Snowman – Python

Part 1 – Game Framework

This project will take several weeks to complete, with enough time for advanced students to customise and extend the solution with extra features.

The project is to build a game that has a pay-to-play feature. The game this tutorial builds is called “Snowman”, a modern variation on the classic “Hangman” game, but slightly less macabre.

The object of the game is to guess a random, secret word, one letter at a time. When the player guesses a letter of the secret word, it’s revealed. If the player reveals the entire secret word, they win. If a player guesses a letter than isn’t in the word then part of the snowman is revealed. If the snowman is completed then the game is lost.

# Game Structure

We’re going to build Snowman with Pygame, a set of modules designed to help users build games. Pygame contains functions for managing sprites (images), animation and collecting input from the keyboard.

In order to use Pygame, we have to “import” it. Importing how you ask Python to make the functions available in your program. Note that we’re also going to import some other modules that we’ll need later.

from pygame.locals import \*

import pygame

import pygame.key

import string

import sys

import random

After the import statements, leave a blank line and type the following:

# Initialise Pygame and set up a window

pygame.init()

pysurf = pygame.display.set\_mode((1024,768))

The first line is just a comment. Comments are notes to yourself, or other people who read your code and always start with a # character. Professional software developers write *loads* of comments in their code, usually to help remind them *why* they did something or to summarise what a long bit of code does.

After the comment, the next two lines initialise Pygame and set up a window that’s 1024 pixels across and 768 pixels high. Once these lines are entered, you can run the program. A window will pop up on the screen, but you can’t close it. Don’t worry about that for now, just return to your code (the window will disappear next time you run the program).

Leave a blank line, then add the following code:

# Save the size of the window for later

width, height = pysurf.get\_size()

# How big should the snowman be?

scale = 10

# Set up where our snowman will be drawn

snX = width / 2 - (scale \* 10)

snY = 10

# Set up some colours

WHITE = (255,255,255)

YELLOW = (255,255,0)

The first line saves the size of the window in two variables: width and height. We also set up a variable scale which will control how big our snowman will appear on the screen.

We also set up snX and snY to store where we’ll start drawing the snowman from. snX is set up so that if our snowman is 20 units wide, it will appear in the middle of the window.

The last two lines set up some colours, using red, green and blue values. (This is how all colours are made using light: by mixing red, green and blue amounts.) Values between 0 and 255 are permitted for each number.

Add another blank line to the program at the end then enter the following functions. Note that each function should have one line of code, ignore where the line has wrapped in this tutorial.

# Draw a line from (x1,y1) to (x2,y2)

def snDrawLine(x1, y1, x2, y2):

pygame.draw.line(pysurf, WHITE, (x1 \* scale + snX, y1 \* scale + snY), (x2 \* scale + snX, y2 \* scale + snY),3);

# Draw a circle at (x,y) of size radius.

def snDrawCircle(x, y, radius):

pygame.draw.circle(pysurf, WHITE, (int(snX + x \* scale), int(snY + y \* scale)), int(radius \* scale),3)

This creates two new “wrapper” functions that will help us draw our snowman. They’re called wrapper functions because they wrap single Pygame functions with extra functionality. These functions will make it much easier to draw our snowman.

Next, create a function to draw the snowman. You can either copy the code below, or you can make up your own picture (use the graphing paper to design this first). Even if you don’t use the code below, read it to work out how it works to help you code your own.

def drawSnowman():

# Ground

snDrawLine(0, 20, 20, 20)

# Body

snDrawCircle(10, 15, 5)

# Head

snDrawCircle(10, 7, 3)

# Hat

snDrawLine(8, 1, 12, 1)

snDrawLine(12, 1, 12, 3)

snDrawLine(12, 3, 14, 3)

snDrawLine(8, 1, 8, 3)

snDrawLine(6, 3, 8, 3)

snDrawLine(6, 3, 6, 4)

snDrawLine(14, 3, 14, 4)

snDrawLine(6, 4, 14, 4)

# Buttons

snDrawCircle(10, 12, 1)

snDrawCircle(10, 15, 1)

snDrawCircle(10, 18, 1)

# Arms

snDrawLine(2, 15, 5, 15)

snDrawLine(15, 15, 17, 15)

# Eyes

snDrawCircle(11, 6, 0.3)

snDrawCircle(9, 6, 0.3)

If you run your program again, then it won’t do anything different unfortunately (assuming you haven’t made any mistakes). This is because we’ve create a function to draw a snowman, but we haven’t actually called it yet.

In order to get the code to run, you have to call the drawSnowman function. At this at the end of your code:

drawSnowman()

pygame.display.update() # Tell Python to update the screen

Now when you run your program a Snowman should appear.

Part 2 – Game Mechanics

Now we have created the image, it’s time to start putting some of the game mechanics around it. The image should gradually be displayed as the player incorrectly guesses letters that make up the secret word.

The way games work is that every time something changes typically the entire screen is redrawn, like drawing a cartoon frame by frame. So, every time something changes in our game (such as a letter is guessed and either a letter is revealed or another part of the snowman is drawn) we will redraw everything.

Of course, this is easy in Python, because as long as we have functions to do so, it’s not really any more work for you, the programmer, to make this happen.

## Gradually drawing the Snowman

We can simply modify the drawSnowman to gradually reveal itself as the player guesses more incorrect letters in the word. It just needs to draw more and more of the snowman as the number of “bad guesses” increases.

In order to do this, we’re going to need to keep track of the number of bad guesses made by the player so we know how much of it to draw, so add a new globalvariable, near the top of the code (underneath the class declaration but outside of any method):

badGuesses = 0 # Track the number of bad guesses the player has made.

This needs to be reset to zero at the start of each game (not just at the start of the script). We’ll be doing more initialisation later, create a function to initialise the game as follows:

def initGame():

global badGuesses

badGuesses = 0

Add a call to initGame() at the bottom of your code, along with the following logic to manage closing the window when the “X” icon is clicked on the window.

done = False

clock = pygame.time.Clock()

initGame()

while not done:

clock.tick(60)

for event in pygame.event.get():

if event.type == QUIT: ## defined in pygame.locals

done = True

pygame.quit()

This code does several things:

1. It sets up a done variable. The game will run only as long as done is False.
2. Initialise a clock variable, this is used to stop the game using too much processor time
3. Calls our initialisation function.
4. Runs a loop, which:
   1. Causes a delay so that our game doesn’t run at more than 60 frames per second.
   2. Asks Python to get a list of events (things that have happened), such as keyboard presses or mouse clicks.
   3. If Python has received an event of type QUIT, then set done to true, so the loop exits.

We’re going to want to redraw the screen every time the player presses a key, so create a new function to draw all of the graphics on the screen at once. The two lines of the function are already in your code, so just indent them and add the function declaration (the first line) at the top:

def drawGame():

drawSnowman()

pygame.display.update() # Tell Python to update the screen

Then add a call to drawGame() just under the call to initGame().

Run this code and you should see your snowman on the screen as before, but now if you click the close icon on the window (the “X” in the top right) then the game should halt.

The drawSnowman() function can be updated to draw more and more of the snowman as the number of badGuesses increases. Also, the game will have to know whether the snowman has been completed, so the method will return true to indicate that the snowman is complete (i.e. game over) or false (for more bad guesses allowed).

At the beginning of your drawSnowman function, initialise a counter. This is going to increase as more of the snowman is drawn:

count = 0

At different points in your drawSnowman method, add the following snippet:

if (count == badGuesses): return False

count += 1

Use either snippet to break up the drawing of the Snowman in to different stages, then test out your code by setting badGuesses to different numbers (in initGame) to see how changing it changes the amount of your snowman that is drawn.

At the end of the function (when the drawing is complete), change the return statement to return True instead of False. This is how the method will indicate that the drawing is complete and therefore, it’s Game Over!

## Selecting the Secret Word and Available Letters

For the game to work, we need two lists of letters:

1. The letters that make up the secret word.
2. The letters that the player can guess from.

We’re going to set this up randomly from a list of words. For now, let’s create a hard-coded array of words inside initGame:

words = ( "POINTLESS", "COMPUTER", "DECORATE", "CONSTRUCT", "PERIMETER" )

Add as many words or your own words as you like. But make sure you use block capitals. However, because we want words to be available inside other functions, we need to add an extra line to tell Python that words is a global variable. **Above** the line that starts words = ("… line add:

global words

Now, still inside initGame, initialise word to be a random choice from words.

global word

word = random.choice(words)

To allow you to easily test your game, print out the word in initGame using:

print(“Secret word is: “, word);

The initGame function should look like this:

def initGame():

global badGuesses

badGuesses = 0

global words

words = ( "POINTLESS", "COMPUTER", "DECORATE", "CONSTRUCT", "PERIMETER" )

global word

word = random.choice(words)

Run your code to make sure that it works.

## Setting up the Letters

Let’s set up render the letters the player can choose from. At the beginning all 26 letters are available, but we’ll need to keep track of them as the player uses them up.

Create a new global variable in initGame to hold all the letters:

global letters

letters = list(string.ascii\_uppercase)

So that the player knows what letters are available, we need to print them on the screen. Drawing text in Python is a bit tricky, so here’s a function to copy in to your code to make it easier:

def drawText(text, x, y, size, colour):

myfont = pygame.font.SysFont('monospace', size)

printedText = myfont.render(text, True, colour)

pysurf.blit(printedText, (x, y))

Next, create another new function to draw the letters:

def drawLetters():

posY = 350

letterSpace = width / 26

count = 0

# Print out all the letters that are in the list of available letters

for x in list(string.ascii\_uppercase):

if x in letters:

drawText(x, letterSpace \* count, posY, 64, (255,255,255))

count = count + 1

Remember that in order to see your work, you’ll need to call this new function, so add a call inside drawGame:

drawLetters()

Run your program and you should see something like this:

