pyGame Lecture #6

(Examples: fruitgame, dotgame)

MULTIFILE PROGRAMS IN PYGAME

I. Why Do We Need a Program Split Among Multiple Files?

It's extremely difficult for a programmer to keep track in her head all of the variables and functions in a file that is thousands of lines long. In order for humans to reliably maintain and upgrade software, it's necessary that the person editing a piece of code only have to keep track of a relatively small number of things all at once. (Psychologists have shown that most humans can only keep track of 3-7 things in their short-term memory at once. This is why most of us can manage to remember a few phone numbers, but the average person has some difficulty remembering their credit card numbers.) Typically, in a python program, when editing a single line of code, in theory any other line of code in the same file may directly or indirectly affect that line. Thus, when tracking down errors, one typically might have to search all around a particular file.

One innovation that helped reduce this complexity was functions. When you are writing code ***inside*** of a function, if you declare a variable inside of the function, it only exists within that function and won't conflict with a variable outside of the function. In addition to any variables you may declare in a function, you can only use the formal parameters (the items written in the parentheses at the top of the function) and any global variables (defined in the file not within any function) in that function. This helps reduce the total number of things you must keep track of, especially if one limits the number of global variables.

However, after a while, even writing a program wiith many functions in one file gets tedious. Imagine having to scroll from line 1327 to line 4434 to track down a single function call!!! Or having to remember where a particular function was written in a file when the file has a hundred functions!!! It might take 10-15 minutes just to track down ***where*** physically in the code the offending function might be.

Thus, the next level of abstraction that helps the programmer deal with complexity in programs is splitting a program into multiple files.

The idea is as follows: in each file, you keep a small number of related functions and constants. In one file you have your "main" program, which you actually run. From that file, you directly or indirectly call all of the other functions, some of which may reside in this main file and others which may reside in other files. If you split up your files in a logical way, then based on what a function does, you'll quickly know which file it's defined in. Once there, since the actual file is pretty small, you should be able to find it quickly. This is the same idea behind organizing a bookshelf by placing all books on the same subject on one shelf. You can think of each file in your program as a "subject" and within that file you have multiple functions that fit within that subject. In the main file, you call various functions that may be located in different files. The organization by files allows for the programmer to focus on a smaller portion of code when editing. It also allows for multiple programmers to work on the same code base in parallel. It's this latter advantage that truly makes software more scalable.

II. The Mechanics of Splitting a Program into Multiple Files

The actual mechanics of splitting a program into multiple files isn't too difficult. If you want one file to contain several variables and functions, put those variables and function definitions in a separate file. In that file, import anything you need (pygame, etc.) for the functions in that file. Now, if you want to call functions from this file from a different file, first put an import statement in the second file. For example, if the file fruitgame.py makes calls to functions in fruitfunc.py, then at the top of fruitgame.py we write:

```
import fruitfunc
```

Then, when we want to call a function in fruitfunc.py that resides in fruitfunc.py we write the function call as follows:

```
fruitfunc.move(fruit)
```

III. Example - Fruit Game

For Fruit Game, we want to move our functions over to a separate file. In addition, trying to remember which index corresponds to which component of a fruit object is quite error prone. Formally, a better way to solve this problem is to use classes and objects, but due to the large learning curve of object oriented-programming, we'll introduce the use of variables (meant to be constant) to make our implementation of fruit objects with lists more readable. Here are the variables we'll declare:

```
SCREEN_W = 1000
SCREEN_H = 600
X = 0
Y = 1
DX = 2
DY = 3
F_ID = 4
```

Recall that F_ID represents is the index into the fruit list that represents which fruit is being represented. Thus, in index 4 of the list of a single fruit, the value stored is either 0, 1, 2 or 3, which correspond to apple, strawberry, kiwi and cherry, respectively. With these declarations, here are the functions for the file fruitfunc.py rewritten from the original fruitgame.py:

```
def move(items):
    for item in items:
        item[X] += item[DX]
        item[Y] += item[DY]

def removeUseless(items):
    total = 0
    for item in items:
        if item[Y] > SCREEN_H:
            items.remove(item)
            total += 1
    return total

def hit(f, mypos, pics):

    if mypos[X] < f[X] or mypos[Y] < f[Y]:
        return False

    if mypos[X] >= f[X] + pics[f[F_ID]].get_width():
        return False

    return mypos[Y] < f[Y] + pics[f[F_ID]].get_height()
```

Notice the enhanced readability of this code!

After looking at the main, one more function was added to this file. The act of creating a new fruit took several lines of code and it's fairly easy to create a function that returns a newly created fruit as follows:

```
def makeNewFruit():
    x = random.randint(1, SCREEN_W)
    which = random.randint(0,3)
    mydx = random.randint(-2, 2)
    mydy = random.randint(3, 8)
    return [x,0,mydx,mydy,which]
```

This is the entirety of the file fruitfunc.py except for the imports at the top of the file:

```
import random
import pygame, sys
from pygame.locals import *
```

The main file, fruitgame.py now doesn't need these functions, though we will end up still including these variables in main again:

```
SCREEN_W = 1000
SCREEN_H = 600
X = 0
Y = 1
DX = 2
DY = 3
F_ID = 4
```

The reason for including these constants again is that if we only include them in fruitfunc.py, then if we want to utilize them in fruitgame.py, then we would have to type `fruitfunc.X` instead of `X`. Since this expression is simply going to be an index into an array, lines of code that utilize these expressions will be extremely long. Consider this line of code from fruitgame.py:

```
DISPLAYSURF.blit(fruitfunc.pics[item[F_ID]],(item[X], item[Y]))
```

If we didn't declare the variables in fruitgame.py, this line of code would be:

```
DISPLAYSURF.blit(fruitfunc.pics[item[fruitfunc.F_ID]],(item[fruitfunc.
X], item[fruitfunc.Y]))
```

This alternative is definitely harder to read than the former. The drawback of the shorter version is that the variables are declared in two files. It's possible that these variables could be defined inconsistently due to human error. This would be a bug

that is very difficult to track down. From a design perspective, it's best to have all of these constant value defined in only one place, so if any of them are wrong, they are wrong everywhere. And if someone wants to change one of these values, they only have to be changed in one place.

Other than duplicating the constants in both files, the main file, fruitgame.py just contains the main game loop. The key changes are that each function call has "fruitfunc.", without the quotes in front of it. Here is the main game loop (top portion of the file is skipped)

```
while True:

    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

        if event.type == MOUSEBUTTONDOWN:

            if pygame.mouse.get_pressed()[0]:
                for f in fruit:
                    if fruitfunc.hit(f, event.pos, fruitfunc.pics):
                        score += pts[f[F_ID]]
                        fruit.remove(f)

    if step%10 == 0:
        fruit.append(fruitfunc.makeNewFruit())

    DISPLAYSURF.fill(WHITE)

    for item in fruit:
        DISPLAYSURF.blit(fruitfunc.pics[item[F_ID]],(item[X],
item[Y]))

    pygame.display.update()
    fruitfunc.move(fruit)
    dropped += fruitfunc.removeUseless(fruit)

    if dropped > 20:
        print("Sorry, you have dropped more than 20 fruits.")
        print("The game is over.")
        print("Your score is",score)
        pygame.quit()
        sys.exit()

    clock.tick(30)
    step += 1
```

IV. Example - Dot Game, 2 Files

Now we will introduce a new example, the dot game, which is loosely based on the web game agar.io. In the game, the player controls a dot. In the field of play there are other dots. If the player's dot intersects with another dot which is smaller, then the player "eats" that dot and the player's dot grows to equal the sum of areas of the two dots. If the player's dot is equal in size or smaller to the other dot, then the player loses. In this version, if the player goes off the screen, the player loses.

The initial version of this game was written all in one file. Then, two more versions were created: one with two files and another with three files. Both of the latter versions have strengths and weaknesses. The logic behind the game is largely similar to the logic utilized in both the rain and fruitgame examples, so for these notes, rather than explaining that logic in detail, we'll just focus on the design decisions in creating the program in different files as well as the mechanics of doing so.

In the two file version, we move all of the relevant functions into a separate file, dotfunctions.py. The main game loop is found in dotgame.py.

The variables used to index into the list for the dot objects are declared in both files. These are as follows:

```
SCREEN_W = 1000
SCREEN_H = 600
DELTA_ME = 1
DELTA = 5

X = 0
Y = 1
DX = 2
DY = 3
R = 4
CLR = 5

LIME = pygame.Color(180,255,100)
PINK = pygame.Color(255,100,180)
WHITE = pygame.Color(0,0,0)
```

R represents the radius of the dot and CLR represents the color of the dot. In this game pink is used for the player and lime is used for the rest of the dots. Remember that R and CLR are just indexes into the list storing a single dot.

To make the code easier to write and make the main game loop shorter, more functions were added to this file. Any function that helped deal with a single dot or a set of dots is included in this file, a total of 9 functions:

```
def changeVel(item, addvelX, addvelY):
    item[DX] += addvelX
    item[DY] += addvelY

def moveItem(item):
    item[X] += item[DX]
    item[Y] += item[DY]

def move(items):
    for item in items:
        moveItem(item)

def removeUseless(items):
    for item in items:
        if item[Y] > SCREEN_H or item[Y] < 0:
            items.remove(item)
        if item[X] > SCREEN_W or item[X] < 0:
            items.remove(item)

def isBigger(me,other):
    return me[R] > other[R]

def eat(me,other):
    me[R] = int((me[R]**2 + other[R]**2)**.5)

def offScreen(me):
    return me[X] < 0 or me[X] > SCREEN_W or me[Y] < 0 or me[Y] >
SCREEN_H

def hit(enemy, me):
    distsq = (enemy[X]-me[X])**2 + (enemy[Y]-me[Y])**2
    return (enemy[R]+me[R])**2 > distsq

def getRandDot():
    x = random.randint(1, SCREEN_W)
    y = random.randint(1, SCREEN_H)
    r = random.randint(5,50)
    which = random.randint(0,3)
    mydx = random.randint(-DELTA+1, DELTA-1)
    mydy = random.randint(-DELTA+1, DELTA-1)
    return [x,y,mydx,mydy,r,PINK]
```

The only new piece of logic here circle-circle intersection. To determine this, we use the distance formula to take the distance between two circles and compare that to the sum of the radii of the two circles.

The key drawback of this split of files is that the variables for the screen and the list indexes are duplicated. For that weakness though, in both files we get to write more compact code, which is easier to read. For the sake of completeness, here is the whole file dotgame.py for this version:

```
import random
import math
import time
import dotfunctions
import pygame, sys
from pygame.locals import *

SCREEN_W = 1000
SCREEN_H = 600
DELTA_ME = 1
DELTA = 5

X = 0
Y = 1
DX = 2
DY = 3
R = 4
CLR = 5

LIME = pygame.Color(180,255,100)
PINK = pygame.Color(255,100,180)
WHITE = pygame.Color(0,0,0)

pygame.init()
DISPLAYSURF = pygame.display.set_mode((SCREEN_W, SCREEN_H))
pygame.display.set_caption("Dot Game!")

clock = pygame.time.Clock()

dots = []
me = [SCREEN_W//2, SCREEN_H//2, 0, 0, 20, LIME]

curT = time.clock()
score = 0
dropped = 0
step = 0
alive = True

while True:
```

```
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

        if event.type == KEYDOWN:

            if event.key == K_DOWN:
                dotfunctions.changeVel(me, 0, DELTA_ME)
            elif event.key == K_UP:
                dotfunctions.changeVel(me, 0, -DELTA_ME)
            elif event.key == K_RIGHT:
                dotfunctions.changeVel(me, DELTA_ME, 0)
            elif event.key == K_LEFT:
                dotfunctions.changeVel(me, -DELTA_ME, 0)

    if step%20 == 0:
        dots.append(dotfunctions.getRandDot())

    dotfunctions.move(dots)
    dotfunctions.moveItem(me)

    for item in dots:
        if dotfunctions.hit(me, item):

            if not dotfunctions.isBigger(me, item):
                alive = False
            else :
                dotfunctions.eat(me,item)
                dots.remove(item)

    dotfunctions.removeUseless(dots)
    DISPLAYSURF.fill(WHITE)
    for item in dots:
        pygame.draw.circle(DISPLAYSURF, item[CLR], (item[X], item[Y]),
item[R], 0)

    pygame.draw.circle(DISPLAYSURF, me[CLR], (me[X], me[Y]), me[R], 0)
    pygame.display.update()

    if not alive:
        print("Sorry, you died with a score of",me[R])
        pygame.quit()
        sys.exit()
    elif dotfunctions.offScreen(me):
        print("Sorry, you ran off the screen. Score=",me[R])
        pygame.quit()
        sys.exit()

    clock.tick(10)
    step += 1
```

V. Example - Dot Game, 3 Files

In this version, we only place the variables meant to be constant throughout the program in a single new file, dotconsts.py. The advantage of doing this is that there is only one place they are defined, so there's only one place we would have to change them if we decided to do so. Also, there's no chance of inconsistency between two different files where the constants are declared separately. (Imagine misstyping the constant in one file or deciding to change it when working on one file and forgetting to go back to the other.) The drawback is that when referencing these constants in either dotgame.py or dotfunctions.py, you have to precede each constant with "dotconsts.", which can get tedious. Here is a sample of a couple of the functions from dotfunctions.py:

```
# Returns true iff mypos is within the picture specified by f.
def hit(enemy, me):
    distsq = (enemy[dotconsts.X]-me[dotconsts.X])**2 + (enemy[dotconsts.Y]-
me[dotconsts.Y])**2
    return (enemy[dotconsts.R]+me[dotconsts.R])**2 > distsq

def moveItem(item):
    item[dotconsts.X] += item[dotconsts.DX]
    item[dotconsts.Y] += item[dotconsts.DY]
```

And here is a sample of the code from dotgame.py in the three file version:

```
# See if I am eating a dot!
for item in dots:
    if dotfunctions.hit(me, item):

        # I die :(
        if not dotfunctions.isBigger(me, item):
            alive = False

        # I grow - my new area is your area plus mine!
        else :
             dotfunctions.eat(me,item)
             dots.remove(item)

# So my list doesn't grow endlessly!
dotfunctions.removeUseless(dots)
```

The code samples contain the three files in their entirety.