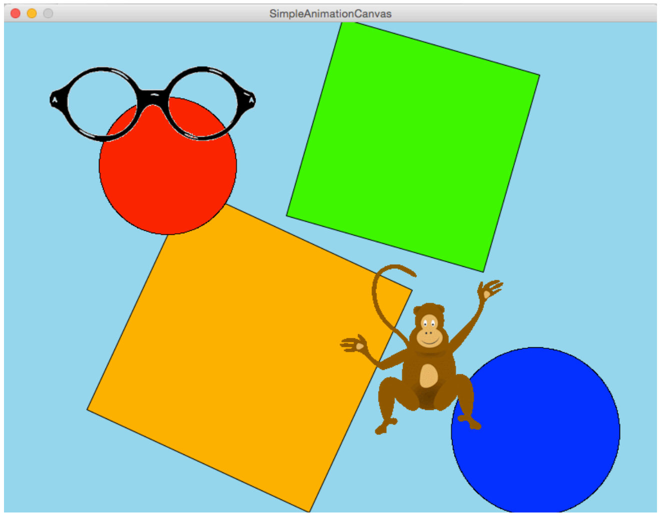


CS111 Computer Programming

Department of Computer Science
Wellesley College

Today: Animation with objects

Run the file simpleAnimation.py. Not from Canopy, from the command line.



Review: Objects

- An object is a data value that has state and behaviors.
- Strings, lists, tuples, dictionaries, and even numbers are also objects in Python.
- As are circles, canvases, points, etc. in cs1graphics
- Behaviors are defined by methods that can be invoked on an object. A method is a named sequence of instructions for an object.
- For example:

```
 sq = Square(size, Point(centerX, centerY))
sq.setFillColor(color)
sq.rotate(initialAngle)
```

Review: Classes

- A class is a description of the shared characteristics (state and behaviors) of a set of objects.
- A class is used like a mold for making objects.
- An object made from a class is called an instance of the class.
- Example of classes include:

```
str       list
int       dict
Canvas    Polygon
```

Today: how to use classes and objects to make our own animation

Classes in Python

- Convention:
  - We will start names of classes with an upper case letter, and continue in lower case (except to indicate word boundaries).
    - class ThisIsALegitimateClassName:
    - class AndThis:
    - class Canvas:
- This is just a convention. But you will confuse readers of your code if you write unconventional names like:
    - class badName:
    - class classWithATERRIBLEname

Modules in Python

- A module is a file containing python definitions.
  - https://docs.python.org/2/tutorial/modules.html
- Convention: name your modules in lowercase.
- Our Intention:
  - Modules will contain definitions of classes
  - Try to keep related classes together in one module
- We're not the only ones
  - cs1graphics.py had a lot of class definitions
  - Many classes: Canvas, Circle, etc.

Importing Modules

- The old, lazy way:
  - from cs1graphics import *
  - Lets us refer to Canvas, Circle, Polygon, etc
- The better way:
  - import cs1graphics
  - Now you must use full names of classes:
  - cs1graphics.Canvas, cs1graphics.Polygon, etc
- The preferred "Pythonic" way:
  - import cs1graphics as cs1g   # or some other short name
  - Now the full names of classes are shortened:
  - cs1g.Canvas, cs1g.Polygon, etc

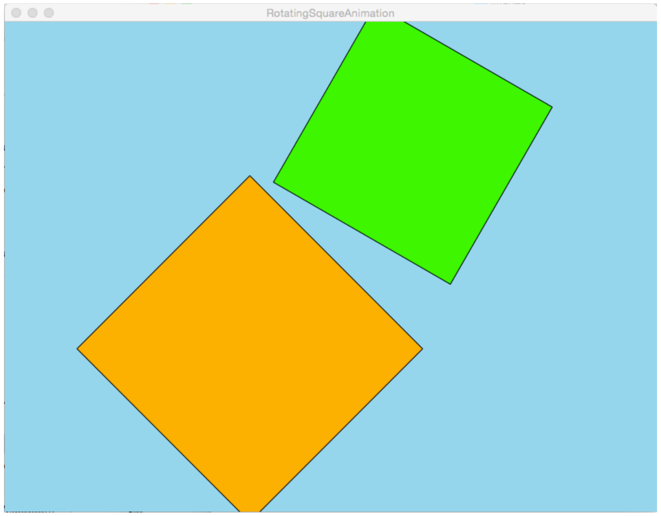
Animation Parts

There are two main parts to our animation framework:

1. *Sprites* are the actresses in an animation. We create a cast of sprites to act in our play. Each sprite knows how to perform its own part. In particular it knows how to update its state for each time step of the animation.
2. *Animations* are the plays in which the sprites act. An animation has a canvas on which visual representations of the sprites are displayed. At each time step of the animation, each sprite is asked to update its state, which often changes how it appears on the canvas. As each sprite changes, we see a "movie" of the sprites' performances.

Just RotatingSquares

Run the file rotatingSquaresAnimation.py. Not from Canopy, from the command line.



Animation Class

```
from cs1graphics import *

class Animation:

    def __init__(self, width, height, color, title):
        # Create canvas for showing the sprites
        self.canvas = Canvas(width, height, color, title)
        # Create empty list of sprites
        self.sprites = []

    def addSprite(self, sprite):
        self.sprites.append(sprite)
        sprite.addToCanvas(self.canvas)
            # Sprite determines how to add itself to canvas.

    def start(self):
        while True: # animation is infinite loop.
                    # Stop it using Ctrl-C Ctrl-C.
            for sprite in self.sprites:
                sprite.step()
```

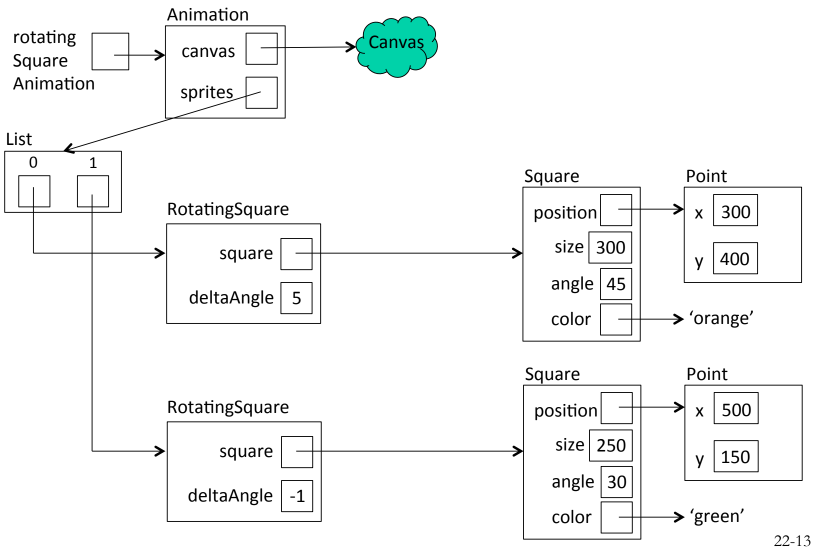
RotatingSquare Class

```
from cs1graphics import *

class RotatingSquare:
    '''Colored square that rotates.'''

    def __init__ (self, centerX, centerY, size,
                  color, initialAngle, deltaAngle):
        sq = Square(size, Point(centerX, centerY))
        sq.setFillColor(color)
        sq.rotate(initialAngle)
        self.square = sq
        self.deltaAngle = deltaAngle

    def addToCanvas(self, canvas):
        canvas.add(self.square)

    def step(self):
        self.square.rotate(self.deltaAngle)
```

RotatingSquare Animation

```
rotatingSquares = Animation(800, 600, 'skyblue',
                            'RotatingSquares')

rotatingSquares.addSprite( \
  RotatingSquare(300, 400, 300, 'orange', 45, 5))

rotatingSquares.addSprite( \
  RotatingSquare(500, 150, 250, 'green', 30, -1))

rotatingSquares.start()
```

Object-Based Diagram



Sprite Inheritance: MovingRotatingSquare

```
class MovingRotatingSquare(RotatingSquare):
    '''Colored square that rotates and
    moves horizontally with speed deltaX'''

    def __init__ (self, centerX, centerY, size, color,
                  initialAngle, deltaAngle, deltaX):
        # Explicitly invoke superclass constructor
        RotatingSquare.__init__(self, centerX, centerY,
                                size, color,
                                initialAngle, deltaAngle)
        self.deltaX = deltaX

    # addToCanvas method inherited

    def step(self):
        RotatingSquare.step(self) # Rotate the square
        self.square.move(self.deltaX, 0) # Move the square
```

MovingRotatingSquare Animation

it's your turn

In **movingRotatingSquareAnimation.py**
add a new sprite and run the app again.

```
movingRotatingSquares = \
    Animation(800, 600, 'skyblue', 'MovingRotatingSquares')


movingRotatingSquares.addSprite( \
    MovingRotatingSquare(300, 400, 300, 'orange', 45, 5, 3))

movingRotatingSquares.addSprite( \
    MovingRotatingSquare(500, 150, 250, 'green', 30, -1, -2))

movingRotatingSquares.start()
```

Sprite Inheritance: BouncingRotatingSquare

```
class BouncingRotatingSquare(MovingRotatingSquare):
    '''Colored square that rotates and moves horizontally
    with speed deltaX, and bounces off vertical canvas edges'''

    # __init__ method inherited

    # Override inherited addCanvas method
    def addToCanvas(self, canvas):
        canvas.add(self.square)
        self.maxX = canvas.getWidth()

    def step(self):
       MovingRotatingSquare.step(self) # Rotate & move square
       pos = self.square.getReferencePoint()
       centerX = pos.getX()
       centerY = pos.getY()
       if centerX < 0 or centerX > self.maxX:
           self.square.moveTo(centerX, centerY)
           self.deltaX = -self.deltaX # Change direction
```

Try it out: BouncingRotatingSquare

```
movingRotatingSquares = \
    Animation(800, 600, 'skyblue', 'MovingRotatingSquares')


movingRotatingSquares.addSprite( \
    MovingRotatingSquare(300, 400, 300, 'orange', 45, 5, 3))

movingRotatingSquares.addSprite( \
    MovingRotatingSquare(500, 150, 250, 'green', 30, -1, -2))

movingRotatingSquares.start()
```

RotatingSquare that inherits from Square!

```
# Version of RotatingSquare that inherits directly
# from Square (a Drawable)
class RotatingSquare(Square):
    '''Colored square that rotates.'''

    def __init__ (self, centerX, centerY, size, color,
                  initialAngle, deltaAngle):
        Square.__init__(self, size, Point(centerX, centerY))
        self.setFillColor(color)
        self.rotate(initialAngle) # rotate self directly
        # self.square = sq # <== no need for this anymore!
        self.deltaAngle = deltaAngle

    def addToCanvas(self, canvas):
        # canvas.add(self.square)
        canvas.add(self) # RotatingSquare *is* a Drawable

    def step(self):
        # self.square.rotate(self.deltaAngle)
        self.rotate(self.deltaAngle) # RotatingSquare *is*
                                     # a drawable
```

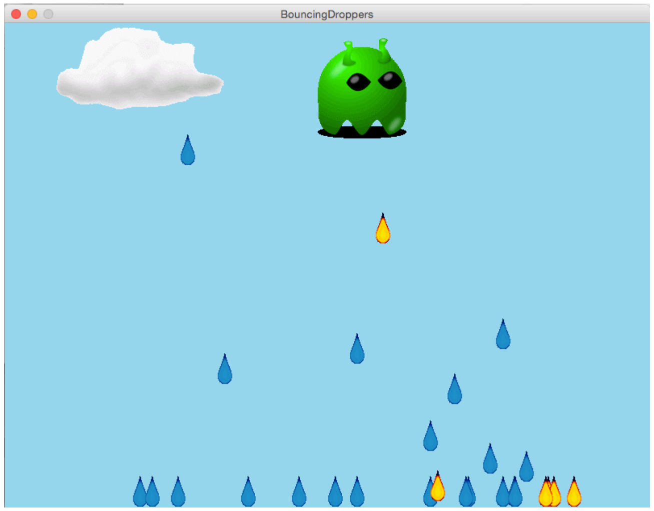
Rotating Shapes

it's your turn

- Run `pulsingCirclesAnimation.py`
- Define and implement a new class `PulsarFromCircle` that inherits from `Circle`. Base your implementation on `Pulsar.py`.
- Update `pulsingCirclesAnimation.py` to include `PulsarFromCircle` objects.

BouncingDroppers

An animation in which a sprite can create other sprites.
Try it out: bouncingDropperAnimation.py. Not from Canopy, from the command line.



BouncingDropper [1]

```
import randomclass
class BouncingDropper (BouncingImage):
    '''Horizontally moving image that bounces off vertical
    edges of canvas and drops a DroppingImage with a
    given probability.'''

    def __init__ (self, centerX, centerY, picfile, deltaX,
                  animation, droppedImage, dropProbability,
                  droppedImageSticks)
        BouncingImage.__init__(self, centerX, centerY,
                               picfile, deltaX)
        self.animation = animation
        self.droppedImage = droppedImage
        self.dropProbability = dropProbability
        # should be between 0.0 and 1.0
        self.droppedImageSticks = droppedImageSticks

    # Inherit addToCanvas from BouncingImage

    # Implementation continued on the next slide
```

BouncingDropper [2]

```
def step(self):
    BouncingImage.step(self) # Move the image
    if random.random() < self.dropProbability:
        # random.random() returns random number between
        # 0.0 and 1.0.

     # Add new FallingImage to animation, with random
     # falling speed  between 5 and 15.
     pos = self.image.getReferencePoint()
     if self.droppedImageSticks:
         self.animation.addSprite(FallingImageSticks(  \
             self.animation, pos.getX(),
             pos.getY() + self.image.getHeight()/2.0,
             self.droppedImage, random.randint(5,15)))
     else:
         self.animation.addSprite(FallingImageDisappears( \
             self.animation, pos.getX(),
             pos.getY() + self.image.getHeight()/2.0,
             self.droppedImage, random.randint(5,15)))
```

FallingImageSticks

```
class FallingImageSticks:
    '''Vertically falling image that sticks to bottom of canvas.'''
    def __init__ (self, animation, centerX, centerY, imageFile, deltaY):
        '''Assume deltaY is positive'''
        self.animation = animation
        img = Image(imageFile)
        img.moveTo(centerX, centerY)
        self.image = img
        self.deltaY = deltaY

    def addToCanvas(self, canvas):
        self.maxY = canvas.getHeight() - self.image.getHeight()/2.0
        canvas.add(self.image)

    def step(self):
        '''sprite falls and sticks to bottom of canvas'''
        pos = self.image.getReferencePoint()
        centerX = pos.getX()
        centerY = pos.getY() + self.deltaY
        if centerY > self.maxY:
            centerY = self.maxY
        self.image.moveTo(centerX, centerY) # Stick at bottom
```

Problem: Sprites never go away!

Animation gets slower and slower as more sprites are added.

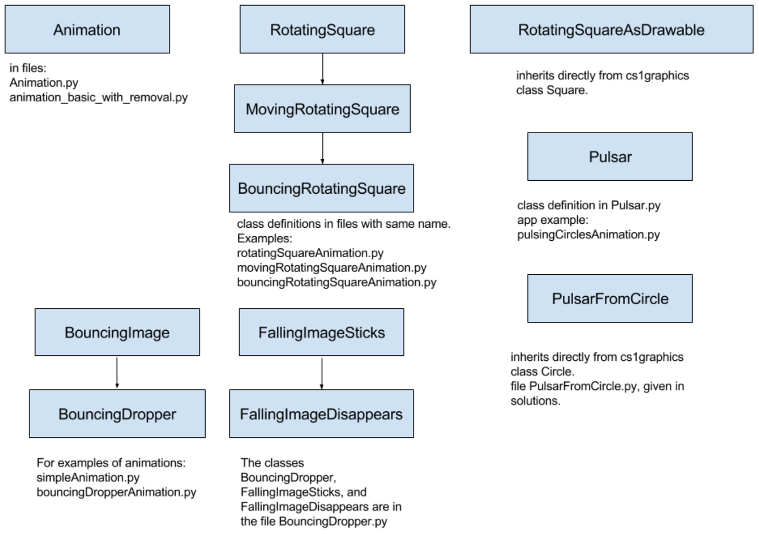
Let's add a method to `Animation` class for removing sprites:

```
def removeSprite(self, sprite):
    self.sprites.remove(sprite) # Remove sprite from list
    sprite.removeFromCanvas(self.canvas)
    # Sprite determines how to remove itself from canvas
```

FallingImageDisappears

```
class FallingImageDisappears(FallingImageSticks):
    '''Vertically falling image that disappears when it
    hits bottom of canvas.'''

    # Inherits __init__ from FallingImageSticks

    # Inherits addToCanvas from FallingImageSticks

    def removeFromCanvas(self, canvas):
        canvas.remove(self.image)

    def step(self):
        '''sprite falls and disappears'''
        FallingImageSticks.step(self)
        if self.image.getReferencePoint().getY() >= self.maxY:
            self.animation.removeSprite(self)
```

All classes and files for Animation



in files:
Animation.py
animation_basic_with_removal.py

class definitions in files with same name.
Examples:
rotatingSquareAnimation.py
movingRotatingSquareAnimation.py
bouncingRotatingSquareAnimation.py

inherits directly from cs1graphics
class Square.

class definition in Pulsar.py
app example:
pulsingCirclesAnimation.py

inherits directly from cs1graphics
class Circle.
file PulsarFromCircle.py, given in
solutions.

For examples of animations:
simpleAnimation.py
bouncingDropperAnimation.py

The classes
BouncingDropper,
FallingImageSticks, and
FallingImageDisappears are in
the file BouncingDropper.py

Benefits of Object-Oriented Programming

- Modularity:
  - Common states and behaviors packaged up
- Polymorphism:
  - Same method can do different things for different types of objects (e.g: **step**)
- Encapsulation:
  - Hide details of how we store the object's information (e.g. different solutions for **MutableString**)
- Inheritance:
  - Objects that are specific types of others share states and behaviors (no repetition of code)

Python vs. Java

Does Python *actually* have encapsulation?

- We can access the state variables of a class directly (nothing is hidden)
- Some languages like Java allow private variables and methods, which can only be accessed within class