The Beauty and Joy of Computing

Lab Exercise 10: Shall we play a game? (Python version)

Note: This is a little sloppy and needs a good proof-reading and editing. We aren't using this in the course this semester, so it's really just here as a draft that people might find interesting or useful.

Objectives

By completing this lab exercise, you should learn to

- Understand and work with layering of multiple sprites (applied to make a side-scrolling background);
- Understand the basics of object-oriented programming and use in Python;
- Program self-cloning "projectiles" for use in a game; and
- Merge separately developed programs into one larger program to support development by teams.

Background (Pre-Lab Reading)

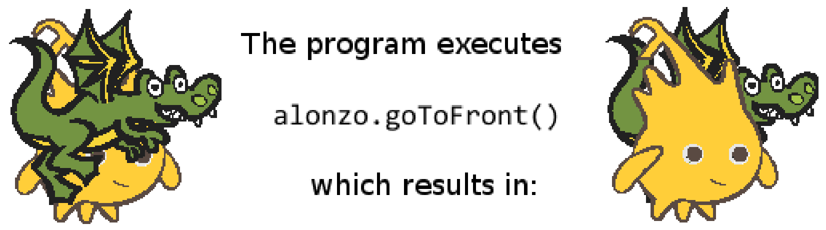
You should read this section before coming to the lab. It describes how various things work in Python/SnapPy and provides sample code and pictures. You only need to read this material to get familiar with it. There is no need to actually do the actions described in this section, but you certainly can do them if it would help you follow along with the examples.

The purpose of this lab is to introduce a few new concepts and techniques that are important for developing larger projects, and games in particular, in Python with SnapPy. These are techniques that will be useful in many of your final projects, so are important to know and understand!

Image Layers

In the activities and examples we've done up until now, sprites have been separate and have not overlapped - what if that is not the case? For example, in the very first lab we added the dragon sprite to the initial Alonzo sprite, and both sprites started in the center of the screen and had to be dragged apart. But before we separated them, how did BYOB decide whether Alonzo or the Dragon should be displayed in those locations where they overlap? This uses the concept of **layers**, which is common now in graphics programs as well as in the drawing tools of programs like PowerPoint or Word. You can imagine each sprite being in a specific layer - with higher layers obscuring lower ones. In both BYOB and SnapPy, each sprite is in a layer by itself that has a specific position relative to all other sprites, and when sprites are created they always are created in a new "top" layer - as a result, there is never any ambiguity about which sprite is visible. So the answer to the question of whether Alonzo or the dragon is shown is "whichever one is in the higher layer." Layers are not fixed, and can change based on program actions. The functions used for this are named `goToFront()` and `goBackLayers(num)`.

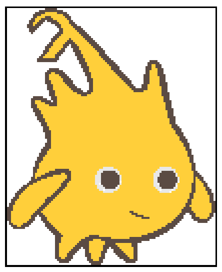
If a sprite executes the `goToFront()` function, it will move to the very top layer and be displayed over any other sprites that overlap with it. For example, when the dragon is added to the base project with Alonzo it is put on top of Alonzo, and if Alonzo later executed the `goToFront()` function it would be put on top of the dragon. We can illustrate that as follows:



The `goBackLayers(num)` function does the opposite, but is more fine-grained since you can indicate a number of layers to send the sprite back, staying on top of some sprites while moving behind some others. To send a sprite to the lowest possible layer you can just use a large argument like 1000, which will max out the layer so that it is behind everything else (as long as the number of layers to go back is greater than or equal to the total number of sprites).

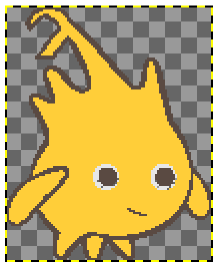
Transparent Pixels

Images are always stored as rectangular areas. For example, the default Alonzo is 104 pixels wide by 128 pixels high. Here's a picture of Alonzo, with the outline of the rectangular area that is the actual Alonzo image marked:



There are a lot of pixels in this rectangular area that are not part of Alonzo, because he's not shaped like a rectangle! Recall that every pixel in a color image is typically a color made up of red, green, and blue components, so what is the color of the pixels that are outside of Alonzo but inside the rectangle above, like the pixels toward the upper right of the rectangle? They look white in this picture, but look at the image above with Alonzo on top of the dragon. If these pixels were white, and Alonzo were on top of the dragon, then there would be a lot of white are around Alonzo where we are actually seeing the dragon instead. On the other hand, look at the white around Alonzo's eyes, and notice that this really is white and does not show the dragon through from underneath.

The reason why these two areas of appear different is that the pixels outside Alonzo are not actually white pixels. Outside Alonzo, the pixels are actually transparent, and they only *appear* white in the rectangle above because the white "paper" (or screen background) is showing through. Picture editors have a more-or-less standard way of representing transparent pixels when you are working with images, and it looks like a black and grey checkerboard pattern. This is what Alonzo looks like when I bring up that image in an image editing program:



The way this is actually represented is with something called an “alpha” channel, which is just another component for each pixel along with the red, green, and blue intensities. For an image format in which each channel is 8 bits, an alpha channel value of 0 means that the pixel is completely transparent, and an alpha channel value of 255 means that the pixel is completely opaque (so that all you see is the color defined by the red, green, and blue components). You can also set the alpha channel to values in between, so for example with an alpha value of 127 you would see the color defined for that pixel, but you it would be lightened and some of the image behind it will come through. So the difference between the white around Alonzo’s eyes and the white in the upper right corner of the rectangle is entirely because the alpha channel value is different in those locations.

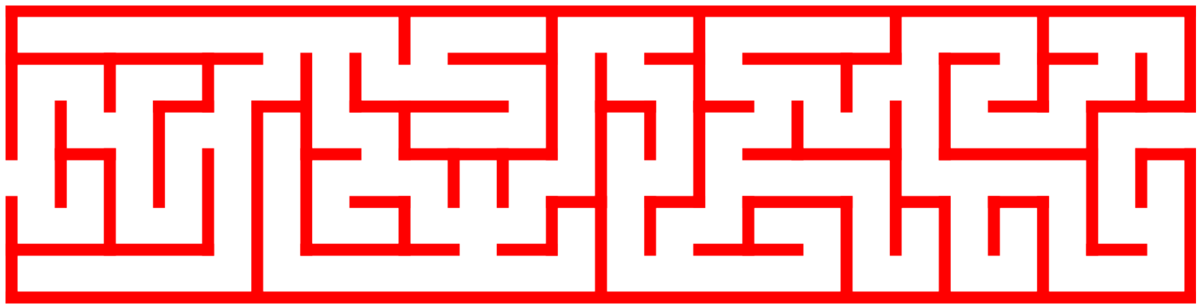
Alpha channels are very important in games for two reasons. First, when you think about layers, as discussed above, when the rectangles for two sprites overlap, what you see from the lower layers depends entirely on the alpha channel value in upper levels: how transparent are the upper-level pixels? The second reason the alpha channel is important is because it defines the actual area of a character or sprite: there is an assumption that transparent areas do not actually correspond to the object represented by a sprite, so when SnapPy tests whether two sprites collide with one another it actually tests specifically whether non-transparent areas of the costumes overlap. If a sprite only overlaps with transparent areas of a sprite’s costume, then it is not detected as a collision.

Note that the PyGame system that is underneath SnapPy is more powerful than this discussion suggests. In PyGame each sprite has a “mask” associated with it, which can be any arbitrary part of the sprite’s image. While SnapPy always uses the non-transparent pixels as the sprite mask, since that’s the way BYOB/Snap work, in PyGame you can make the mask larger or smaller than this without any problem. There are special cases in game design where this additional flexibility might be useful.

Side-Scrolling Background

Most of you are probably familiar with side-scrolling video games, in which the background slides left and right behind a character as it progresses through the game. However, SnapPy follows the BYOB design, in which backgrounds are fixed and cannot be moved, which creates a challenge. So while we think of this as a scrolling background we actually use a sprites as the background, making sure the background sprite is in the bottom layer so that characters will appear on top of the background sprites. Sprites can be moved around and even placed partially off the stage, which is exactly what we need to create a side-scrolling game.

The first step in creating a side-scrolling effect is to create the background. We want to fill up the stage from top to bottom, and so the background should be 360 pixels high. The width is really determined by your needs, and in this example we will have a background that is equal to 3 full stage widths. Therefore, we create the following 1440 pixel wide image for a maze game:



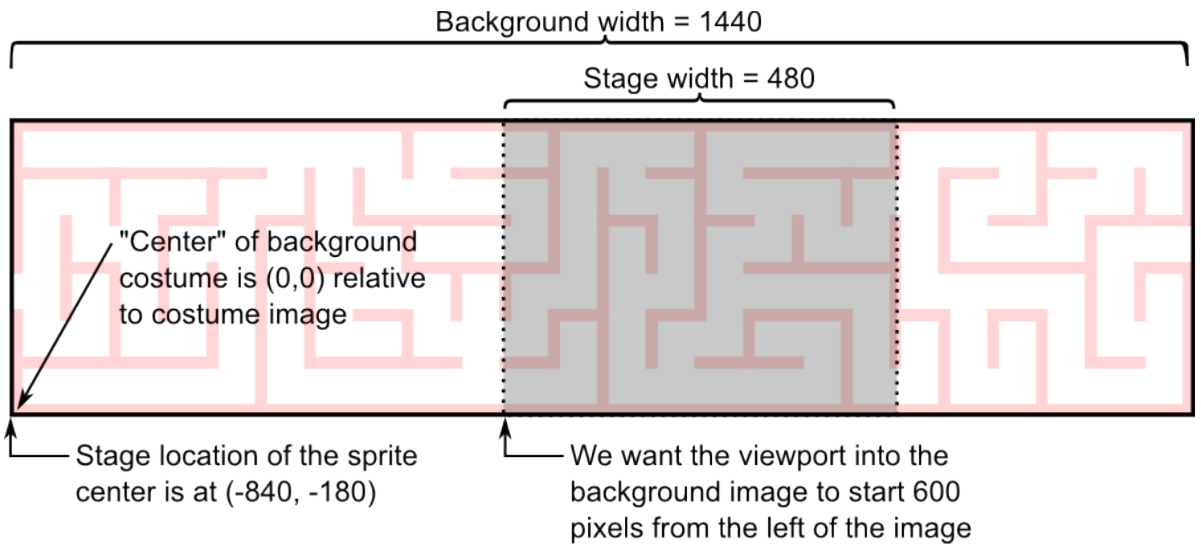
Since positioning the background sprite must be precise, we need to address a small concept that we haven't discussed: the unfortunately-named "**center**" of a sprite[1]. When we say to put a sprite in a particular (x,y) location, what does that mean? Does the lower-left corner of the sprite get placed at (x,y)? The center of the sprite gets placed at (x,y)? The answer is that BYOB allows us to define any point we want as a reference point in a costume, and that point gets placed at (x,y). Since this spot on the sprite is the one part that stays in the same location even when the sprite rotates, it is the "center of rotation" of the sprite - that's the meaning for the name, even if it seems a little strange at first. You can set the costume "center" for any costume by calling costume.setCenter(pos). For example, we can load the maze costume from a file and set the "center" to (0,0), which is the lower-left corner of the image, using the following sequence of operations:

```
bg_costume = snappy.CostumeFile('Maze.png')
bg_costume.setCenter((0,0))
```

Now if this costume is used by a sprite, and we set the sprite location to (x,y), then the lower left corner of the Maze.png image will be placed at (x,y). If we want to start with the far left of the background displayed so that we can move through our maze going to the right, then you would initialize the background sprite so that the lower left corner of the costume (our "center," remember?) is at the lower-left corner of the stage, which is position (-240,-180) by doing "`background = snappy.Sprite((-240,-180), bg_costume)`".

You can view a scrolling background as a viewport into the larger background image. Consider an example where we have the background image from above, which is 1440 pixels wide, and we want to display the part of this background that starts 600 pixels from the left onto the 480-pixel wide stage. That might look something like this:

[1] To keep the correspondence with BYOB/Snap, we kept the same name in SnapPy.


Background width = 1440
Stage width = 480
"Center" of background costume is (0,0) relative to costume image
Stage location of the sprite center is at (-840, -180)
We want the viewport into the background image to start 600 pixels from the left of the image

By shifting the background sprite position to (-840,-180) from it's original position of (-240,-180) we have shifted the background to the left by 600 pixels. Side-scrolling then just consists of updating the x coordinate of the background sprite a little at a time. For example, if you change the x coordinate of the background sprite by -10 (using `background.changeXBy(-10)`) then the background will slide to the left 10 pixels - and note the the background shifting to the left will make it look like any character sprites don't move will appear to move to the right with respect to the background. Similarly, using `background.changeXBy(10)` will slide the background to the right 10 pixels.

One final thing to consider when creating this background sprite, related to the "Layers" discussion above: You want to make sure that the background is behind all the other sprites. The easy way to do this is to just send the background to the back after all sprites are initialized, using something like "`background.goBackLayers(1000)`" (assuming you don't have more than 1000 sprites!).

Object-Oriented Programming Concepts and Terminology

In the previous labs, we have worked with a small number of sprites, and every sprite was specifically created and programmed through scripts that operated on that sprite. Scripts can create interesting and complex actions for individual sprites, but we often want to have many sprites. Think about an interesting game: there are often tens or hundreds or even thousands of characters that you must keep track of. In addition to characters, we might also have other game items such as weapons, clothing, buildings, that we also need to track. We obviously don't want to create a specific sprite for each entity in a game in which there are thousands of characters, and what saves us is the fact that many items share a similar structure and set of actions that they can perform. A style of program design that handles this very nicely is called **object-oriented programming**. While a full explanation of object-oriented design is beyond what we will do in this class, the basics are easy to understand.

The name “object-oriented programming” actually refers to several different styles of programming. While Java and JavaScript are both object-oriented languages and have very similar names, the style of object-oriented programming that each uses is different. Java is a class-based object-oriented language, while JavaScript is a prototype-based object-oriented language. These names may not mean a lot to you now, but we’ll return to them after we discuss some of the basic concepts. Since Python, like Java, is a class-based language, we focus primarily on that style in this lab.

Object-oriented programming is organized around **classes**, which are programming elements that share certain characteristics. Classes might represent things that you could potentially see, like characters in an animation, players or weapons in a game, or bigger program components such as backgrounds. However, classes can also represent more abstract and non-visible things like fractions or matrices or student lists. In a fighting game we might have a class for something as general as weapons, or for something slightly more specific like daggers which we might refer to as the “Dagger” class. **Objects** are specific instances of a class, so the dagger that your player is holding could be an instance of the Dagger class. If there is another player holding a different dagger, then that might be a different instance of the same general class. In SnapPy, every sprite is an object from a general “Sprite” class, and when you create a specific sprite (such as Alonzo) you have just created an instance of the sprite class. Note that you can create multiple sprites that look exactly alike - multiple Alonzos, for instance - but these are in fact separate objects.

It object-oriented programming, every object can contain attributes and methods:

- An **attribute** is a variable that is associated with one particular object. For example, an object from the “Dagger” class might have attributes such as weight, size, and sharpness. The collection of all attribute values of an object is referred to as the **state** of the object.
- A **method** is a function or a script that causes an object to perform some action. A method can access or change the attributes associate with this object, or it can perform some other action. For example, our Dagger class might have methods such as throw or stab, reflecting game play actions, or a draw method that draws it on the screen. A special kind of method called a **constructor** defines the actions required to initialize all of the attributes of an object - in Python, when an object is created the constructor creates attributes for the object by simply assigning initial values to them.

Consider a SnapPy sprite: the attributes for a sprite object include things like its current position (x and y coordinates), direction, and whether it is visible, and the methods include functions that operate on sprite objects, such as the `move()` and `hide()` functions. You’ve actually been using objects and methods all along, but we didn’t refer to it that way. For example, if you used `alonzo.hide()` in a program, this is running the `hide()` method for the sprite instance named `alonzo`.

Returning to the two types of object-oriented programming, in **class-based programming** the programmer defines the class, and there are no objects that are instances of the class until one is explicitly created. For example, even though the Sprite class is defined in SnapPy, with methods defined that can initialize and work with Sprite objects, there are no actual Sprites until one is created by some other code. In contrast, prototype-based programming is based on the idea that new objects are created by cloning existing objects. An object that is used primarily for the purpose of cloning other instances of this class is called a prototype object, which is where this style gets its name. In your earlier work with BYOB, you may have noticed that there is a “clone” block - this is because BYOB is a prototype-based language. There’s obviously a lot more to object-oriented programming than has was just described - we’ll talk about a few other issues in the Post-Lab Reading, but for now let’s look at how the basics work by working through an example.

Working with Objects

In this section, we will go through a complete example showing how to set up a class with both attributes and methods, and create objects that are instances of that class. For this example, let’s make a silly animation: we want cars to drive around on the stage, bouncing against the edges. A new car is created every time a car runs over a magic hat that moves randomly around the stage, and there are two bombs on the stage that also move around randomly and any car that runs into a bomb is destroyed. Since the number of cars is controlled by luck (where the hat and bombs are randomly placed), we don’t know how many cars we need, so we can’t just create all the sprites manually at the beginning. Just because they are all cars and share some characteristics doesn’t mean they will all look the same. There are a few different “styles” of cars, and the style of a car is randomly chosen when it is created and stored in an attribute of the car object.

From an object-oriented programming standpoint, we want to define a class for cars, and then create a new car object both at the beginning of the animation and every time an existing car hits the magic hat. Each car object will have all of the attributes that are common to all sprites (position, direction, etc.), and will have two attributes that are specific to our car definition: a speed and a style. In other words, a “Car” extends the notion of a “Sprite” by adding additional properties, or attributes. We will look at the specific Python code to do this later, but our needs should be clear: We need to be able to declare a new class for cars, and indicate that it is extending the notion of a Sprite, and then we need a constructor that will set up a car object with the appropriate costumes, size, random initial position and direction, and the car-specific attributes for speed and style.

Here’s the code that does this in Python - we’ll describe in detail how this works below the code.

```
class Car(snappy.Sprite):
    def __init__(self):
        snappy.Sprite.__init__(self, (0,0), snappy.CostumeBuiltIn('car2'), hidden=True)
        self.loadCostume(snappy.CostumeBuiltIn('car-cow'))
        self.loadCostume(snappy.CostumeBuiltIn('car-blue'))
        self.loadCostume(snappy.CostumeBuiltIn('car-bug'))
        self.setSize(60)
        self.setRotStyle(0)
        self.goTo((random.randint(-139,139), random.randint(-133,133)))
        self.pointInDirection(random.randint(0, 360))
        # Car-specific attributes are speed and style
        self.speed = random.randint(5,10)
        self.style = random.randint(0,3)
        self.switchToCostume(self.style)
        self.show()
        snappy.launch(self.move_car_forever)
```

Let’s walk through this code line by line. The first line is what tells Python that you are defining a class - this always starts with the word “class,” followed by the name you have selected for the class you are defining (we are defining the “Car” class), and if this class is extending an existing class then you put the class being extended in parentheses (our Car class is extending the snappy.Sprite class).

Underneath the “class” line are methods that operate on the objects in that class. All these definitions are indented underneath the “class” line so that it is clear that they are part of the class. All methods in a class should have a first parameter named “self” which will always refer to the object that is being operated on by that method. In other words, if we call “alonzo.hide()” then Python will actually pass the “alonzo” object as the “self” parameter, and then we can access Alonzo’s attributes in our methods using notation like self.position. The only method we’re shown above is a definition for the oddly-named “__init__” function. This is the constructor, whose job it is to initialize any new object, and regardless of what you name your class or its attributes, the constructor will always be named “__init__”.

Now let’s look at the code in the __init__() constructor: It starts with a line that is a call to snappy.Sprite.__init__(), so what is this? Well snappy.Sprite is the class we are extending, so our new Car object is going to be snappy.Sprite object with some additional things added, and this is a call to the snappy.Sprite constructor to initialize all the attributes of the object that are common to all sprites. After this call returns, the next 7 lines of code adjust the Sprite properties to add additional costumes, set the size and rotation style, and then set the position and direction to random values. The line starting with the number sign (or hash tag, if you prefer to call it that) is a comment: it doesn’t actually do anything, but is just a note left to make the code easier to understand. The next line creates and initializes the “speed” attribute of the car. Note that we refer to it as “self.speed” since it is the speed attribute of the current object, which is called “self.” Similarly, the next line creates and initializes the style attribute to one of 4 possible styles. Next, we change the costume according to the style attribute and make the car visible by calling the show() method. Finally have this last line that calls the “snappy.launch” function, which takes a function name as an argument and starts that function running on its own. Notice that I said “takes a function name as an argument” - that means that self.move_car_forward is a function, so what is it? This is another method defined for the Car class, shown below:

```
    def move_car_forever(self):
        while True:
            self.move(self.speed)
            self.if_on_edge_bounce()
            if self.touching(hat):
                self.goTo((random.randint(-139,139), random.randint(-133,133)))
                Car()
            elif self.touching(bomb1) or self.touching(bomb2):
                self.kill()
                return
```

Note that this is underneath the constructor shown earlier, and indented so that it is still part of the “Car” class. Since it is another method in the Car class, its first (and only in this case) parameter is self. So the snappy.launch call that we had at the end of the constructor starts this function running so that the car moves.

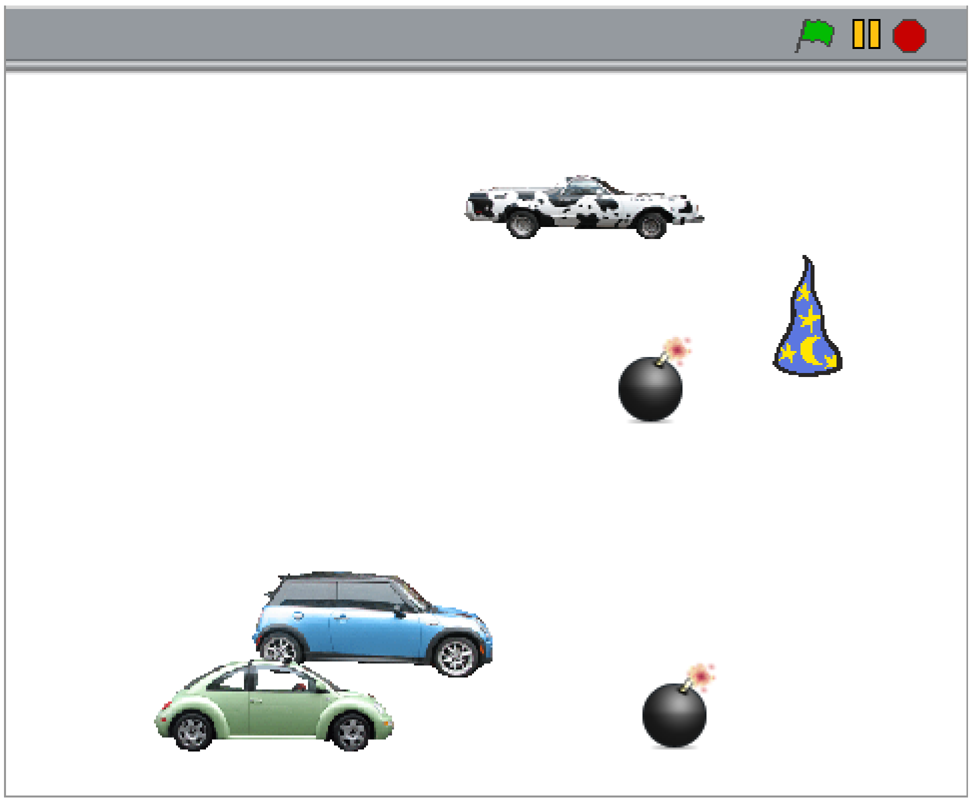
Again, we can look through this line-by-line: The “while True:” loop is the Python version of the BYOB “forever” loop. So while looping forever the car moves itself some distance (determined by the speed attribute), it bounces if it hits the edge of the stage, and then has actions defined if it ends up touching the hat or one of the bombs (these sprites are defined elsewhere). If it hits the hat, it jumps to a new random location and then calls Car() - this call, using a class name as if it were a function to call, is how you create an object of that class and initialize it using the constructor. In other words, in addition to moving our existing car to a new location, we also create a brand new car, which will initialize itself by placing it at a random location with a random direction, speed, and style.

Finally, when we start the animation we need to create the first car. That’s simple, and it just means that we include the following code:

```
@snappy.startOnGreenFlag()
def startCar():
    Car()
```

In other words, when the green flag is clicked, we call Car() to create the first car object and set it in motion.

The full code for this example is available on the class web page in the “Lab Exercises” section. You should download that code and read through it carefully. It’s important that you fully understand this example! When it is run, the animation might look like this after running for a little while:



Combining Projects

Almost all major software development is done by teams of developers, and not by a single person. Coordinating the work of a team is not easy, but it is a skill that's developed with lots of practice. Obviously, a team wouldn't work very well if there was only one copy of a program and one computer that could be used for development - while that was OK for the simple pair-programming activities we've done in these labs, for large projects its best if everyone can work independently on a piece of the project and then combine their pieces into a single large program later. All "version control systems" (a concept mentioned Lab 1) provide support for merging programming done by different team members, and while we could certainly use a version control system for our Python code that is a little heavy-handed for this class.

For this class, we'll just combine projects using cut-and-paste, but to do this you should plan ahead for it. In particular, you should make sure that all of the code for a particular sprite is in a contiguous region of the program file. While this isn't necessary for the code to work properly, planning ahead like this will allow you to copy all of the code for a specific sprite out of one program and paste it into another. Another (and actually much better) option is to keep code for separate parts of the program in separate files, and then use "import" to include a file developed by one team member in the main program written by a different team member. You'll experiment with the simpler cut-and-paste technique in the activities below.

Self-Assessment Questions

Use these questions to test your understanding of the Background reading. Answers can be found at the end of the lab exercise.

1. When you access an attribute, you always include the name of an object. For example, you might say bluecar.speed to get the speed of an object named bluecar. Why is it important go give the name of an object? What would happen if, in our animation, "speed" was a global variable rather than tied to an object?
2. Consider defining a "Fraction" class, where each object is a fraction with an integer numerator and integer denominator. What are some attributes and methods that this class might contain? Note that there's no single right answer here - see what you can think of!

Activities (In-Lab Work)

Activity 1: For the first activity, you are to implement the side-scrolling technique described in the pre-lab reading. The 1440-pixel wide maze background (Maze.png) from that discussion is available on the class web site along with this lab write-up, so your first step is to download that to your computer. Next, start a SnapPy program named Lab10-Maze.py where you use the techniques described in the pre-lab reading to load this image in as a costume, set the costume center, and position it so that you are looking at the leftmost 480-pixel wide viewport into the background image.

Once you have the background sprite set up, add scripts that run when the left and right arrows are pressed that will shift the background in the appropriate direction. Experiment with different shift amounts (10 pixels? 5 pixels?) so that the motion looks smooth. The only requirement is that the speed at which you move to the right is the same as the speed at which you move to the left.

If you have done this correctly then you should be able to scroll the maze left and right with arrow keys. That's not very exciting right now, but hopefully you can see the potential! Once you've got this working, everything should be saved in Lab10-Maze.py.

Activity 2: For this activity, you'll complete a basic maze game using the scrolling background created in Activity 1. You should add code to create an Alonzo sprite in addition to your background sprite, and immediately set the size of Alonzo to 28%. The maze corridors are just the right size for Alonzo at 28%, so make sure you use the correct size!

Let's look at the Maze.png image more carefully by bringing it up in an image editor. Here's what a part of that image looks like in the editor:

See the checkerboard pattern? Remember what that means from the Pre-Lab reading? That means that those are “transparent” areas, and even though those pixels are inside the sprite’s image they are not considered part of the background sprite. This is actually different from BYOB, which doesn’t allow these interior “holes” to not be part of the sprite.

Why is this important? Because we can tell whether our Alonzo character overlaps a wall of the maze just by checking whether Alonzo “collides with” the background, which we can do in SnapPy with a call to “`alonzo.touching(background)`” - this returns True if Alonzo overlaps a wall, as illustrated below.

`alonzo.touching(background)`
returns `True`

`alonzo.touching(background)`
returns `False`

Of course, testing if Alonzo is inside a wall isn’t right - what we really want to do is make sure Alonzo never gets in the wall in the first place! This is one place where SnapPy has an extra feature that is not in BYOB. In BYOB you have to actually move Alonzo, check if he has collided with the wall, and then move him back quickly if the move put him inside the wall. But in SnapPy, we can check whether a certain move will cause a collision before we actually make the move by using a second, optional parameter to the “touching” method. If we want to see whether changing the x coordinate of Alonzo by “xchange” and the y coordinate by “ychange” will cause a collision with the background you can call the predicate `alonzo.touching(background, (xchange, ychange)`. So if you are moving 5 spaces at a time, and want to see whether moving Alonzo 5 spaces to the left would cause a collision, you would test whether `alonzo.touching(background, (-5, 0))` returns `True`.

So with that explanation, here is what you need to do to complete the maze program: When the left or right arrow are pressed, check whether moving Alonzo in that direction would cause a collision and if the movement does not cause a collision then you can shift the background in the appropriate direction. Similarly, when the up or down arrow is pressed, if the movement will not cause a collision then move Alonzo in the appropriate direction. Remember: horizontal movement (left or right) will cause the background to move, while vertical movement (up or down) will cause Alonzo to move. And in no case should the movement cause Alonzo to collide with the wall.

Once you've got this code written and running, you should be able to move Alonzo through the maze where the program forces Alonzo to stay in the passageways. Make sure everything is saved in Lab10-Maze.py, which is the file you will submit for Activities 1 and 2.

Activity 3: This activity does not build on the previous two, so start a new program named Lab10-Shooter and set it up with the basic snappy statements (import snappy at the top, and snappy.start() at the bottom). For this activity, you will create a program that shoots arrows, and in Activity 4 you'll merge this with another program that will turn this into a complete game. For this activity, you are to set up Alonzo and define an Arrow sprite that allows you to aim and shoot arrows. Start a new program (call it Lab10-Activity3.py), and add the code that will create a basic Alonzo sprite at location (0,-150) with Alonzo shrunk down to 30% size.

Next, create an Arrow class, as describe in the Pre-Lab reading - the Arrow class should be an extension of the snappy.Sprite class, and you should include a constructor. In making this constructor you will learn how to use an additional feature of the Python language, default values for parameters, and you will learn one new concept, that of creating a new object by cloning an existing object. Giving parameters default values is a very useful feature of Python: when you define the function, you can follow a parameter name with "=value" and when the function is called, if a value is not given for that parameter then the default value is assigned. We will use this technique to make a constructor for the Arrow class that can be used as a copy constructor: a special constructor that initializes a new object by copying all the values from an existing object. This is useful for us because we can create a new arrow that we can shoot, and initialize it from the aiming arrow so it is the same size, pointing in the same direction, etc. Here's what the constructor should look like:

```
def __init__(self, location=(0,0), clone=None):
    if (clone == None):
        snappy.Sprite.__init__(self, location, snappy.CostumeBuiltIn('Clock-hand'))
        self.setSize(20)
        self.pointInDirection(30)
    else:
        snappy.Sprite.__init__(self, proto=clone)
```

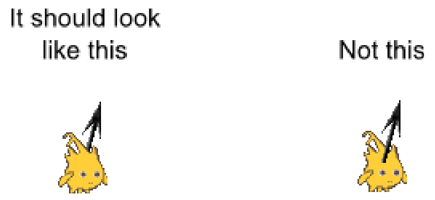
So if we call this and provide a non-empty "clone" argument, then it will make a copy of that object. Otherwise, it will create a new arrow at the given location and set the size to 20% and initial direction to 30 degrees. To create a new arrow object to use as an aiming object, you then you just do this:

```
aiming = Arrow((0,-150))
```

If you want to make a copy of the aiming arrow so that you can shoot it, you would do this:

```
shot = Arrow(clone=aiming)
```

You should add the first of these statements, to create the aiming arrow, to your program. Notice that the arrow and Alonzo will both be at the same location, so the issues of layers from the Pre-lab reading is important here. You want to make sure that alonzo is on top of the arrow, and not vice versa. Just use Alonzo's `goToFront()` function so that you get this result:



Now you need to add some scripts that rotate the arrow - you should rotate counterclockwise by 10 degrees when the left arrow is pressed and clockwise by 10 degrees when the right arrow is pressed. You should put in tests so that you never rotate beyond horizontal (in other words, the angle should always be between -90 and 90, inclusive).

Next, you need to make it so Alonzo can shoot an arrow. In particular, when the spacebar is pressed your program should use the copy constructor (code is above!) to create a new Arrow object, and then you'll write code (described below) that moves the arrow in the direction that it is pointing until it hits the edge of the stage, at which point it should destroy itself, for which using the friendly method named `kill()`.

As a first step, create a method in the Arrow class named "shoot" that will manage the arrow's motion and actions once it is created and shot. Then we can program an event handler that clone the aiming arrow and shoots this new arrow when the space key is pressed:

```
@snappy.startOnKey(' ')
def shoot():
    shot = Arrow(clone=aiming)
    shot.shoot()
```

So what should "shoot" do? This arrow should keep moving in the direction it is pointing until it reaches the edge of the stage, and then it should delete itself. Basically we want to move some number of steps (for example, 10 - this will control the speed of the arrow) inside a while loop. Looking ahead a little bit, there will be several conditions that will cause the arrow to stop moving - eventually we'll have targets, and need to stop when the arrow hits a target. This can lead to a really complicated condition for the while loop, so we use another fairly common loop pattern: the **loop until done pattern**. This pattern uses a Boolean variable to indicate when we are done processing and should exit the loop. This is a local variable that we will name "done" that should be initialized to False before the loop, and inside the loop we can set the variable to True when we detect that we should exit the loop. Here is the basic pattern, including the "move

10 steps” block to move the arrow and a block at the end to delete this arrow after the loop exits (when we’re done with it).

```
def shoot(self):
    done = False
    while not done:
        self.move(10)
        if self.touching_edge():
            done = True
        # Other tests and actions will go here!
    self.kill()
```

You should build all these scripts and test them out. Once everything is entered and saved, run the program and then click on the green flag. Does everything work OK? Hopefully so!

Once all of this is working, make sure all your work is saved as Lab10-Activity3.py.

Activity 4: Finally, we want to finish out our game by having something to shoot at. Start off this activity by doing a “Save As” in your program window using the name Lab10-Activity4.py to start a new program file. The real point of this activity is to think about how a multi-person team might work on a game like this. Pretend that you have divided the tasks for creating this game: your job (which you did in Activity 3!) was to initialize the game and make it so that Alonzo could aim and shoot arrows. My job is to create a dragon that flies around randomly, and includes a “die” script that gets executed every time it gets hit by an arrow. I have created a fancy script that starts the dragon at a random edge of the stage (left, right, or top), and has it fly around randomly until it is hit and dies. After developing this and testing it enough where I’m confident that it works, I made sure all of the code that could be used in the rest of the program is located together in the program file, and then I saved my program as Lab10-DragonCode.py - this file is on the class web page, and you can (and should!) download it and save it on your computer.

After downloading this program, you need to load it into your project. To do this, open the file in any text editor (you could use a Python program editor window or you could just use Notepad). You should copy-and-paste my code into your program - put it at the top of your own program, but after the “import snappy” line. Now my code and yours are both part of one larger program! Test it out: run the program, and if there are no errors in loading try clicking on the green flag. You should see the dragon flying around randomly - you can still aim Alonzo’s arrow and shoot arrows, but the arrows will fly right through the dragon because the arrow script doesn’t know about the dragon yet. There are several things that can go wrong with this step, so make sure you’ve got all this code running properly before moving on.

Recall that the “shoot” script deletes a flying arrow when it hits the edge of the stage. We would now like to add a way for the flying arrow to indicate to the dragon script that it has been hit with an arrow. There are several ways you could do this, but the way we’ll do it here is to simply have a global variable named hitflag that is set to True by any arrow that hits the dragon. We also set the done flag to True, since the arrow should stop flying once it hits the dragon. This test and flag update code should be added to the bottom of the “while not done” loop in the shoot method, giving us the following final code (note the “global” statement we had to put at the top!):

```
def shoot(self):
    global hitflag
    done = False
    while not done:
        self.move(10)
        if self.touching_edge():
            done = True
        if self.touching(dragon):
            hitflag = True
            done = True
    self.kill()
```

All that we’ve done when the dragon is hit is to set a variable, which has no effect on the game unless we have some other code that checks the value of that variable. We do this in the dragon scripts that you downloaded. As the dragon moves, it checks the hitflag at each step, and if it sees that it has been set to True then it processes the hit (in the provided code, it counts the number of hits, and quits after 5 hits). You don’t have to write this code, since it has been provided for you, but here’s a slightly simplified version of the dragon code:

```
    while not hitflag:
        dragon.move(7)
        dragon.if_on_edge_bounce()
```

If you have put all these pieces together properly, then you should have a complete game! Start it by clicking on the green arrow, and try shooting the dragon. When you hit the dragon 5 times, it will do a small animation (with the dragon breathing fire), and then then it will tell you how long it took you to kill the dragon. Note that you didn’t have to do very much programming work for this part, because your partner (me!) wrote all of the dragon’s scripts. When you work on team projects, try to plan out your work so that each person is responsible for just a few sprites, and figure out ahead of time how they will communicate with each other. Then each person can work on their own code, and all the parts can be combined together at the end! However, make sure you leave time for putting things together - most novice programmers seriously underestimate how long this will take. This is one of the challenges of working in a team!

Once you are finished with this, make sure it is saved as Lab10-Activity4.py.

Submission

In this lab, you should have saved the following files: Lab10-Activity1, Lab10-Activity2, Lab10-Activity3, and Lab10-Activity4. Turn these in using whatever submission mechanism your school has set up for you.

Discussion (Post-Lab Follow-up)

More on Object-Oriented Programming

One of the key features of object-oriented programming, which we didn't describe in very much depth, are the notions of sub-classes and inheritance. Classes are often specializations of other classes, as it was when we created a Car class that was a special version of the more general snappy.Sprite class. As another example, a game might have an "Enemy" class, but then there are times when you might want to be more specific and have a class for specific types of enemies, such as an "Alien" class, a "Zombie" class, or a "Dragon" class. All enemies might share certain attributes (like a health value), but there are also some attributes that are specific to a type of enemy - of these choices, only a dragon would have a "wingspan" attribute (OK, I just made an assumption that an alien wouldn't have wings, but seriously, has anyone ever seen an alien with wings?).

Using object-oriented programming, a situation such as this would be handled by defining an "Enemy" class, and then defining the others as what are called **sub-classes**, so for example the "Dragon" class would be a sub-class of the "Enemy" class. Going the other way, we say that the more general class (like "Enemy") is a **super-class** of the more specific class (like "Dragon"). A sub-class **inherits** all of the attributes of its super-class, which means that the sub-class will have attributes for everything defined in the super-class, without having to define them again. If we defined an "Enemy" class that has attributes "health" and "location", and then defined a "Dragon" sub-class where the definition only has a "wingspan" attribute, every dragon would still have its own health and location simply because it is a sub-class of "Enemy".

The terminology that is used in object-oriented programming is to refer to an "**IS-A**" relationship. To see if one class should be a sub-class of another, ask yourself if you can put the phrase "IS-A" between the two class names. For example, we can say "A Dragon IS-A Enemy" (not grammatically correct, but give me a little break here), so the Dragon class should be a sub-class of the Enemy class. This makes sense because if one thing "is" another, then it should have all the characteristics (attributes) and can perform the same actions (method) as the second thing. To consider another example, if you are creating an information system for a university, you might have classes for "Professor" and "Student"; however, since you can't put "IS-A" between those two class (in either order) it doesn't make sense for one to be a sub-class of the other. You could define a "Person" class and then both of these could be sub-classes of "Person" since a "Professor IS-A Person." We could even define another level of class, such as "Freshman" - then a "Freshman IS-A Student" and a "Student IS-A Person." This is another nice example of abstraction in computing: we can define a sequence of sub-classes that get more and more detailed and specific, and at each level we can (mostly) ignore the details that are hidden in the super-class.

Terminology

The following new words and phrases were used in this lab:

- *attributes*: variables that are associated with a specific object or sprite
- *child object*: an object that is created by cloning a prototype object - in cloning, the original object is known as the "parent object" and the new object is a "child object"
- *class*: a general class of object – individual objects are instances of some class, which defines the common characteristics of the class (weapons, monsters, etc.)
- *constructor*: a script that is run to initialize various attributes in a new object
- *costume center*: The one point on a sprite that specifies its placement (location) and serves as the point that it rotates around
- *inherit*: the process by which a sub-class includes the attributes and methods of its super-class
- *IS-A*: the relationship between a sub-class and its super-class - for example, a "Dagger" IS-A "Weapon"
- *layers*: A depth indication for sprites, so that the front/top sprite is shown in front of sprites in lower layers
- *loop until done pattern*: An iteration pattern in which a Boolean variable (almost always named "done") controls when the loop should stop iterating
- *methods*: scripts that are specific to a particular object or sprite, performing actions on or for that object
- *object-oriented programming*: a style of program design that is oriented around defining programmatic objects in terms of the attributes and methods (note: object-oriented programming has many other aspects that are beyond the scope of this lab!)
- *parent object*: an object that creates another object by cloning - in cloning, the original object is known as the "parent object" and the new object is a "child object"
- *prototype*: an object that is defined so as to serve as a template or model of objects from a particular class
- *prototype-based programming*: a style of object-oriented programming in which new objects are created by cloning a prototype object
- *sub-class*: A class that is a specialization of another class - for example, a "Dagger" is a special type of "Weapon", so a "Dagger" class might be a sub-class of a "Weapon" class
- *super-class*: The more general class of two classes related by an IS-A relationship - in the example given in the sub-class definition, the "Weapon" class is a super-class of the "Dagger" class

Answers to Pre-Lab Self-Assessment Questions

1. The main point here is that every object needs a different version of an attribute. In the example given in the question, if "speed" were a global variable then all cars would move at the same speed. On the other hand, with attributes tied to specific objects, if you create 5 new car objects, you have also created 5 new "speed" variables, one for each new car, and each one can be used independently of the speeds of the other cars.

2. The two obvious attributes are the numerator and denominator of the fraction. There are lots of choices for methods: you could have a method that reduces a fraction to lowest terms, stamps out the fraction, multiplies a fraction by an integer, or adds two fractions together. Basically anything you might want to do with a fraction defines an action on that fraction, and any of these actions could be methods in the Fraction class.