Invaders From Space!

Making a game in an hour and learning Python too!

Requirements

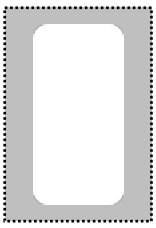
For this recipe, you will need:

- One laptop, with Python 2.7 installed
- This worksheet
- A keen sense of adventure
- Some starting files from one of your charming, clever and quick-witted demonstrators.

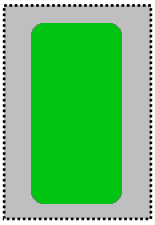
If you've got all that, please make sure you are siting comfortably, and we shall begin.

Let's-a go!

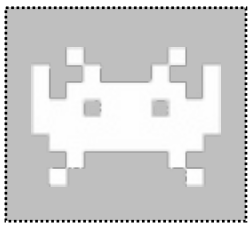
First things first; we aren't going to be starting quite from scratch - we've got some images for you to use (though if you're particularly adventurous you could put in your own instead). You'll see a folder called '***images***', in it are the following files:



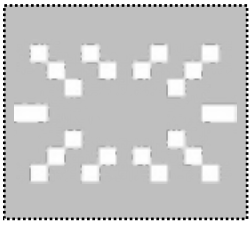
bullet.png



laser.png



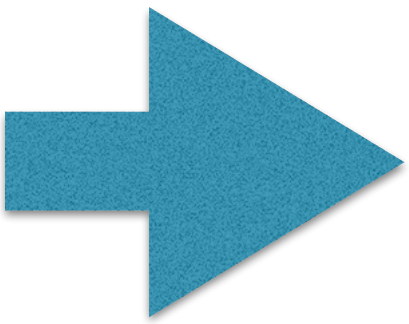
invader.png

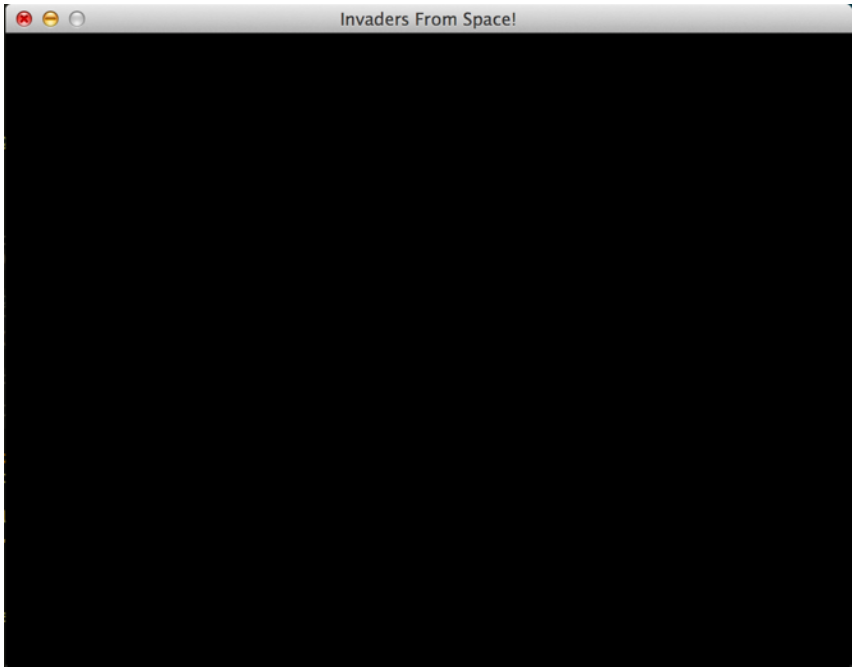


explosion.png

player.png

These pictures will be used by our game so that we can actually see what we are doing. Without them, the game looks like this. Which sucks.





Another important thing is that you have a folder, ***pyglet.*** This is the folder containing the 'library' that we are going to use to make our game-creating much, much easier. You don't need to do anything with this, it's just important that it is sitting there!

Lastly, you should also have a pair of python files, which we've provided just to give you a head start. If you want to know more about them (and we haven't answer your questions in the workshop) please ask us afterwards!

The first file, ***invaders.py***, should start a bit like this:

```
#!env python
"""This is the first of two starter files. This is the one that you
will run to start the program."""
import pyglet


class InvadersWindow(pyglet.window.Window):
    """This class does all managing: it draws to the screen, and
    updates all the bits and pieces flying around the screen!

    Extends pyglet.window.Window, overwriting the on_draw method.
    """
```

The second, ***objects.py***, should start like this:

```
"""Contains code for all the game objects."""
import os

import pyglet

# Obtain the path to this script. This workaround is
# just for importing as a module; it's not needed otherwise.
# We then use this to add the image path to where pyglet searches.
pyglet.resource.path.append(
   os.path.join(os.path.dirname(os.path.realpath(__file__)), 'images'))
```

So what's the deal?

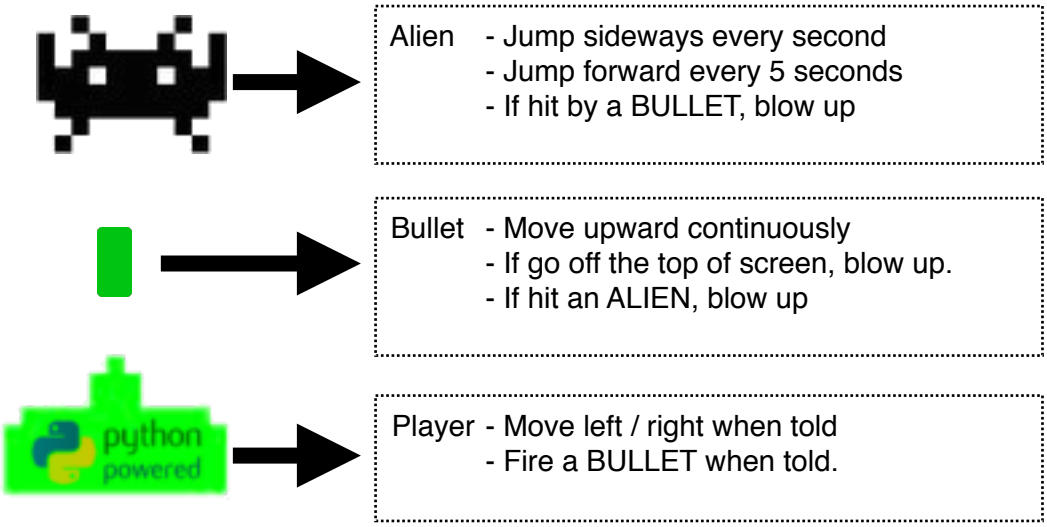
We are going to assume that you understand basic Python, and can search the internet to answer questions. We will help you in the workshop, but the exact answers aren't in the handout., because exact answers don't exist. Anyway….

These two files are what we will be adding to to make our completely original Invaders In Space! Already, there are some things that might be new to you; most important of them is the new keyword **`class`**. This keyword is our gateway into our first advanced concept: ***objects!***

Object Orientated Design is a modern, popular and effective way of writing programs. You've already seen it, because it's how Python is built! The idea is simple: instead of one large gloopy mess, you build a program out of little individual pieces. These individual pieces know what they need to do themselves, but don't worry about anything else. It's a bit like putting on a play, but instead of one script, you give each actor a small script that just has their cues and lines on it. They won't know what other actors are doing when they're not involved, but they know how to respond when they are needed.

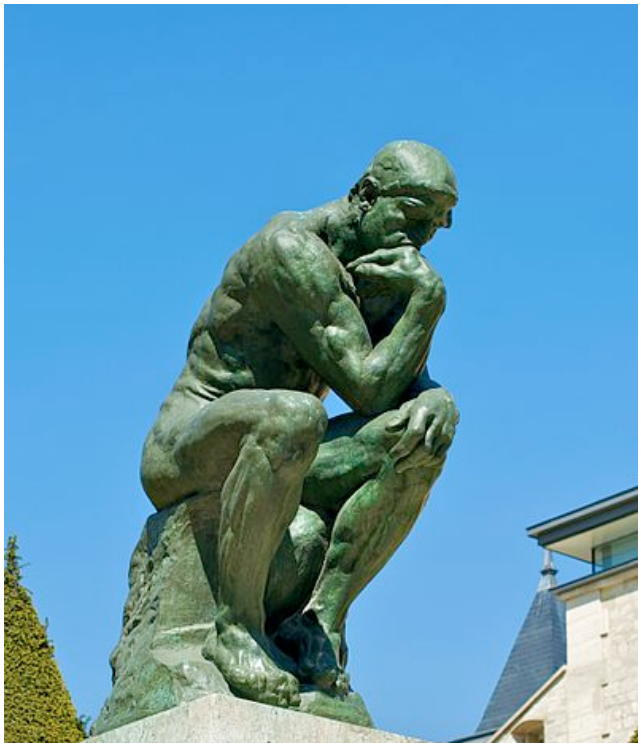
If we were going to imagine, say, a simple space invaders game, this **object orientation** actually makes perfect sense!

That way, if we decide that a player should be able to move up and down too, then we don't need to fiddle with bullets or aliens, they are all fine as they are.



Great. I am enlightened. What now?

So glad you asked! Let's look at the code. You can see lots of lines that start with **#,** or lines that are surrounded by 'triple quotes' (" " "). These are comments. You should leave these for yourself, and read any that you find. They are ignored by the computer, but are left in amongst the code, so that people can read them to help figure out what is going on! Let's go through the comments now…



"Hmm, nice comments!"

Great! Now we've read them, we're going to add code (and comments) to the files to make this basic skeleton into a game.

Step 1 - Basic Game Structure 101

Games are actually quite tricky programs to make. They need to be responsive, which means efficient, but they need to look cool too. The important thing, like with objects, is to separate what the game's 'state' is (where all the bits are, how fast they are moving etc) from how we actually put it on a screen.

There are a couple of reasons for this, but the main one is simple: computers can do millions of bits of maths a second, but a screen might only be able to refresh itself 60 times a second. So we should get everything ready before each screen refresh, ensuring that there is no delay. If we do calculations at the same time as the drawing, then our game will run at different speeds on different screens! Not good!

The way we write a game is to have a main **loop.** The loop runs as quickly as it can, doing updates if it can and drawing if it can, but if it can't do either of those then it just skips it. The skeleton file ***invaders.py*** already has this set up: managing your state is done in the **`update`** function, and the drawing is done in the **`on_draw`** function. The drawing will be done as frequently as possible, but the update is slightly different. We actually need to choose how often we run our **`update`** functions in real (sometimes called wall) time; this has the important effect that each time **`update`** is called it gets passed the amount of time in seconds since it was last called (**`elapsed_time`**). This number can be really really small, like 0.00000001, but it is still important. Pyglet, the library we are using, has a clock that lets us tell it how often we want to run things, or to run things after a delay. You can see it being used in the **`run_game`** method. When you use this to schedule a function, the first argument passed to that function will always be the time in seconds since the function was last called (or first scheduled).

Step 2 - Where to begin

If you look at the ***objects.py*** file, you'll see that we've got a **`GameObject`** class (which extends the class **`object`**) there that has a **`draw`** method. It doesn't do anything about moving, or listening to a keyboard, or firing or anything. But each of our things we might make will move differently, so one size doesn't fit all. So we make new classes for each of them, that will ***extend*** **`GameObject`**. By extending like this we get all the functions and variables from the original class, and can add or change them to suite our needs. Let's start with the player!

1. Add a new class in ***objects.py***, called something like **`Player`**, that extends **`GameObject`**. (Look at the syntax where **`GameObject`** extends **`object`**)
2. In your main game, create a **`Player`** and make sure you draw it in the **`draw`** function.
3. Now, we have to think about controls! Essentially every time we **`update`** we should make some changes based on whether we are pressing keys. Give **`player`** an update function, and call it from the main window's update. The easiest way to do this is with pyglet's **`KeyStateHandler`**. Give player one of these, and make sure it receives keyboard data from the main window. Then you can ask it which keys are pressed down in the update function.
4. Pick two keys to use for moving left and right, and give your **`Player`** a speed variable (think of this as a speed in pixels-per-second!) Then, based on whether those keys are pressed in **`update`**, you can figure out how far the player should move based on the speed variable

Our complete example code for this section is in the folder ‘stages/stage-2/’.

Step 3 - Give yourself a gun

So now we have a player tank that can move left and right. Excellent! Let’s give it the ability to fire bullets.

1. Create a new bullet class in ***objects***.*py*. We’ll call ours **`Bullet`**, but feel free to use your imagination. This class could be similar to the player, but will change the **`image`** variable to the bullet image, and it will move upwards when it is updated.
2. In the window, create a list of bullets. Then make sure in your update and draw functions that you update and draw all the bullets that might be in it! You should also think about removing bullets that go off the top of the screen somehow. We’ll go over a neat way to do this involving **`destroy`** and a Python feature called ‘list comprehensions’.
3. In your player’s init function, add an argument that refers to the main window (we’ll call ours **`window`**), and store its value somewhere.
4. Choose a key that will be your fire button, and then add code to the player that will, when the button is pressed, add a **`Bullet`** to the list in **`window`**.

Now try firing! You’ll notice something is a bit crazy: you can just spit out an endless stream of bullets! Why is this?

That's right! The update function is being called about 120 times a second; each time it runs AND you are holding down your fire key, a new bullet is made. Ouch. Let's fix this, using another type of scheduling. If you look in the **`GameObject`** class, at the **`explode`** method, you'll see we've used **`pyglet.clock.schedule_once`** to call the destroy function just once a few moments later. This is really handy, letting us pick how long in *real time* a chain of events takes. Let's use this to solve our bullet craziness…

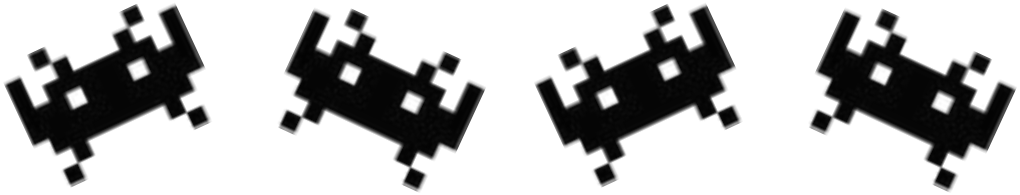
5. Think about how you might know if you've fired already.
6. Then think about using the clock to let yourself know you can fire again. We did it with only one extra variable and one extra function. Have a think, and we'll help you out!

Again, our complete code for this stage is in the 'stages/stage-3' folder. Don't peek now though! That's cheating!

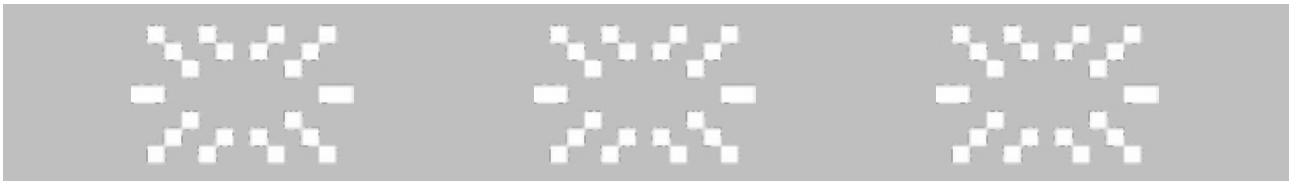
Step 4 - Unleash The Alien Swarm!

This is a big one! We'll be on hand to help, but we're going to leave a lot of the details up to you: it's your game, after all.

You'll need to make a new Alien class. Our Aliens hop sideways, but yours can move however you like. Get an Alien class, make one alien appear at the top of the screen, and make it move sideways on its own. You can reuse things we've learnt so far to do this, but if you get stuck, let us know!



Right. Now we have a moving target! We've provided some basic collision detection for you via **`GameObject`**. The **`has_hit`** function returns True if the object you are calling it on has it the object you pass in as an argument. Sounds like this would suit our bullets perfectly! Think about how this might be useful somewhere in your update function, and what you want to do if you do hit an Alien! (Hint: we seem to have also given you an **`explode`** function… Hmmm…)



Well, that's the basics! In the time we have left, we're going to open the floor to you guys! What do you want your game to be like? How many Aliens do you want? Do they keep coming, or are there only a certain number? Do they slowly creep forward, or do they stay put? Do they fire back? We'll help you figure it out, but if you want to see what our finished version looks like, you can find it in the folder called 'stages/Complete'. Happy hunting!