In-Class Assignment 1 – *And Now for Something Completely Different!*

Welcome to the class!

In this assignment, you will write a quick program that prints the phrase, “…and now for something completely different!” on the screen in random locations and colors.

All in-class assignments are designed so that after you’ve correctly accomplished a primary numbered task (1, 2, 3…) you should have a working program. Execute the program (F5) before you start the next task to make sure your program is in working form.

*Physics Goals*: The students should be able to position an object on the screen.

*Programming Goals*: The student should be able to run python programs, assign values to variables and call functions, import and use packages, comment and uncomment code, and understand versioning.

*Instructions:*

1. Create a duplicate file. The first this you should do when starting a programming session is to save a new program under a new name – normally just changing the version number at the end of the file. This ensures that nothing that you do while programming will completely destroy your previous work. Saving versions every time you finish a new feature even if you continue working is also a good procedure. When you first open one of the templates in this course, you should immediately save it by replacing the “template” to your name with the numeral “1” after it.

The following steps are how to open and save a python document:

* 1. Create a folder for your coursework on your computer.
  2. Download the template from Blackboard and place it in your working folder.
  3. Double click on the Python(x,y) icon to run the environment. A small Python(x,y) window should appear on your screen.
  4. Open the SciTE editor. From the Application pull down menu, choose SciTE, then press the green check box to its right. A large text editor window should appear with the name (Untitiled) – SciTE.
  5. Load the template. Select File > Open… and choose the template you’re going to modify. In this case, it should be “IC1 – CompletelyDifferent\_template”.
  6. Save your working file. Select File > Save As… to save the file. Include your name and the initial version number by removing the “\_template.py” from the filename, and add “\_your\_name\_1.py”. Including the .py extension is important so that SciTE will autoformat your Python code. So in this case, “IC1 ‑ CompletelyDifferent\_your\_name\_1.py”
  7. You should run the template at this point either by selecting Tools > Go from the menu or pressing the hotkey F5. The presentation should come up blank.

1. Write a message to the screen. Use the gameclass function screenprint() to write the message in the view() function. If you just write gc.screenprint( screen , “text” ), the message inside the quotation will appear on the screen. A set of characters in quotation marks is called a string, and Python treats it as an ordered sequence of letters that you can reproduce and manipulate. The call has to be aligned with the text above and below it.
   1. The text should be the string constant: “And Now for Something Completely Different!”
   2. Congratulations! You’ve made your first working PyGame program!
2. Rewrite the message in the center of the screen. The previous step used a string constant in the screenprint() call. Programmers usually use variables to store information that long because it makes the code cleaner and changes easier. In this step, you will use variables to store the message you want to display on the screen and the numbers that represent position on the screen where it will appear.
   1. Create a variable message and assign the string “…and Now for Something Completely Different!” to it
   2. Replace the string constant in the gc.screenprint() call with the variable message.
   3. Assign the ordered pair (a “tuple” in Python parlance) ( screen\_size[0] // 2 , screen\_size[1] // 2 ) to a variable called position.
      1. The double-slash // denotes an integer division: 5 // 2 = 2 because 2 is the nearest integer lower than 2.5
   4. After the variable message in the screenprint() call, write a comma and then the position variable name

Your message should now flit around randomly on the screen.

1. Randomize the position of the message on the screen. Now that the position of the message is stored in a variable, it’s easy to modify it in the code. Here, you will learn to import packages with useful functions for you to use, in this case the package of randomization functions called random.
   1. Import the random module at the beginning of the program using the import command: import random.
   2. Create two variables, max\_x and max\_y. These will be the largest values possible for the upper left hand corner of the message, and so their values should be less than the values in screen\_size, but you may choose any other number you want.
   3. Change position by using a random.randint( 0 , max\_x ) call to assign a random numbers to the x-coordinate and a similar call for the y-coordinate of the variable. You may have to change your values based on where the text shows up on the screen.
2. Slow it down. The message should hop around faster than you’d like. Uncomment the pg.time.wait() call in the main() function to slow it down. The function doesn’t run because it’s been commented out—the hash # tells Python not to use anything to its right. Usually this is done to explain what the program is doing, but it is also done to keep provisional parts of the code from running during development and error checking. By deleting the hash mark and making sure the call is vertically aligned with the other parts of the main() function, the wait() function will execute when Python reaches it. It will now wait a random amount of time from 50 ms to 0.5 s before switching positions.
3. Make it pretty. Randomize the size and color of the message using the randint() function, just like you did for the position of the in step 3.
   1. Assign a random value between 10 and 50 to a variable called size
      1. The size of the text is the height of the capital letters in pixels.
   2. Append that variable to the arguments in the gc.screenprint() call.
   3. Assign different random values between 128 and 255 to three variables, red, green, and blue. These should be different values, or you’ll just get a gray scale.
   4. Assign a three-valued tuple, ( red , green , blue ) to a variable called color.
      1. Colors in PyGame are represented by three different numbers in a tuple, like this. For this class, a large number of colors have been predefined for you in the gameclass as global constants with English names.
   5. Append that variable in the gc.screenprint() call.
4. Format the message. Break the message into two parts so that it can be written more compactly in two lines, placing the second line below the first one.
   1. Create a second set of variables message2 and position2 for the second line of the message. You may want to rename the variables message and position as message1 and position1 to make things easier on your brain.
   2. Try to get the two lines to be about the same size when you break the message into two parts.
   3. The second line has to be below the first line by a distance size, so use the first position to define the coordinates of the second position, like so:
      1. position2 = (position1[0] , position1[1] + size)
      2. At this point, you can consider [0] and [1] as computer-speak for the x and y components of the position. They allow you to address the components separately when you need to, but store the two components in a single variable.

In-Class Assignment 2 – *Background Architect*

This worksheet will teach you to translate a set of dimensions into numbers that the computer can use to draw a field. You will do this by making a blueprint for the computer in terms of important coordinates for drawing out a sports field. You will use these blueprints to draw out the sports field in the next class.

*Physics Goals*: Vectors, Extension

*Programming Goals*: Preparing test data, planning a solution, coordinate – mapping, working with offsets and relative positions

*Instructions:*

1. Look up the dimensions of a soccer pitch on-line, and draw out the lines here:



1. Break the pitch into lines that form rectangles and circles. How many of them are there of each?
2. For each rectangle find its position in x and y coordinates (*in feet*) from the upper left hand corner of the pitch and its length and width in feet. Write them in the table below.

KEEP THIS TABLE! YOU WILL USE IT IN THE NEXT CLASS!

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Identifier | Type | Corner/Center | | Size | |
| x | y | Length/Radius | Height |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

1. If the screen is 800 pixels wide by 600 pixels tall and you want the pitch to fill the length of screen, how many pixels represent one foot?
2. Use that number to rewrite the coordinates and size of each rectangle or square below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Identifier | Type | Corner/Center | | Size | |
| x | y | Length/Radius | Height |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

In-Class Assignment 3 – *Background Artist*

This worksheet will teach you to use the blueprints from the class to draw a complex object made of lines, polygons, and circular arcs on the screen.

*Physics Goals*: Vector Position, Scaling

*Programming Goals*: Computer arithmetic, using functions, drawing to screen, scaling

*Instructions:*

1. Tell the program the size of the pitch. Change the size of the drawing by inputting the length and width of the pitch into the list of positions in the size variable. Use meters or feet, whichever you analyzed the points for in Assignment 2.

When you execute this, you should have a little white rectangle in the upper left hand corner of your screen. If you’ve done it improperly, it will be a little white square.

1. Color the background. Change the color of the background by adding a color parameter, e.g., gc.GREEN, to the draw\_pitch() call.

This will give you a little green rectangle in the upper left hand corner of the screen.

1. Center the rectangle. Here you will use a mathematical operation to find the place the upper left hand corner of the object should be to center the pitch on the screen.
   1. Assign a list containing the x- and y- components of the central position of the screen to a variable, and append that variable that to the draw\_pitch() call. You can use the screen\_size variable to help you find the position. To find the width of the screen, use screen\_size[0], and use screen\_size[1] to find the height.
   2. Find the coordinates on the screen in the upper left hand corner. Write this mathematically, in terms of the position of the center (cx, cy) of the screen and the length and width (l,w) of the pitch:

x =

y =

* 1. In the draw\_pitch() function, assign the length and width of the field to two coordinates. These should be taken from the field\_size variable, with the field\_size[0] being the length of the field and field\_size[1] being its height.
  2. Then use these in the draw\_pitch() function to assign the upper left hand corner’s coordinates to two variables, one for the x-coordinate and one for the y-coordinate in terms of the center, say pos\_x and pos\_y.
  3. In the call to background.detail(), change the (0,0) to a list representing the upper left hand corner of the pitch.

If you’ve done all this right, this will draw the little rectangle at the center of the screen.

1. Draw the lines. Now you’re going to want to draw the individual lines on the screen. You’ll have to use your worksheet from last time to do this. All of this will occur in the draw\_pitch() function Follow the following instructions for the shape you listed in each line in the table that you made in the last class.
   1. Input the formulas for the points that define the shape using coordinates relative to the width and height of the drawing:
      1. A line needs two points, the beginning and the end, both in tuple form.
      2. A circle needs one point for its center and a number for its radius.
   2. Draw the lines and circles using the relative coordinates:
      1. Drawing a line uses the form:

pg.draw.line( surface , color , start , finish , width )

* + 1. Drawing a rectangle uses the form:

pg.draw.rect( surface , color , ( pos , size ) , width )

* + 1. Drawing a circle uses the form:

pg.draw.circle( surface , color , pos , rad , width )

* + 1. Drawing a semicircle uses the form:

pg.draw.arc( surface , color , ( pos , size ) , start , finish , width )

You’re drawing the shape on the surface in that color with the given width. If width is left off of the last three objects, Pygame will fill the object for you.

For the color, use gc.WHITE and for the width use either 1 or 2.

You should have a small version of the soccer pitch drawn on your screen after you’re done with this part. You probably won’t be able to see the details.

1. Scale the pitch. Now you want to scale the soccer pitch to the width of the screen. The current width of the screen is kept in the first element of the screen\_size variable, so you can use that number screen\_size[0].
   1. Send the screen width to the draw\_pitch() function. This will become the full\_length variable in the draw\_pitch() function.
   2. Create a scaled\_width variable that is the full length of the screen in the x-dimension has the same ratio of the width to length in the other direction. Do the calculation for the width of the field first:

W =

* 1. Modify the position of the upper left hand corner and the height and width variables in your draw\_pitch() function to use the scaled\_size variable instead of the field\_size variable.
  2. Change the definition of the pitch to use scaled\_size instead of the field\_size.

1. Resize the screen. Finally, you can check to see how well your algorithm works by enabling the screen resize function.
   1. Let PyGame know that you want the window to be resizable. In the original definition of the screen in main, change the Boolean constant False to the PyGame constant RESIZABLE.
   2. In the event loop, uncomment the if-statement.
   3. Resize and recenter the background. The first requires a call to the resize() method in the Background class using the new\_size variable as an argument.

If you’ve done everything correctly, your background should change size to fit the screen as you change the size of the window.

In-Class Assignment 4 – *Buttons and States*

This worksheet will introduce you to activity by making a simple state machine that is controlled by the user through clickable on-screen buttons.You will have one button that controls a light, switching it on or off, and you will have another button that controls that button.

*Physics Goals*: None

*Programming Goals*: Using objects, keeping track of program states

*Instructions*:

1. Create a button. The first thing you will do is create a button, which will be an instance of the gameclass Button object.
   1. *Initialize the first button*. Creating an instance of an object requires you to assign a variable to the class to which it will belong. It may require several parameters to make it unique. Those can be found in the \_\_init\_\_() method that generates a new instance. In this case, you’ll need:
      1. a position on the screen, called pos. This is the upper left hand corner of the button in pixels from the upper left hand corner of the screen. This is a two-element list.
      2. a size for the button, called size. This is a two-element list with the width and height of the button, in that order.
      3. Three colors for the button, called colors. This is a three-element list with three colors chosen from the list in the gameclass. The first two entries, which reflect the normal color of the button and the color the button when it’s depressed can be any color you’d like. You can find a list of colors by opening the gameclass.py file and looking through its code. The last entry is the color of the button when it is inactive; it should be gc.GRAY.
      4. Then next entry is the text that is displayed on the button. For this button it should be an empty string, which is just twin quotation marks “ ”.
      5. The next entry is for the border of the button. In this case, you can just write None, which is a Python constant. If you’d like a border, you can try to decipher the docstring.
      6. The final entry is how long to pause before allowing the button to be pressed again. This is the number of times the view() method is called before it reactivates. You want this to be a small number, say 5.

You should write the following code to create the button in the main() function’s initialization section, along with definitions for the variables in the call:

switch\_button = gc.Button( pos , size , colors , " " , None , 5 )

If you run the program right now, you won’t see anything (although it should run – try it). You will need to send the button to the view() function to see it on the screen.

* 1. At the end of the main() function’s initialization section, I have left an empty list. This list will be sent to the view() function to draw each of the objects you create. You will have to type the name of your button in between the square brackets.

You should now be able to see the button when you run the program, but if you click it, nothing will happen. You’ll need to activate it in the control loop in the main() function.

1. Recognize user input. Next, you will make the button active in the control loop. To do this, you’ll need to detect that the button has been pressed and then check if it was pressed when it is on top of the button. The Button object already has a method for that, so you won’t have to write your own. You will have to write the control functions in the loop. This has three steps:
   1. *Check if the mouse button was pressed during the previous cycle of the main() loop*. This check goes in the control loop at the end of the main() function. There are already two checks in the control loop: QUIT and VIDEORESIZE, which you are already familiar with. This next check will look to see if the mouse button has been depressed using the MOUSEBUTTONDOWN pygame constant. Use the same syntax as the previous two checks to see what the mouse button is doing. This if-statement will have to be aligned with the other ones. (The program won’t run until there’s something under the if-statement, so you’ll add that next.)
   2. *Find the position of the mouse*. Just below the if-statement, and indented once (four spaces is standard for Python), assign the position of the mouse, which is stored in pg.mouse.get\_pos(), to a variable of your choice. (The program will now run again, but the button on the screen will not do anything at this point).
   3. *See if the button was clicked*. To activate the button, under the get\_pos() call, call the is\_clicked() method of the Button class with the position variable as an argument. To call a method of an object, you need to follow a simple procedure: (1) write the name of the object, (2) write a period, (3) write the name of the method with two parentheses after it, and (4) put the required arguments in parenthesis. That will look like this:

switch\_button.is\_clicked( position )

The is\_clicked() method will check to see if the mouse position is located within the square you defined for the button. If it is, then it will return a Python Boolean constant True, otherwise it will return a Python Boolean constant False. Later on, that will allow you to use the call to run an if-statement and to use a variable to store the information.

Right now the result of the method is lost because it’s not assigned to a variable, but the method executes and it alters the state of the button, which is all you need right now. Because now, if you click the button, you’ll notice it flash.

Try it.

1. Create an indicator light. You will next create an indicator which is an instance of the Light class. It will be controlled by the button you have already made.
   1. *Create the indicator in* main(). Do this the same way as you created the button, but use the Light class instead of the Button class. This takes three arguments:
      1. A pair of screen coordinates in a list directly to the left of your button.
      2. The radius of the button.
      3. A pair of gameclass colors in a list, indicating the color of the light when it’s on (the first one) and the color the light when it’s off (the second one).
   2. *Add the light’s name to the object list*. The light should now be drawn on the screen when you run the program, but it will not turn on and off. You will have to make it do that in the event loop.
   3. *Turn the light on and off*. You’ll use the is\_clicked() statement to decide whether to change the state of light. To do that:
      1. In the event loop, put the is\_clicked() call into an if-statement. That is, write if beforehand and a colon afterwards. This will break you program.
      2. Under that, and indented, call the toggle() method for your light.

Your button should now turn your light on and off.

1. Create a start button. Finally, you will make a start button to control your other button and the light associated with it. The Button class has an active property: when a button is inactive, it cannot control anything, when it’s active, it can. This lets you build start buttons for your games. Here’s how:
   1. *Deactivate the first button.* You’ll want to control the button that’s currently on the screen with the start button, so deactivate it when it’s created. Just below the line where you define it, place a call the deactivate() method on the button.
   2. *Create a new button.* Follow all the steps in part 1, except label the button using the string constant “Start” and giving it a new, different and conspicuous position. This should make a new button appear on the screen, but it shouldn’t do anything yet.
   3. *Make the button clickable.* Just as you did in part 2, place an is\_clicked() call in the event loop. You can now check to make sure that the button is working before you start using it to control the other button.
   4. *Activate one button with the other.* Now you’ll have to make an if-statement out of the is\_clicked() method, just as you did in part 3. Instead of toggling the original button, you’ll want to activate() it.
   5. *Deactivate one button with the other.* Next you’ll nest if-statements. In this case, you’ll use the is\_active() method of the switch button to determine what do when the start button is pressed. You’ll use an if-statement again, under which you’ll used the deactivate() method to turn the switch button off. The activate() call will go after an else command (with a colon after it). The if-is\_active() and else- statements are vertically aligned, indented from the if-is\_clicked() statement. The activate() and deactivate() calls are indented from the if and else commands.
   6. *Turn off the light.* When the switch button is deactivated, turn off the light using its deactivate() method.
   7. *Change the labels.* Now, the start button is turning the activity off half of the time, which decidedly isn’t starting anything. It needs a better label. So, use the Button class’ relabel() method to change it. It takes a string argument. Change the label every time the start button is clicked so that its label reflects what it will do when you click it – say, “Start” or “Stop”.

In-Class Assignment 5 – *Logic Lights*

This worksheet will introduce you to computer logic by building an interactive set of indicators that light up when different combinations of buttons are pressed on the screen.

*Physics Goals*: None

*Programming Goals*: Boolean Logic, Using Objects

*Instructions*:

1. Create a switch. In this step, you’ll simulate a switch by using a Button object to control a Light object. The light has already been placed on the screen. When the button is pressed it will switch the light from being on to being off, and vice versa.
2. Create a button. Label it “A” using the same method as we used in the previous worksheet. Best practice is to make variables for each of the arguments in the object initialization. The button should overlay the “A” already printed on the screen – try to guess where that is when you position the button, but don’t get too hung up on it now.
3. Add the button to the buttons list by placing its variable name in between the brackets. When you run the program, the button should now appear. Now you can modify the position so that it does overlays the symbol on the screen.

Pressing the button should now toggle the light. The code for triggering it is already in the control loop, and it requires the buttons to be in the correct positions, so when you add more buttons place them in the list in order.

1. Create an indicator for a logical combination. Make an indicator light that turns on when the A button is inactive and turns off when it’s active.
2. Create a Light() object as the result indicator. Try to place it over the circle labelled 2 (because this is part 2). Call it resultA, using a variable for the position. You can use the already-present variables radius and light\_colors for the obvious parameters of a Light() object.
3. Add the light to the results list by placing its variable name in between the brackets.

The light should now appear on the screen in the “off” condition and should not yet change. Move it around to overlay the circle labelled 2.

1. Use an if-statement in the main loop to turn the light on and off using its activate() and deactivate() methods based upon the state of the indicator connected to the “A” button. You can check the status of a light by using its is\_active() method.

The two lights should now have opposite activation values.

1. Create two more switches. Place buttons over the letters “B” and “C” on the screen, place them in the buttons list in order, and include the corresponding labels on them.
2. Create two more indicators. Place indicators over the next two results. These should light up when:
3. Either button B or button C is active, and
4. Both button B and button C are active.
5. Create a final indicator. This one will turn on when A and B are not active or it is not the case that B is inactive or C is inactive. Try using intermediate variables for if-statement.
6. Test your logic. Complete the following chart using your program:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | -A | B ˅ C | B ˄ C | -(A ˄ B)  ˅  -(-B ˅ -C) |
| T | T | T |  |  |  |  |
| T | T | F |  |  |  |  |
| T | F | T |  |  |  |  |
| T | F | F |  |  |  |  |
| F | T | T |  |  |  |  |
| F | T | F |  |  |  |  |
| F | F | T |  |  |  |  |
| F | F | F |  |  |  |  |

Is it all right?

In-Class Assignment 6 – *Stopwatch*

This worksheet will introduce you to working with time by allowing you to build your first object class. The stopwatch has two different states (on and off, running or stopped), and a value that changes only when the stopwatch is on. You should be able to reset the stopwatch as well.

*Physics Goals*: Vector position, time

*Programming Goals*: Creating Custom Classes

*Instructions*:

1. Make a timer object. Begin by creating a time property in the Timer class and updating it in the update() method.
   1. Create a Timer object called timer in the main() function. It will need the position of the upper left hand corner of the output when you print it to the screen in part 2 in the form of a list of two numbers.
   2. Initialize the time property in the \_\_init\_\_() method. Initialization creates an initial value for a variable, in this case use self.time = 0.0. The prefix “self.” means that the time is accessible only to the particular Timer object that the time belongs to.
   3. Put a print statement in the draw() method of the timer. This will tell you if your time is updating correctly by printing the current value in the output window.

Right now, your time is not updating, so it should just repeat “0.0” over and over.

* 1. Every time the update() method is called, increase the time property by the value time sent to the method from main().

After you have created this method it will not activate on its own. You need to make a call to the method from the game loop to activate it, and each activation will update the time a little bit.

* 1. Add a call to the update() method of the timer in the model section of the game loop.

Now, the output on the screen will increment each time it prints.

1. Print that number to the screen. You’ll need to choose a position for your stopwatch’s output, and to convert it to string form.
   1. Create a string from the time property using the built in str() function and print it instead of printing the time property directly.
   2. Use the screenprint() function from the gameclass to print to the screen. This should print to the upper left hand corner of the screen.
   3. Create some properties for the x and y coordinates of the upper left hand corner of your output in the \_\_init\_\_() method.
   4. Change the position of the timer by changing the (0,0) tuple in the timer definition to another number.
2. Make a start button. This is very much like the last two assignments. Create a button that will turn the timer on and off labelled “Start/Stop”. This will require both making a button and modifying the timer to include a property that keeps track of its activity state, and change the state when the start() method is called. The update() method in the timer should only update after the button has been pressed, and it should stop when it is pressed again.
3. Make a lap button. Create a button that will pause the display but let the stopwatch continue updating.
   1. You will need to create a button in main() labelled “Lap/Clear”, and then send the button to the view() and control() functions
   2. You will need to create properties for a display time value and a lap on/off Boolean in the \_\_init\_\_() method of your timer so that your lap() method can pause the display while the timer continues to count.
   3. Modify the lap(), update(), and draw() methods so that the lap button pauses the display. The first toggles the display boolean, the second copies the timer value to the display value when the timer is active, and the last uses the display value instead of the timer value.
   4. Modify the control function so that when the lap button is clicked, it calls the lap() method.
4. Reset the stopwatch. Modify the lap() method to clear the stopwatch if the lap button is pressed when the stopwatch is off.
5. Format the output. To create a more realistic display string, such as hh:mm:ss.ff, rather than just showing the total number of milliseconds use the format\_time() function from the gameclass in place of the str() call in the draw() method.
6. Actively label the buttons. In the control() function, relabel the buttons when pressed.

In-Class Assignment 7 – *Jumping Jack Man*

The reason for learning about time in a game is so that we can simulate motion. Simulating motion is called animation. There are two kinds of animation, active and passive. In active animation, things move procedurally according to the physics of the situation. In passive animation, each frame is drawn individually. Old cartoons, like Bugs Bunny and Cinderella, were all passive animation. The advent of computers allowed for active animation, which is used by Pixar and DreamWorks. Although we’d like to do most of our animation through active simulation, some things like people walking are better suited to using a sequence of passive stills.

In this exercise you will take a set of passive stills and animate them so that they look like the object is moving.

*Physics Goals*: Time

*Programming Goals*: Animation

*Instructions*:

1. Create a set of object properties to track the state of the avatar
   1. Create properties in the \_\_init\_\_() method to:
      1. Track the state of the avatar
      2. Keep track of the time the avatar has been in that state
      3. Keep track of the length of time the state should persist before switching
   2. Increment the time in the state in the update() method
   3. In the update() method, switch to the next state if the time in that state is longer than the length of the state.
      1. Use a print statement to provide output so that you can check that it’s working
2. Use selection rules to draw the avatar in those states
   1. Use an if-elif-else structure to choose which jumping jack man to draw.
   2. If you haven’t already, modify the update function to return to the rest state every time the jumping jack man has completed a jumping jack.