

JyScript User's Guide

Java has several features that make it attractive for mathematical illustration. First, it was designed with powerful graphical capabilities that make the production of high-quality illustrations possible. In addition, Java enables us to create dynamic illustrations, such as animations, that may be used to demonstrate or explore a mathematical idea more effectively. Finally, Java programs may be easily distributed over the Internet as applets running in web browsers.

This guide describes JyScript, a Python module that allows programmers to write Python code to create illustrations easily in Java. In particular, JyScript uses the same set of graphics commands as Bill Casselman's PiScript, a Python module for creating figures in PostScript. My intent is to allow programmers to create illustrations in either PostScript or Java with a single graphics command set and without having to master much Java.

It is, at least in theory, possible to accomplish anything with JyScript that can be accomplished with pure Java. This may not be readily apparent, however, as my intention is to sweep Java under the rug as much as possible and give programmers the pleasure and efficiency of developing purely in Python. My experience is that JyScript programs may be developed much more quickly than the equivalent programs in Java. I trust that experienced Java users can determine how to add capabilities they may find missing.

JyScript programs will be run with Jython, an implementation of Python written in Java and freely available on the Internet. In essence, we will program in Python but have access to all the Java classes. Instructions for installing both JyScript and Jython are given in an appendix to this guide.

The JyScript home page is

<http://merganser.math.gvsu.edu/david/jyscript>

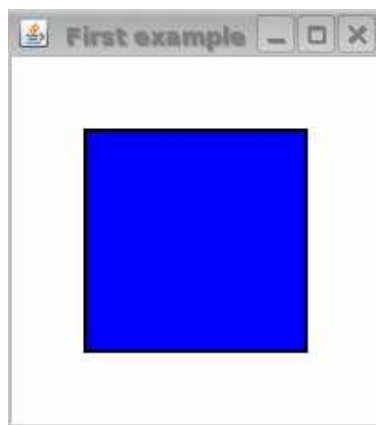
while the PiScript home page is

<http://www.math.ubc.ca/~cass/piscript>.

This manual provides an introduction to using JyScript and attempts to explain its novel features assuming that the reader is familiar with PiScript. An appendix provides a list of all the graphics commands and how they may be used.

1: A first example

We'll now look at a simple example, which causes the following window to appear when run.



Notice the following two components: a “frame,” consisting of the title bar and the gray border, and a “panel,” the white region on which the blue rectangle is drawn. You may wish to think of this like an artist’s canvas inside a frame. Our program will need to create the frame and display it on the screen, create the panel and put it inside the frame, and give instructions to the panel for drawing the figure.

Here is the program that does it. Place the code in a file `Box.py` and run it with the command `jython Box.py`.

```
from JyScript import *
from javax.swing import *

def draw(panel):
    panel.beginpage()

    panel.newpath()
    panel.box(40, 40, 120, 120)
    panel.fill(0, 0, 1)
    panel.setlinewidth(2)
    panel.stroke()

    panel.endpage()

frame = JFrame('First example')
jypanel = JyPanel(200, 200, draw)
frame.getContentPane().add(jypanel)
frame.pack()
frame.setVisible(True)
```

This example contains several crucial ingredients, which every JyScript program needs, so let’s study it carefully.

The first two lines

```
from JyScript import *
from javax.swing import *
```

import the JyScript module as well as the Java classes in the `javax.swing` package. With `jython`, we may access any Java class; one of the classes in `javax.swing` is `JFrame`, which will provide us with a frame to display on the screen and hold the panel.

The drawing instructions are given in a function called `draw`, the body of which looks a bit like PiScript code. There are a few differences, however, from PiScript. First, we do not need to call the `init()` and `finish()` functions; the initialization will be performed elsewhere and we don’t need to `finish()` since we are not writing to a file as PiScript does. Also, the `draw` function has an argument, `panel`, which is referenced in all the graphics commands. For reasons that will become apparent later, we need to specify in this way which panel we are drawing on because we may create figures with several panels.

The default coordinate system has the origin at the lower left corner of the panel with the units in the horizontal and vertical directions in pixels. Just as in PiScript, this coordinate system may be modified using, say, `translate`, `rotate`, and the like.

After defining our function `draw`, we finish with

```
frame = JFrame('First example')
jypanel = JyPanel(200, 200, draw)
frame.getContentPane().add(jypanel)
frame.pack()
frame.setVisible(True)
```

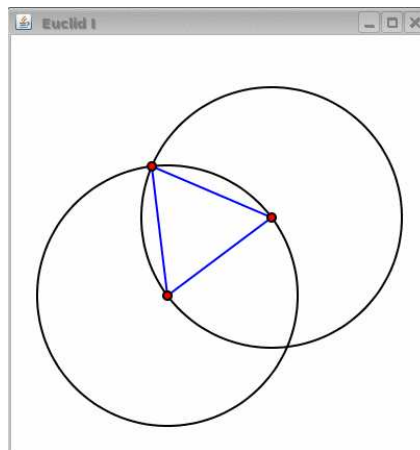
The first line creates the frame that holds the figure. Notice that the text “First example” appears as the title of the frame. Next, we create a `JyPanel`, the panel on which we will draw. To create a `JyPanel`, we specify its width, height, and the function that contains the instructions for drawing on it. This is where the panel’s initialization takes place. `JyPanel` is provided by the `JyScript` module.

Finally, the third line places the `JyPanel` into the `JFrame` (we’ll say more about this later); the fourth line asks for space to be allocated to all the things placed inside the frame (in this example, there is only our panel); the last line causes the `JFrame` to appear on our screen.

2: Another example

Our first example is a typical `JyScript` program: many other figures may be created from it by simply changing the body of the `draw` function.

The next example, for the most part, should be self-explanatory, while demonstrating a greater range of the graphics commands. Besides importing the `math` module, this program differs from the first only in the `draw` function, which we include below.



```
def draw(p):
    p.beginpage()
    p.antialiasing(True)           # We'll discuss this in a moment
    p.purestroke(True)            # This too

    p.gsave()
    p.center()                    # Set up the coordinate system
    p.scale(50)
    p.scalelinewidth(2)
```

```
p0 = [-1, -1]                # Define the centers of the circles
p1 = [1, 0.5]
dx = p1[0] - p0[0]
dy = p1[1] - p0[1]
dist = math.sqrt(dx*dx + dy*dy)

p.newpath()                  # Draw first circle
p.circle(p0, dist)
p.stroke()

p.newpath()                  # Draw second circle
p.circle(p1, dist)
p.stroke()

p.gsave()                    # Set up coordinate system for triangle
p.newpath()
p.translate(p0)
p.setrad()
p.rotate(math.atan2(dy, dx))

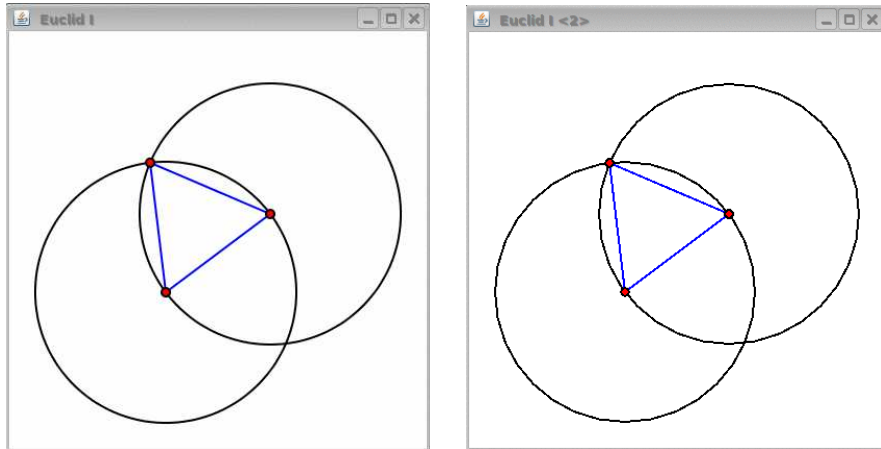
p.gsave()
p.moveto(0, 0)
for i in range(3):          # Draw triangle
    p.lineto(dist, 0)
    p.translate(dist, 0)
    p.rotate(2.0*math.pi/3)
p.closepath()
p.stroke([0, 0, 1])
p.grestore()

for i in range(3):          # Draw vertices of triangle
    p.newpath()
    p.circle(0, 0, 0.08)
    p.fill([1, 0, 0])
    p.stroke()
    p.translate(dist, 0)
    p.rotate(2.0*math.pi/3)

p.grestore()                # Restore coordinate system
p.grestore()
p.endpage()
```

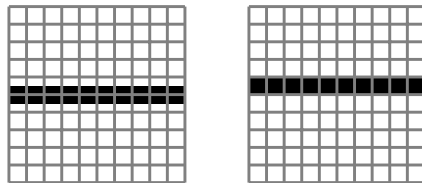
Most of this should look familiar to PiScript users. Two novel features are the `antialiasing` and `purestroke` commands, which we will now explain.

A fundamental difference between graphics in PostScript and Java is that Java images are typically rendered at low-resolution. For instance, the last example was drawn on a rectangular grid of 400×400 pixels. This can result in poor-quality images; paths, in particular, can appear jagged. Antialiasing is a technique used to smooth out these jagged edges. Shown below is the same image with antialiasing set to `True`, on the left, and `False`, on the right. Most figures will benefit from antialiasing.

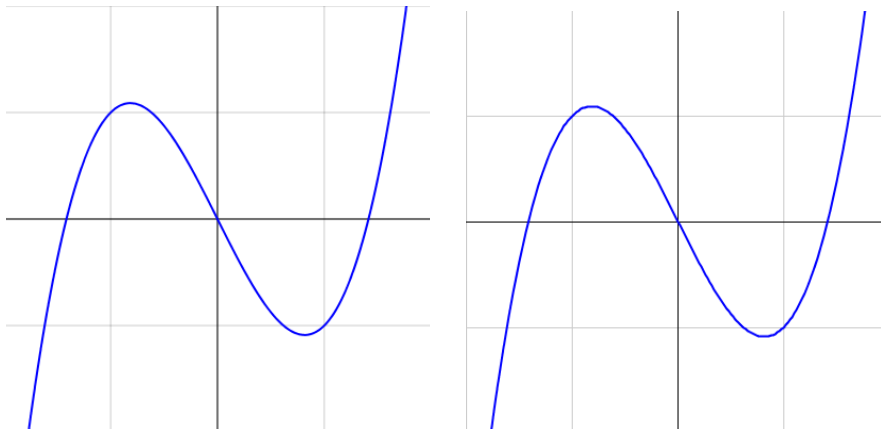


Zooming in on these images (assuming you are reading this online) shows how antialiasing creates the illusion of smoothness by appropriately shading pixels near the paths.

Now suppose that we want to draw a horizontal line, whose width is one pixel, from pixel (0, 5) to pixel (10, 5). As seen on the left below, the line will straddle two rows of pixels. Java typically adds 1/2 to pixel coordinates so that such a line will lie entirely inside a row of pixels, as shown on the right.



When dealing with mathematical illustrations, it is sometimes important to disable this feature: if we compute the precise location of our path, we don't want Java to distort it in this way. Shown below is the graph of a cubic function with `purestroke` set to `True`, on the left, and `False`, on the right. The effect may be difficult to see unless you look closely. If reading online, the difference will be apparent if you zoom in (look near the local maximum and minimum). My experience is that this effect is not frequently important, but it can lead to serious problems in a few instances. Graphs will typically benefit from having `purestroke` set to `True`, but if you are simply drawing boxes, you are best off setting it to `False`.



By the way, here is the draw function used to create these figures.

```
def draw(p):
    p.beginpage()

    p.antialiasing(True)
    p.purestroke(True)

    p.gsave()                # Set up the coordinate system
    p.center()
    p.scale(100)

    p.newpath()             # Draw the grid
    for i in range(-2, 3):
        p.moveto(i, -2)
        p.lineto(i, 2)
        p.moveto(-2, i)
        p.lineto(2, i)
    p.stroke(0.8)

    p.newpath()             # Draw the axes
    p.moveto(-2, 0)
    p.lineto(2, 0)
    p.moveto(0, -2)
    p.lineto(0, 2)
    p.stroke()

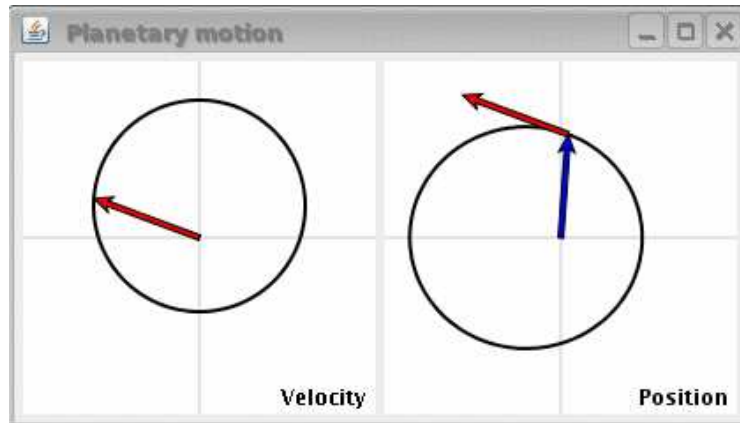
    p.newpath()             # Draw the function
    p.setlinewidth(2)
    p.graph(lambda x: x**3 - 2*x, -2, 2)
    p.stroke([0, 0, 1])

    p.grestore()
    p.endpage()
```

You may wish to note the use of the `lambda` function here as a convenient way to define mathematical functions.

3: Layout

It sometimes happens that we would like to place two or more panels into a single frame. Here's an example demonstrating Hamilton's theorem relating the velocity and position of, say, a planet moving under the gravitational influence of the sun.



We will do this using two JyPanels. But first, we define functions to do the drawing in the two panels.

```
def velocity(p):
    p.beginpage()
    ...
    p.endpage()
def position(p):
    p.beginpage()
    ...
    p.endpage()
```

Now we create our JFrame and two JyPanels:

```
frame = JFrame("Planetary motion")
velocitypanel = JyPanel(200, 200, velocity)
positionpanel = JyPanel(200, 200, position)
```

Here comes the novel part. Remember that in earlier examples, we added our panel to the frame using

```
frame.getContentPane().add(panel)
```

Now that we have two panels, we add them in the same way, but we need to specify how they should be positioned. This process is called *layout*. To have the panels placed from left to right in the order added, we will use a FlowLayout provided by the Java package `java.awt`

```
from java.awt import *
frame.getContentPane().setLayout(FlowLayout())
frame.getContentPane().add(velocitypanel)
frame.getContentPane().add(positionpanel)
```

Then we conclude with the usual:

```
frame.pack()
frame.setVisible(True)
```

To summarize, besides the definitions of the `velocity` and `position` functions, the program looks like this:

```
frame = JFrame("Planetary motion")
p0 = JyPanel(200, 200, velocity)
p1 = JyPanel(200, 200, position)
frame.getContentPane().setLayout(FlowLayout())
frame.getContentPane().add(velocitypanel)
frame.getContentPane().add(positionpanel)
frame.pack()
frame.setVisible(True)
```

The two most common layouts are the `FlowLayout`, which we just met, and `BorderLayout`, which we will see later. If you need a more sophisticated layout, you may want to consult the Java tutorial at: <http://java.sun.com/docs/books/tutorial/uiswing/layout/visual.html>.

One new feature here is the addition of text, which works like this:

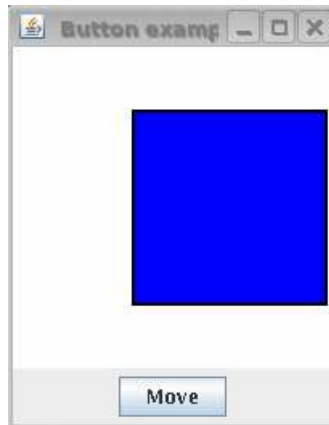
```
p.setFont(12)
p.moveTo(p.width()-5, 5)
string = "Position"
p.shift(-1, string)
p.show(string)
```

We first set the size of the font, in pixels, and then move to the place where we wish to place the text. The `shift` command causes us to move horizontally by a specified factor times the length of the string. In this case, we move back the entire length of the string so that the text is right-justified. Finally, we display the text with the `show` command.

Now that we have several panels in our figures, we can explain why the drawing function needs to have the panel as an argument. When the panels are painted, Java calls a function, `paintComponent`, in the `JyPanel` being painted. The `paintComponent` function, in turn, calls the function, which we have often called `draw`, that is specified when the panel is constructed. The panel calls `draw` passing itself as the argument so that the function knows the appropriate panel on which to draw. This is a little inconvenient, but it seems a fair price for the advantage of working with several panels in one frame.

4: Adding a button

Up to this point, our figures have been static, much like PostScript figures. One motivation for using JyScript is that our diagrams may be made to respond to changes that we ask for. To begin with a simple example, let's go back to our first example, `Box.py` and add a button, which causes the box to move to the right by 5 pixels when pressed.



A couple of new issues show up here. We will need to create a button, tell it what to do when pressed, and add it to the frame. Let's begin with the drawing function, however. We introduce a variable `x` to keep track of where the box will be drawn, but otherwise the function is unchanged.

```
x = 40
def draw(p):
    p.beginpage()

    p.newpath()
    p.box(x, 40, 120, 120)
    p.fill(0, 0, 1)
    p.setlinewidth(2)
    p.stroke()

    p.endpage()
```

We need a function, which we name `move`, to be called when the button is pressed.

```
def move():
    global x
    x += 5
    panel.repaint()
```

Here, the variable `x` is increased by 5 and we ask the `JyPanel` to paint itself again. This is crucial since otherwise we will not see any changes made to the figure. Forgetting to ask the panel to repaint is a common mistake when making figures dynamic.

Next, we need to create a button that displays the word "Move" and tell it to call the `move` function when pressed.

```
button = JyButton("Move", move)
```

Finally, we have some layout issues in adding the button to the frame. Since we want the button underneath the panel, a `FlowLayout` will not work. Instead, we need a `BorderLayout`, which divides the frame into five pieces: a large area in the center and smaller regions to the south, west, north, and east. We will put the panel in the center region and the button in the south region.

To use the BorderLayout, we need to import a Java package

```
from java.awt import *
```

then we place our panel in the center with

```
frame.getContentPane().add(panel, BorderLayout.CENTER)
```

We could add the button with

```
frame.getContentPane().add(button, BorderLayout.SOUTH)
```

but then the button would expand to fill the entire lower portion of the frame. Instead, we add the button into a new panel, provided by Java's JPanel, and add that panel to the south region.

```
buttonpanel = JPanel()
buttonpanel.add(button)
frame.getContentPane().add(buttonpanel, BorderLayout.SOUTH)
```

That's it! Here's the whole thing.

```
from JyScript import *
from javax.swing import *
from java.awt import *

x = 40
def draw(p):
    p.beginpage()
    p.newpath()
    p.box(x, 40, 120, 120)
    p.fill(0, 0, 1)
    p.setlinewidth(2)
    p.stroke()

    p.endpage()

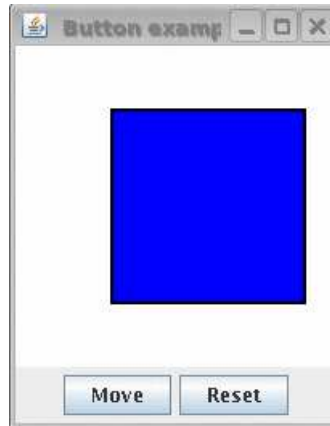
def move():
    global x
    x += 5
    panel.repaint()

button = JButton("Move", move)
buttonpanel = JPanel()
buttonpanel.add(button)

frame = JFrame('Button example')
panel = JPanel(200, 200, draw)

frame.getContentPane().add(panel, BorderLayout.CENTER)
frame.getContentPane().add(buttonpanel, BorderLayout.SOUTH)
frame.pack()
frame.setVisible(True)
```

If you click the button a few times, the box will disappear off the right side of the figure. We need to add a button to reset the figure to its original configuration.



We have seen everything we need to know, but let's look at the details anyway. We need:

- a function to reset the variable `x`. (Don't forget to repaint!)

```
def reset():
    global x
    x = 40
    panel.repaint()
```

- to create the new button and tell it what to do when pressed.

```
resetbutton = JyButton("Reset", reset)
```

- to add the reset button to `buttonpanel` (JPanel's default layout is `FlowLayout`).

```
buttonpanel.add(resetbutton)
```

5: Animations

Clicking that button sure does wear me out. Wouldn't it be nice to have the program to do it for us? We can do this with a `JyTimer`, an object that will call a specified function at regular intervals. We create a timer like this:

```
timer = JyTimer(25, move)
```

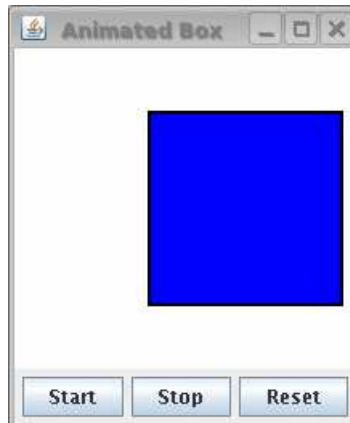
The argument 25 tells the `JyTimer` how often it should call this function. Times here are measured in milliseconds so this gives us 0.025 seconds between function calls or 40 frames per second. (In this example, I changed the `move` function to increment `x` by 1 rather than 5 to produce a smoother animation.)

We also need buttons to start and stop the timer.

```
startbutton = JyButton("Start", begin)
stopbutton = JyButton("Stop", pause)
```

The buttons will call the functions:

```
def begin():
    timer.start()
def pause():
    timer.stop()
```



For the sake of completeness, here is the entire program.

```
from JyScript import *
from javax.swing import *
from java.awt import *

x = 40
def draw(p):
    p.beginpage()
    p.newpath()
    p.box(x, 40, 120, 120)
    p.fill(0, 0, 1)
    p.setlinewidth(2)
    p.stroke()

    p.endpage()
def move():
    global x, panel
    x += 1
    panel.repaint()
def reset():
    global x
    timer.stop()
    x = 40
    panel.repaint()
def begin():
    timer.start()
def pause():
    timer.stop()
```

```
timer = JyTimer(25, move)

startbutton = JyButton("Start", begin)
stopbutton = JyButton("Stop", pause)
resetbutton = JyButton("Reset", reset)

buttonpanel = JPanel()
buttonpanel.add(startbutton)
buttonpanel.add(stopbutton)
buttonpanel.add(resetbutton)

frame = JFrame('Animated Box')
panel = JyPanel(200, 200, draw)

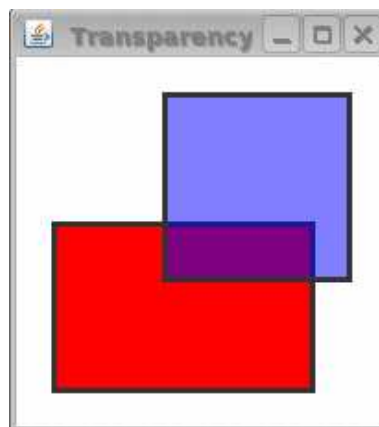
frame.getContentPane().add(panel, BorderLayout.CENTER)
frame.getContentPane().add(buttonpanel, BorderLayout.SOUTH)
frame.pack()
frame.setVisible(True)
```

You may notice that we do not need to set the frame's layout explicitly. This is because the frame's default layout is a BorderLayout.

6: Transparency

Generally speaking, painting an area will cover up whatever happens to be there already. Java, however, allows us to paint with colors having a prescribed amount of transparency. To do this, we specify colors by adding a fourth component, called the *alpha value*, to the list of red, green and blue values. For instance, [0, 0, 1, 0.5] creates a pure blue that is half transparent. An alpha value of 1 is opaque while an alpha value of 0 is transparent.

Here's an example:



The red rectangle is filled and stroked first. Then the blue rectangle is filled with an alpha value of 0.5. Notice how the colors blend in the overlap.

```
def draw(p):
    p.beginpage()
```

```
p.setlinewidth(3)
p.newpath()
p.box(20, 20, 140, 90)
p.fill(1,0,0)
p.stroke(0.2)

p.newpath()
p.box(80, 80, 100, 100)
p.fill(0, 0, 1, 0.5)
p.stroke(0.2)

p.endpage()
```

A second way to achieve this effect is with JyPanel's `setcomposite` function. For instance, if we say

```
p.setcomposite(0.5)
```

then any colors subsequently painted have their alpha values multiplied by 0.5. For instance, if we modify the draw function above, like this:

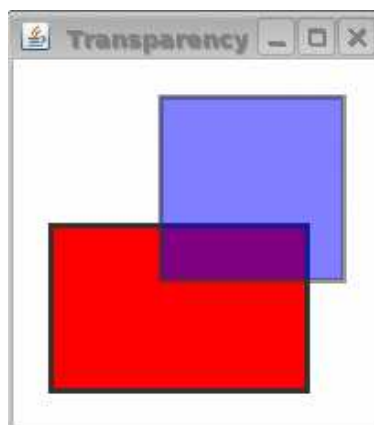
```
def draw(p):
    p.beginpage()

    p.setlinewidth(3)
    p.newpath()
    p.box(20, 20, 140, 90)
    p.fill(1,0,0)
    p.stroke(0.2)

    p.setcomposite(0.5)
    p.newpath()
    p.box(80, 80, 100, 100)
    p.fill(0, 0, 1)
    p.stroke(0.2)

    p.endpage()
```

then we obtain:



You will notice that this figure is similar to the earlier one except that the outline of the second rectangle is also transparent.

The composite value is part of the graphics state so we may undo the effect of `setcomposite` by wrapping it between `gsave()` and `grestore()` calls.

One use of transparency is to create animations in which a feature of the diagram fades in or fades out. Since this can be useful when, for example, illustrating proofs, I've included an example here where the blue box, in the previous example, fades in over the red box.

```
from JyScript import *
from javax.swing import *
from java.awt import *

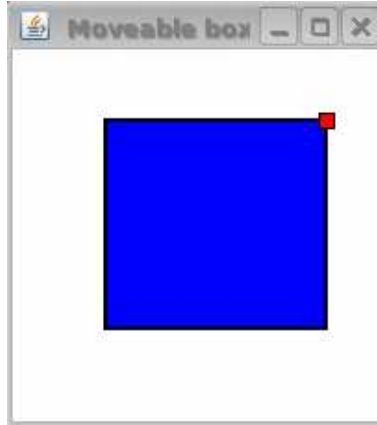
alpha = 0
def draw(p):
    p.beginpage()
    p.setlinewidth(3)
    p.newpath()
    p.box(20, 20, 140, 90)
    p.fill(1,0,0)
    p.stroke(0.2)

    p.setcomposite(alpha)
    p.newpath()
    p.box(80, 80, 100, 100)
    p.fill(0, 0, 1)
    p.stroke(0.2)

    p.endpage()
def increment():
    global alpha
    alpha += 0.01
    if alpha >= 1.0:
        alpha = 1
        timer.stop()
    panel.repaint()
def begin():
    timer.start()
timer = JyTimer(25, increment)
button = JButton("Start", begin)
frame = JFrame('Transparency')
panel = JyPanel(200, 200, draw)
buttonpanel = JPanel()
buttonpanel.add(button)
frame.getContentPane().add(panel, BorderLayout.CENTER)
frame.getContentPane().add(buttonpanel, BorderLayout.SOUTH)
frame.pack()
frame.setVisible(True)
```

7: Moveable points

A final kind of interactivity is given by including points that may be moved inside our figures. For instance, the red point in the following figure may be moved and, as it does, the rectangle is changed so that the red point is one endpoint of a diagonal.



There's not a lot of additional coding required to add moveable points, but a few ideas are necessary before using them. First, moveable points are objects that hold certain data: the current point at which they reside, the coordinate system in which they are drawn, information about how they are drawn, and a function to be called when they are moved.

In this example, the moveable point is created with

```
redpoint = Moveablepoint(1, 2, move)
```

This gives us a point whose coordinates are (1, 2) and that will call the function `move` when an attempt is made to move the point.

The move function looks like this:

```
def move(point, x, y):  
    point.setpoint(x, y)
```

This function is automatically called when we try to move the point. The arguments to the method are the point we are attempting to move and the coordinates `x` and `y` to which we are trying to move it. The reason the point is an argument is that we may have many moveable points in a figure and we may want to do the same thing to each of them when moved. This way, we do not have to keep explicit track of which point is being moved. The body of the function simply updates the coordinates of the point. The coordinates `x` and `y` are in the coordinate system in which the point is drawn.

There are a few parameters we may set that describe how the point is to be drawn. For instance, we may choose:

```
redpoint.setstyle(Moveablepoint.SQUARE)
```

which causes the point to appear as a square (a circle is the default). Other parameters are listed in an appendix and will be clear in the next example.

We also need to register the point with the `JyPanel` so that the panel can send mouse clicks to the point:

```
panel = JyPanel(200, 200, draw)
panel.addmoveable(redpoint)
```

Finally, we may use the point's coordinates in the drawing function:

```
p.box(0, 0, redpoint.x, redpoint.y)
```

and we draw the point using the panel's `placemoveable` function

```
p.placemoveable(redpoint)
```

The coordinate system in place at the time of this function call determines how the coordinates `x` and `y` are interpreted in the move function.

Here's the entire code:

```
from JyScript import *
from javax.swing import *
def draw(p):
    p.beginpage()

    p.scale(50)
    p.translate(1, 1)
    p.newpath()
    p.box(0, 0, redpoint.x, redpoint.y)
    p.fill(0,0,1)
    p.setlinewidth(2)
    p.stroke()

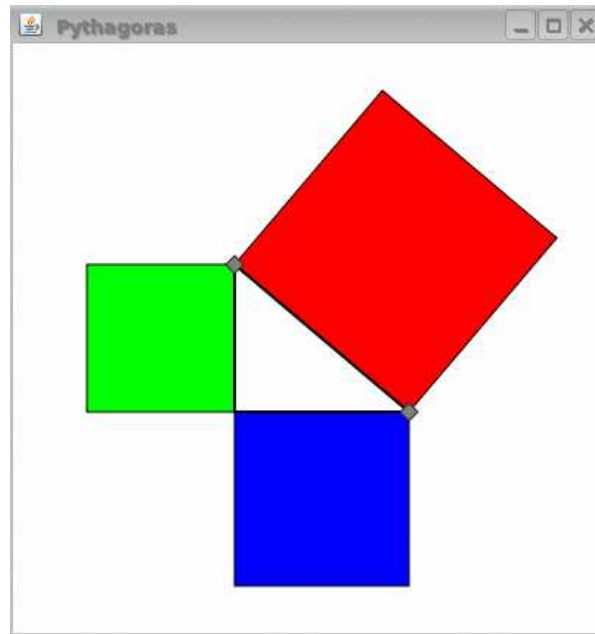
    p.newpath()
    p.placemoveable(redpoint)
    p.endpage()

def move(point, x, y):
    point.setpoint(x, y)

redpoint = Moveablepoint(1, 2, move)
redpoint.setstyle(Moveablepoint.SQUARE)

frame = JFrame('Moveable box')
panel = JyPanel(200, 200, draw)
panel.addmoveable(redpoint)
frame.getContentPane().add(panel)
frame.pack()
frame.setVisible(True)
```

In the next example, we use two `Moveablepoints`, which are constrained to move either horizontally or vertically. This example also illustrates the many attributes of `Moveablepoints` that may be set.



We create our panel and Moveablepoints like this:

```
panel = JyPanel(400, 400, draw)

p1 = Moveablepoint(1, 0, horizontal)
p1.setFillcolor(0.5)
p1.setStrokecolor(0.2)
p1.setsize(6)
p1.setStyle(Moveablepoint.DIAMOND)
panel.addmoveable(p1)

p2 = Moveablepoint(0, 2, vertical)
p2.setFillcolor(0.5)
p2.setStrokecolor(0.2)
p2.setsize(6)
p2.setStyle(Moveablepoint.DIAMOND)
panel.addmoveable(p2)
```

Here are the functions that move the points

```
def horizontal(m, x, y):
    if x < 0: return
    if x > 3: x = 3
    m.setpoint(x, 0)
def vertical(m, x, y):
    if y < 0: return
    if y > 3: y = 3
    m.setpoint(0, y)
```

Finally, the points are used inside the draw function as

```
def draw(p):
    global p1, p2
    p.beginpage()

    p.antialiasing(True)
    p.purestroke(True)

    p.scale(50)
    p.translate(3, 3)

    a = p1.x
    b = p2.y
    c = math.sqrt(a*a + b*b)
    ...
    p.endpage()
```

8: Writing images

We may save a figure in a `JyPanel` to either a PNG or JPEG file. To create a PNG file, simply use

```
panel.writePNG("box.png", 2)
```

where the argument 2 represents a scaling factor. For instance, if the dimensions of the `JyPanel` are 200×200 pixels, the dimensions of `box.png` will be 400×400 . By using a large scaling factor, we can create high-resolution images.

To create a JPEG file, use

```
panel.writeJPEG("box.jpg", 2, 0.5)
```

where the argument 2 represents the scaling factor and 0.5 represents the compression factor.

9: Creating applets

So far, we have used JyScript to create new windows that pop up on our screen. To distribute an illustration or animation widely, we may wish to post it on the web as an applet that may be loaded into a web browser. This involves a slight modification, which we'll now describe.

We will demonstrate by converting our first program, `Box.py`, to an applet. The first change is that we need to create a class, subclassing Java's `JApplet`. Everything we write will go into this class. To do this, we begin with

```
from JyScript import *
from javax.swing import *
class BoxApplet(JApplet):
```

which announces our intention to create an applet called `BoxApplet`. One thing is important here: Java requires that the name of the file in which we write our code agree with the name of the class, so we write into `BoxApplet.py`.

When we run applets in a web browser, a frame is provided by the browser so we no longer need a `JFrame` as we did previously. Any initialization should go into the applet's `init` function:

```
def __init__(self):
    panel = JyPanel(200, 200, self.draw)
    self.getContentPane().add(panel)
```

Here is the `draw` function:

```
def draw(self, p):
    p.beginpage()

    p.newpath()
    p.box(40, 40, 120, 120)
    p.fill(0, 0, 1)
    p.setlinewidth(2)
    p.stroke()

    p.endpage()
```

Surprisingly, that's all the code we need to write. Since we are writing a class, any references to functions and variables we define in the class should be preceded by `self`. Forgetting this is probably the most common and aggravating mistake you will make.

Here is the entire file `BoxApplet.py` alongside the original `Box.py`.

```
from JyScript import *
from javax.swing import *
class BoxApplet(JApplet):
    def __init__(self):
        panel = JyPanel(200, 200, self.draw)
        self.getContentPane().add(panel)
    def draw(self, p):
        p.beginpage()
        p.newpath()
        p.box(40, 40, 120, 120)
        p.fill(0, 0, 1)
        p.setlinewidth(2)
        p.stroke()
        p.endpage()

from JyScript import *
from javax.swing import *
def draw(panel):
    panel.beginpage()
    panel.newpath()
    panel.box(40, 40, 120, 120)
    panel.fill(0, 0, 1)
    panel.setlinewidth(2)
    panel.stroke()
    panel.endpage()
frame = JFrame('First example')
jypanel = JyPanel(200, 200, draw)
frame.getContentPane().add(jypanel)
frame.pack()
frame.setVisible(True)
```

We now need to turn this into Java byte code using the Jython compiler `jythonc`. Debugging code compiled with `jythonc` can be rather tedious since the error messages are not as explicit as with `jython`. For this reason, to test and debug an applet, it is often more convenient to simply place the applet into

a `JFrame` and run the applet with `jython`. For instance, after the definition of the `BoxApplet` class, append

```
frame = JFrame()
boxapplet = BoxApplet()
frame.getContentPane().add(boxapplet)
frame.pack()
frame.setVisible(True)
```

and then use `jython BoxApplet.py`.

Remove these last lines when you have a working version and create a `jar` file using `jythonc`:

```
jythonc -jboxapplet.jar -c BoxApplet.py
```

The applet may be included on a web page by placing an applet tag in the HTML code:

```
<applet
  code=BoxApplet
  width=200 height=200
  archive=boxapplet.jar>
</applet>
```

We may now run the applet by loading the HTML page into a browser. An intermediate level of testing may be performed by using Java's `appletviewer`. For instance, if `boxapplet.html` is the name of our HTML file, we may use

```
appletviewer boxapplet.html
```

So that they do not pose a security threat when running in someone's browser, applets have some restrictions, such as not being able to read and write files.

10: Free advice!

Here is a collection of suggestions starting with the more technical.

- In dynamic illustrations, don't forget to `repaint`.
- In a dynamic illustration that requires a lot of calculation, put as much of the calculation in the function that calls `repaint` rather than in the function that does the drawing. The drawing function will be called at times you might not expect, such as when the window is raised to the front of the desktop. The best strategy is therefore to separate calculations from drawing.
- Those with some experience with graphics in Java may want access to the `JyPanel`'s graphics object. It is available with the `getgraphics()` command.
- Provide reset buttons. It's easy to mess up illustrations, sometimes due to the programmer failing to foresee some user actions, so give the user a way back to the original state.
- Use visual cues rather than text. Experience shows that relationships between elements in a figure that are expressed with visual cues, such as color and shape, are more easily detected than when text is used.

- In a series of illustrations, use color consistently and with purpose. For instance, color points that move red and points that follow another point's motion gray.
- As figures become more complicated, use muted colors or grays for peripheral elements in an illustration. For instance, when creating a figure that illustrates a construction, it can be effective to have elements left over from earlier parts of the construction fade into the background. One way to achieve this is with a `JyTimer` that modifies the alpha values of the colors you are using.
- Provides less choice for the user rather than more. There are many reasons that you might want to create a dynamic illustration, from communicating a mathematical idea to a new audience to exploring a new concept for your own understanding. Generally speaking, an illustration should have a focus. Make sure that the user's choices are so few that he is unable to avoid the point you are trying to make. Simple animations are often effective in helping achieve this goal.

Appendix A: Installing Jython and JyScript

To install Jython, you will first need to install a version of the Java Developer's Kit, available at <http://java.sun.com/javase/downloads/index.jsp> and add the `bin` directory to your executable path.

Jython may be downloaded from <http://www.jython.org>; installation instructions are also available from that page. This can be as simple as typing

```
java -classpath . jython-21
```

in a command window with the current directory set to the one in which you extracted the jython files.

Finally, you will need to get the JyScript files, available in a zip file from

```
http://merganser.math.gvsu.edu/david/jyscript
```

and extract them into a directory. You then need to edit the jython registry so that the `python.path` variable points to this directory. On my installation, JyScript is installed in

```
/home/david/python/jyscript
```

and jython is in

```
/home/david/jython2.2.1
```

I edited one line in the file

```
/home/david/jython2.2.1/registry
```

to read

```
python.path = /home/david/python/jyscript
```

Note that you will probably need to remove the comment ("`#`") before `python.path`.

Appendix B: JyPanel commands

General

- `beginpage()`
- `endpage()`
- `width()`
- `height()`
- `getgraphics()`
- `setdeg()`
- `setrad()`
- `todegrees(a)`
- `toradians(a)`
- `gsave()`
- `grestore()`

Drawing

- `moveto([x,y])`
`moveto(x,y)`
- `rmoveto([x,y])`
`rmoveto(x,y)`
- `lineto([x,y])`
`lineto(x,y)`
- `rlineto([x,y])`
`rlineto(x,y)`
- `curveto([x1, y1], [x2, y2], [x3, y3])`
`curveto([x1, y1, x2, y2, x3, y3])`
`curveto(x1, y1, x2, y2, x3, y3)`
- `rcurveto([x1, y1], [x2, y2], [x3, y3])`
`rcurveto([x1, y1, x2, y2, x3, y3])`
`rcurveto(x1, y1, x2, y2, x3, y3)`
- `quadto([x1, y1], [x2, y2])`
`quadto([x1, y1, x2, y2])`
`quadto(x1, y1, x2, y2)`
- `rquadto([x1, y1], [x2, y2])`

- rquadto([x1, y1, x2, y2])
- rquadto(x1, y1, x2, y2)
- arc([x,y], r, a1, a2)
arc(x, y, r, a1, a2)
- arcn([x,y], r, a1, a2)
arcn(x, y, r, a1, a2)
- circle([x, y], r)
circle(x, y, r)
- closepath()
- newpath()
- currentpoint()
- box(w, h)
box(llx, lly, w, h)
- graph(f, x0, x1): Graph of $y = f(x)$
graph(f, x0, x1, N)
graph([f, g], t0, t1): Parametric curve $x = f(t), y = g(t)$
graph([f, g], t0, t1, N)
- arrow(): See the PiScript manual for argument list
- quadarrow(): See the PiScript manual for argument list
- stroke(): Stroke with linewidth in pixels
stroke(gray)
stroke(r, g, b)
stroke([r, g, b])
stroke(r, g, b, alpha)
stroke([r, g, b, alpha])
- tstroke(): Stroke with linewidth in current coordinate system
tstroke(gray)
tstroke(r, g, b)
tstroke([r, g, b])
tstroke(r, g, b, alpha)
tstroke([r, g, b, alpha])
- fill()

fill(gray)

fill(r, g, b)

fill([r, g, b])

fill(r, g, b, alpha)

fill([r, g, b, alpha])

- clip():
- antialiasing(boolean)
- purestroke(boolean)

Coordinate changes

- center()
- scale(s)
scale(sx, sy)
- translate([x, y])
translate(x, y)
- rotate(angle)
rotate([x, y], angle)
rotate(x, y, angle)
- concat([m00, m10], [m10, m11], [m20, m21]): compose this affine transformation with the current transformation
concat([m00, m10], [m10, m11])

Drawing attributes

- setcolor(r, g, b)
setcolor(r, g, b, a)
setcolor([r, g, b])
setcolor([r, g, b, a])
- setcomposite(factor)
- restoredefaultcomposite()
- setgray(gray)
- scalelinewidth(factor)
- setlinewidth(width)
- setlinejoin(c)
- setlinecap(c)
- currentlinecap()

- `currentlinejoin()`
- `setmiterlimit(miterlimit)`
- `setdash(array, phase)`

Text

- `setfont(size)`
`setfont(name, size)`: where `name` = "serif," "sanserif," or "monospaced"
`setfont(name, style, size)`: where `style` = "plain," "bold," or "italic"
- `shift(self, factor, string)`: move the current point horizontally by `factor` times the width of the string.
- `show(string)`: display `string` at the currentpoint
- `centershow([x,y], string)`
`centershow(x, y, string)`

Moveables

- `placemoveable(moveablepoint)`
- `addmoveable(moveablepoint)`
- `removemoveable(moveablepoint)`

Writing images

- `writePNG(filename, scalefactor)`
- `writeJPEG(filename, scalefactor, compressionfactor)`

Appendix C: Moveablepoint commands

- `setpoint(x, y)`
- `setsize(size)`: the size is given in pixels
Default is 4.
- `setstyle(style)`
`style = Moveablepoint.CIRCLE, Moveablepoint.SQUARE, Moveablepoint.DIAMOND`
Default is `Moveablepoint.CIRCLE`.
- `setfilled(boolean)`
Default is `True`.
- `setfillcolor(color)`
Default is `(1, 0, 0)`.
- `setstroked(boolean)`
Default is `True`.
- `setstrokecolor(color)`

Default is (0, 0, 0).

- `setstrokesize(size)`

Default is 1.