Make your own retro platformer

Code your homage to Rainbow Islands in Python — a vertical scrolling platformer where enemies meet incredibly colourful deaths

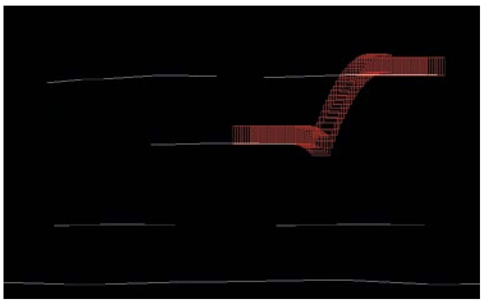
AUTHOR
JORDI SANTONJA

Jordi currently works as a software engineer in Norway for a company that develops systems to design infrastructure: roads, railways, utilities... He originally comes from Spain and, in his spare time, codes video games.

Download the code from GitHub: **wfmag.cc/wfmag57**

Platform games have long been about dexterity: running, jumping, and avoiding enemies and pits. The genre's roots go back to Nintendo's *Donkey Kong*, released in 1981, and gradually evolved from single fixed screens to scrolling levels, and then to 3D.

Rainbow Islands: The Story of Bubble Bobble 2, developed for arcades by Taito in 1987, stood out from other platformers thanks to its vertical level design and unique attacks. The player makes their way from the bottom to the top of narrow stages, and can cast rainbows which can be used as both temporary platforms and an attack that destroys enemies. The rainbows kill the enemies if they collide with them when created, or, when destroyed, if they fall down over them. A rainbow is destroyed when the player character jumps over it, or after a certain time from its creation.



This is what happens when the player jumps just after leaving a platform: they jump in the air like a cartoon character. Hence its name, 'coyote time'.

We're going to code our own take on *Rainbow Islands* in Python and Pygame Zero, with the help of Shapely for the collision detection. Head to **wfmag.cc/shapely**, where you'll find out how to download and install it.

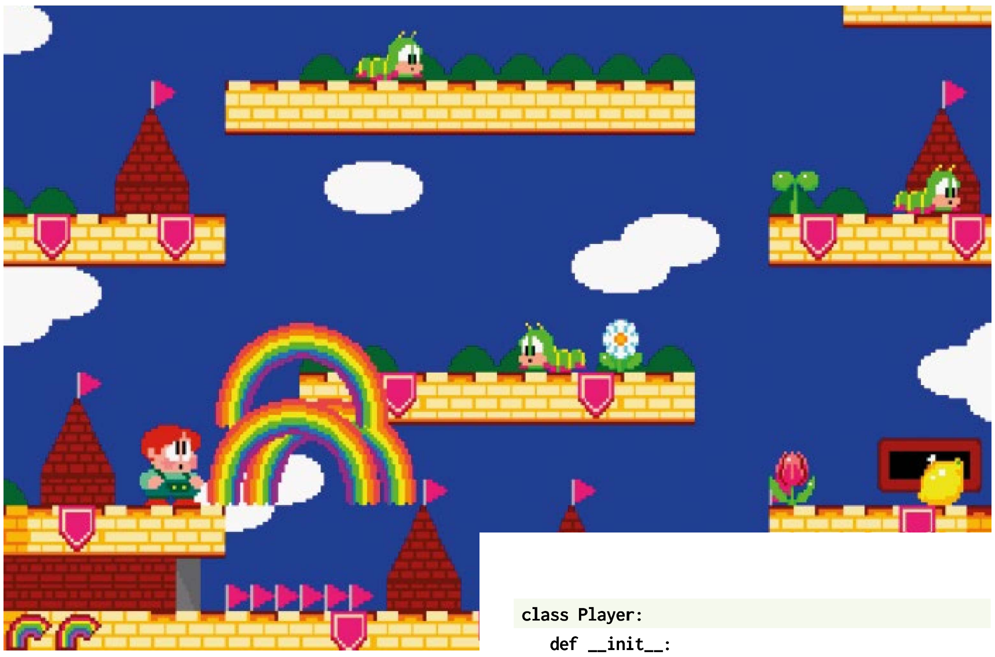
DRAW PLATFORMS

Let's start coding some of the game's main mechanics. We'll first test the player's interactions with platforms: jumping, falling, and walking around. We need some platforms to check all this, and to draw them we use the program **Listing01_DrawPlatforms.py**. We can draw lines with the mouse, **RETURN** starts a new line, and the **SPACE** bar prints the resulting lines to the Python console.

In the game, the player's character can jump up through a platform without colliding with it, but when they're dropping down they'll land on top of the platform. This way, the player can advance upwards by jumping from platform to platform. To achieve this, the platforms consist of a single open polyline from left to right, which doesn't intersect itself and has no loops.

PLAYER PROTOTYPE

The first lines of **Listing02_PrototypePlayer.py** contain the platforms, called `surfaces` in the code, copied from the result of the previous program. The rest of this listing is quite complicated and long, but it contains the main foundation of the full game. After the platforms come some player


Rainbow Islands: The Story of Bubble Bobble 2 – the 1987 classic we're using as the basis for our Python project.

parameters: size, acceleration, jump speed, terminal speed, and lateral speed. You can change them at will to see how they affect the character's movement.

Next, the `Player` class is defined. The `update` function contains its main behaviour, where the collision with the platforms is handled. The collision is computed only when the player is falling down, that is, when `speedY >= 0`. An object with a speed greater than zero goes down. When the speed is less than zero it goes upwards, and it doesn't collide with anything. The collision is computed using Shapely's `intersection` function between two Shapely geometries: the platform line and the player bounding box. Both are defined as Shapely `LineString`. The intersection result consists of one or more points. In order to avoid missed intersections when the player is falling down at a high speed, the player's bounding box size is expanded down by the speed value. The highest point of the intersection is assigned to the player's position. This way, when the player's character intersects a platform, it will remain at the top.

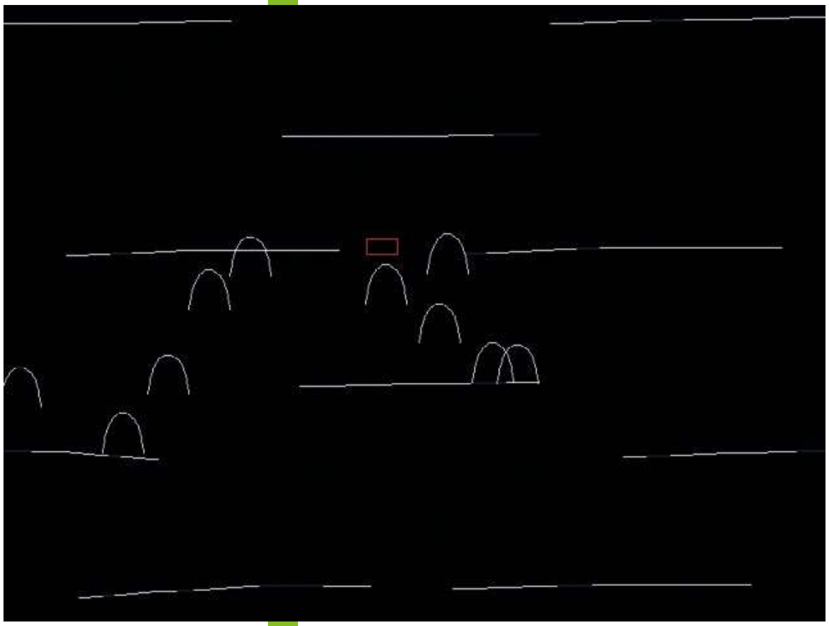
After the collision detection, the new position is computed from the old position and speed, limit checks are performed, and finally, input management from the keyboard is done with the three functions: `jump`, `left`, and `right`.

The pseudo-code for this prototype can be described as follows:

```
surfaces = [point lists]
player parameters
class Player:
   def __init__:
      centre = initial position
      lineString = box around centre
   def draw:
      draw lineString lines
   def update(surfaces):
      if speedY >= 0:
         for all surfaces:
            intersection surface / box
            if intersection:
               centre.Y = intersection highest
point
      centre.Y += speedY
      translate(lineString)
   def jump:
      if not jumping:
         speedY = -playerJumpSpeed
         jumping = True
   def left:
      centre.X -= playerLateralSpeed
   def right:
      centre.X += playerLateralSpeed
player = Player()
def draw():
    screen.clear()
    for all surfaces:
       draw surface
    player.draw()
def update():
    if keyboard.left:
        player.left()
    elif keyboard.right:
        player.right()
    if keyboard.up:
        player.jump()
    player.update(surfaces)
```

PROTOTYPES

When you have an idea for a new mechanic, one of the best ways to check its feasibility and fun factor is to build a prototype. A prototype must be fast to code, and needs no fancy graphics: some boxes and lines will do. But it needs to show the mechanic as best as it can be implemented, so if it's successful, it can be translated directly to the final game with finished graphics. I recommend taking a quick and dirty approach to prototyping – build it in the engine or programming language you're most comfortable with. If the prototype's successful, it can be translated into proper code in the final platform.



The code 'Listing08_PrototypeShootRainbows_Polygon.py' is used to test the creation of rainbows and the correct placement of the player on the rainbows. Please try it, tweak the code, and play around to test the results.

GAME PARAMETERS

To test the possible values of the parameters of the game, we can code them into variables that can be easily changed, either in the code or in a file. Be it the speed of the player character, the height of its jumps, or the size of the rainbows, it can be changed any time to immediately test its effects on the gameplay. The goal is to iterate rapidly through the possibilities to find the most satisfying and fun values.

VERTICAL SCROLLING

To tackle the scrolling, we need some platforms. To make them, we'll adapt the drawing platforms listing to get **Listing03_DrawPlatformsVertical.py**, where pressing the arrow keys 'Up' and 'Down' displaces the whole screen vertically. Then we adapt the player's prototype code to allow the vertical scrolling in **Listing04_PrototypeVerticalMovement.py**, which defines the new variable `screenPosition`, which stores the vertical viewing position. All the operations are performed as before, with the intersections done in the world coordinates. In the `draw` functions, `screenPosition` is subtracted from all objects' vertical position to draw them in the right place on the screen after scrolling. `screenPosition` is updated in `screenPositionUpdate` from the vertical position of the player.

Another tweak has been introduced here. If you've played around with the first prototype, you may have noticed that if you press 'Jump' after you fall from a platform, your character jumps. To avoid this double jump, a check has been introduced when the character is in mid-air:

```
    if not self.jumping and self.speedY >= 4:
        self.jumping = True
```

This sets the character as jumping when the vertical speed downwards is higher than 4, thus avoiding a double jump. This still allows the jump in mid-air just after falling from a platform. This is a useful technique in platform games called 'coyote time'. The next step is to rename the '4' to something more meaningful, `coyoteTimeVerticalSpeed`, and test different values.

ENEMIES

Let's prototype the enemies now. In **Listing05_PrototypeEnemies1.py** and **Listing06_PrototypeEnemies2.py**, we code two basic enemy behaviours: back and forth on a platform, and falling down from the border of a platform.

The first enemy moves over a platform without leaving it. That behaviour is achieved by using a line to intersect the platform. The line is placed in front of the enemy to find the vertical position of the platform and place the enemy accordingly. If the line doesn't intersect the platform, it means the enemy has passed the border, so it must reverse its trajectory to go back to the platform.

The second enemy moves over a platform, but when it reaches the platform's border, it falls down. The collision box is similar to the player's character one, so we could reuse the code here.

SHOOT RAINBOWS

Next, prototype the rainbows. We reuse the code from **Listing04_PrototypeVerticalMovement.py** to create the new file **Listing07_PrototypeShootRainbows.py**. Here, every time the player presses the **SPACE** bar, a new rainbow is created. As the rainbow is placed in front of the player, a new variable is defined: `directionX`, which can store two values, +1 and −1, facing right and facing left.

"Now we have a basic prototype for all the main elements of our platformer"

The new rainbow is then added to the current platforms as an arc, so the player can walk and jump on it. But in the first prototype, we find that the player's box gets into the arc when walking on top.

To fix this, we must compute a `Polygon`/`LineString` intersection, that draws a `LineString` completely inside the `Polygon`. Then we get the highest point of the resulting `LineString` to place the player's character. Now the player correctly rests on top of the arcs without intersecting it. The file **Listing08_PrototypeShootRainbows_Polygon.py** contains the changes.

Another change is the size of the bounding box to detect intersections. When you jump from a platform to the one above, you'll suddenly appear on top of the platform, which is a jarring behaviour. We can reduce the size of the bounding box to avoid this.

COLLECTABLES

We have enemies and rainbows. What happens when an enemy is killed by a rainbow? An item is released that can be collected by the player. **Listing09_PrototypeShootCollectables.py** prototypes how these items fly from the enemy and land on a platform.

Here, when the player presses the **SPACE** bar, a new flying item is created at the character's position. Then it starts to fly in a random direction, computed with a pair of `randint` – one for horizontal speed and the other for vertical speed – and it flies until it lands on a platform. The collision detection is similar to the player character collision detection.

The `Collectables` class contains two lists, `collectablesFlying` and `collectables`, that store all the collectables, and defines the function `addCollectable` to add a new flying collectable. When a flying collectable lands on a platform, it's deleted from the `collectablesFlying` list and added to the `collectables` list.

DESTROY RAINBOWS

The **Listing10_PrototypeDestroyRainbows.py** program prototypes both ways to destroy a rainbow: when it reaches its time limit and when the player jumps over it.

In this code, the rainbows are independent entities, stored in the class `AllRainbows`. This class keeps two lists: `rainbows` and `fallingRainbows`. When the player shoots a rainbow, it's appended to the `rainbows` list, and when the player's character jumps over a rainbow, or its time of life `rainbowTimeLife` has ended, it's removed from the `rainbows` list and appended to `fallingRainbows`. When the falling rainbow exits the screen, it's removed from the list.

When the player's character lands on a rainbow, its speed is compared against `playerVerticalSpeedToDestroyRainbow` and, if bigger, `allRainbows.rainbowFall` is called to destroy the current rainbow and to add a new falling rainbow. Another tweak to the player's behaviour is also introduced here: the variable jump. When the player presses the **SPACE** bar longer, the character jumps higher, and with a short press, the character jumps shorter. This has been achieved by calling `stopJump` when the **SPACE** bar is released.

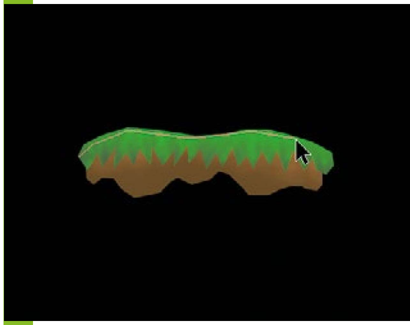
GRAPHICS

Now we have a basic prototype for all the main elements of our platformer: the player's character, the platform, enemies, collectables, and rainbows as weapons. With some adjustments, it's possible to start putting it all together to get the final game. It's also the time where we can start creating the graphics. My preferred software is GIMP (**gimp.org**), but you can use any other software capable of creating graphic files with transparency – PNGs in our case. Please replace the provided graphics with your own – I'm not an artist!

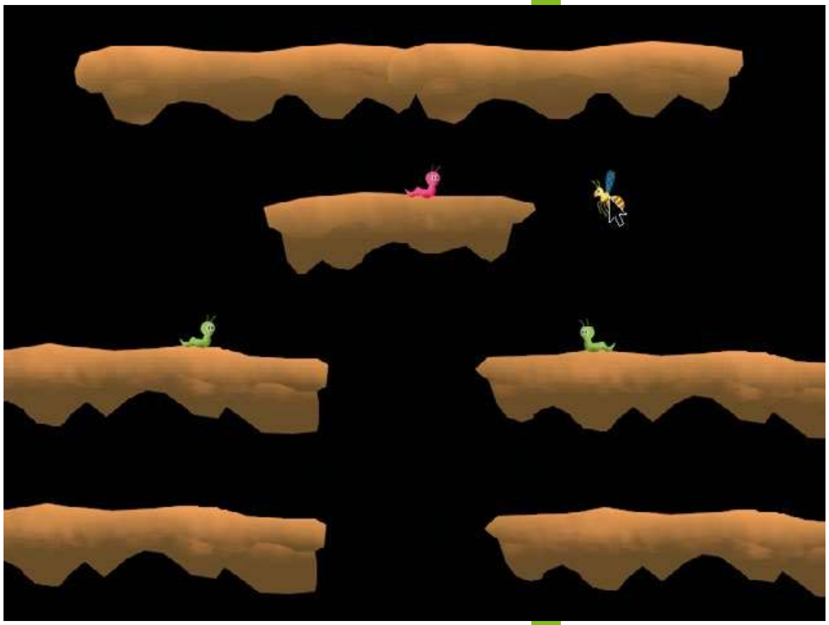
PLATFORMS

With the help of GIMP, we've designed a set of platforms to be placed on the screen. The list of platform file names is stored in the file **Listing11_PlatformNames.py** under the name `platformNames`. Now we must draw a line over each to define the top line where objects can land. We draw the lines with the program **Listing12_DrawPlatformLine.py**, and, after pressing the **SPACE** bar, the result is stored in the file **Listing13_PlatformLines.py** under the variable `platformLines`.

Next, we create the level by placing the platforms in their final positions with the program **Listing14_PlacePlatforms.py**. Here, we use the mouse to move and place every platform, the ➜


^ 'Listing12_DrawPlatformLine.py' helps us to define the platform surface where the game objects will rest.

v After creating the map, 'Listing19_PlaceEnemies.py' allows us to place the enemies on the platforms.



arrow keys to select platforms and to scroll the screen, and the **SPACE** bar to print on the Python console the final distribution of the platforms. Every entry of this list contains the index of the platform and the global position on the screen. This list is copied into the variable platforms in the file **Listing15_Platforms.py**.

The code in **Listing16_TestPlatforms.py** combines the player and the rainbows from **Listing10_PrototypeDestroyRainbows.py** and the newly created platforms, stored in the new class `AllPlatforms`. Here, we discover that the previous jump height is not enough to reach the next platform, so we increase `playerJumpSpeed` from 12 to 14.

At the moment, our background is plain black. As we've drawn some clouds as platforms, it feels appropriate to have the background fade from black to blue as the player goes up. The colour is computed in the function `backgroundColourUpdate` and stored in `backgroundColour`. This code also draws the platform collision lines to check that everything works as intended.

PLAYER

Now it's time to test the player and rainbow graphics with the program **Listing17_TestPlayerGraphics.py**. Here, we've defined the variable `drawLines` to enable or disable drawing the collision lines. In the final game it must be `False`, but for debugging and testing, it's convenient to activate it. The player can shoot up to three rainbows at a time. To test it, the variable `numberOfRainbows` has been introduced, and set to 3. Later, it must start with 1 and be incremented every time a specific item is collected. To allow a small delay in the creation of the rainbows, each rainbow is created with a `creationTime`, and it becomes active and visible when the time arrives.

All the player image names are stored in the variable `playerImageNames`, and then loaded into actors at the list `actors` in the class `Player`. Then, in the `draw` function, the right image is selected from the current state of the player.

ENEMIES

Let's place the enemies on the level map. I've created three types of enemies: one that crawls on the same platform, another that crawls on platforms but falls from its borders, and a third that flies. The image file names are stored in the file **Listing18_EnemyNames**. It's a list of lists: every element of `enemyNames` is an enemy, containing the list of all the graphic files that define that enemy. The first element in every list is the enemy facing right, the second one is facing left; and if an enemy has more graphics, all odd positions face right and all even positions face left.

To place the enemies on the map, the program **Listing19_PlaceEnemies.py** is used. With the arrow keys 'Left' and 'Right', we can select the enemy type and its direction: facing left or right. With 'Up' and 'Down', we move the map, and with the **SPACE** bar, the list of enemies and positions is printed on the Python console. We copy this result on the file **Listing20_Enemies.py**, in the variable named `enemies`. Every element of this list stores four values: the enemy index, its direction +1 or –1, and the two screen coordinates X and Y.

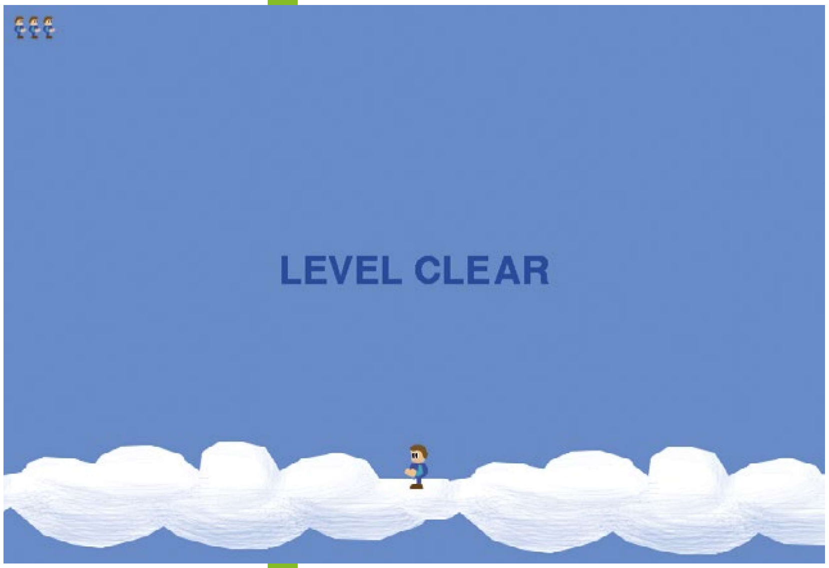
FULL GAME

Now it's time to put all these elements together. **Listing21_FullGame.py** gathers all the previous pieces of code to connect them and create a full level. We now have a huge chunk of code where it can be difficult to find anything. To fix that, we need to split the code into manageable chunks.

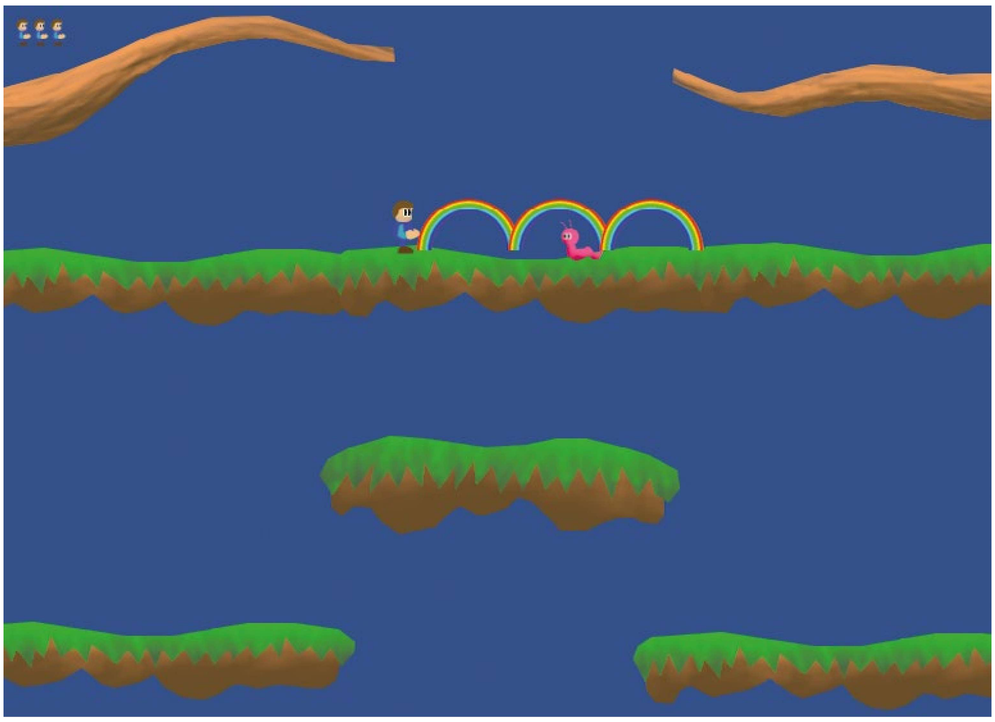
Code files named **Listing22** to **Listing29** have been extracted from **Listing21_FullGame.py**, and called from **Listing30_FinalGame.py**. Some adjustments have been made to pass around parameters and objects, as the global variables defined in **Listing21_FullGame.py** aren't accessible from the extracted modules. **Listing23_IntersectionRectangles.py** is a

COYOTE TIME

If the jumping from platforms is pixel-perfect, sometimes it can feel unfair to miss the platform by a pixel and fall to the void without the possibility to jump. To avoid this, some platformers implement a 'coyote time', where the player can still jump after having left the platform and being mid-air, in a similar way the cartoons realise they are in mid-air and start falling.



Can you reach the top of the stage? And more importantly, can you create a more challenging map?



It's not pretty, but it's still the basis of a platform game similar to *Rainbow Islands*.

simple rectangle intersection calculation. Its function **RectanglesIntersect** returns **True** or **False** when the two input rectangles overlap (or don't). A rectangle is defined by its central point and its half size. This function performs a faster intersection than with **LineStrings** or **Polygons**.

Listing24_Rainbows.py has all the rainbow-related operations. The class **AllRainbows** keeps two lists: **rainbows** and **fallingRainbows**. A rainbow will only intersect an enemy at the exact time of its creation, when **timeFromCreation == 0**. A rainbow won't destroy anything at any other time, but enemies or the player will be affected if they interact with it. Elements in **fallingRainbows** can, however, destroy enemies if they intersect – this is computed with the **RectanglesIntersect** function.

The code in Listing26_Collectables.py manages the collectable items with two main lists in the **Collectables** class: **collectablesFlying** and **collectables**. When an enemy dies, it releases a collectable by adding a new element to the **collectablesFlying** list at the same enemy position but with a random speed. When the flying collectable lands on a platform, an element from the **collectableNames** list is selected randomly. One of the elements is a small rainbow – when collected by the player, this adds to the number of rainbows they can cast at once.

"In this guide, we've learned how to code a platform game from scratch"

Listing27_Player.py contains all the player's character stuff. When an enemy collides with the player, the number of lives is decremented, the player's moved to its starting position, the number of rainbows restarts at 1, and all the enemies are restored. When the number of lives reaches 0, the game ends.

Listing30_FinalGame.py imports all the previous modules, creates the game classes, and calls them at **draw** and **update**. It manages the keyboard input, and also manages the **levelClear** variable: when the player character reaches the top of the platform map, it's set to **True** and the level ends.

NEXT STEPS

In this guide, we've learned how to code a platform game from scratch, starting with prototypes to test the mechanics in a complete level. The work is far from done, though.

From here, you can develop the project further: add more enemies, draw new graphics, add sounds and music, more levels, bosses, and so on. Or, even better, prototype your own type of platformer featuring a completely new game mechanic. It could be fantastic!

FINITE-STATE MACHINES

If your player system has more than two states, you must consider using a finite-state machine to control all the states and transitions between them. This way, it's much easier to know what to do when the user presses 'Jump' and the protagonist is in the air falling after a hit by an enemy, for example.