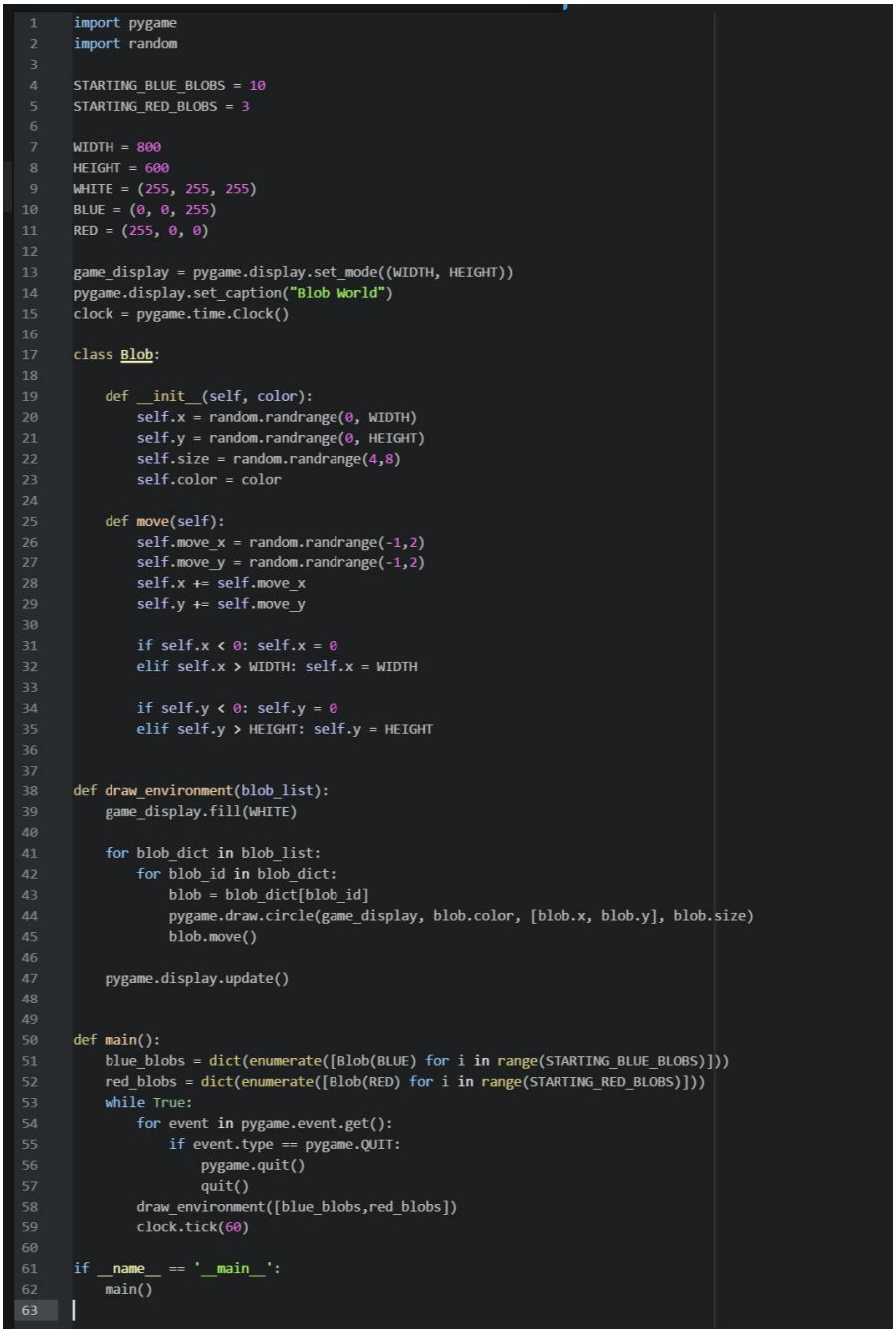
Creating the Environment

A Class is what we need to create an object, so we'll begin with a `Blob` class. Python's objects have a bunch of "special methods" often called `magic methods`. The most common one of these is the `__init__` method, but there are many others, many of which we'll touch on. The `__init__` method (pronounced: "dunder init method", where "dunder" describes the double underscores) is a method that we can use to specify anything that we want to happen when the object is initialized. Any time someone defines a variable as being an object of the `Blob` class, the dunder init method will run. Let's populate this method with some basic information, like maybe a starting location for the blob, a size, and a color of the blob:



```
import pygame
import random

STARTING_BLUE_BLOBS = 10
STARTING_RED_BLOBS = 3

WIDTH = 800
HEIGHT = 600
WHITE = (255, 255, 255)
BLUE = (0, 0, 255)
RED = (255, 0, 0)

game_display = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Blob World")
clock = pygame.time.Clock()

class Blob:

    def __init__(self, color):
        self.x = random.randrange(0, WIDTH)
        self.y = random.randrange(0, HEIGHT)
        self.size = random.randrange(4,8)
        self.color = color

    def move(self):
        self.move_x = random.randrange(-1,2)
        self.move_y = random.randrange(-1,2)
        self.x += self.move_x
        self.y += self.move_y

        if self.x < 0: self.x = 0
        elif self.x > WIDTH: self.x = WIDTH

        if self.y < 0: self.y = 0
        elif self.y > HEIGHT: self.y = HEIGHT


def draw_environment(blob_list):
    game_display.fill(WHITE)

    for blob_dict in blob_list:
        for blob_id in blob_dict:
            blob = blob_dict[blob_id]
            pygame.draw.circle(game_display, blob.color, [blob.x, blob.y], blob.size)
            blob.move()

    pygame.display.update()


def main():
    blue_blobs = dict(enumerate([Blob(BLUE) for i in range(STARTING_BLUE_BLOBS)]))
    red_blobs = dict(enumerate([Blob(RED) for i in range(STARTING_RED_BLOBS)]))
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                quit()
        draw_environment([blue_blobs,red_blobs])
        clock.tick(60)

if __name__ == '__main__':
    main()
```

The init method has two arguments: `self` and `color`. The `self` argument can be called *anything* you want, but "self" is the convention. `self` is the instance object. We will use `self`, to create and access an object's attributes from within the class, as well as outside the class (where the object's variable name takes the place of "self"). The `color` argument is going to be our only variable for now. Now we can add other regular methods too.

The idea of this is that it's going to interact in a limited environment, with a fixed width and height, so we'll go ahead and add some handling for if this blob happens to exceed boundaries:

```
  def move(self):
        self.move_x = random.randrange(-1,2)
        self.move_y = random.randrange(-1,2)
        self.x += self.move_x
        self.y += self.move_y

        if self.x < 0: self.x = 0
        elif self.x > WIDTH: self.x = WIDTH

        if self.y < 0: self.y = 0
        elif self.y > HEIGHT: self.y = HEIGHT
```

```
import pygame
import random

WIDTH = 800
HEIGHT = 600
WHITE = (255, 255, 255)
BLUE = (0, 0, 255)
RED = (255, 0, 0)

game_display = pygame.display.set_mode((WIDTH,HEIGHT))
pygame.display.set_caption('Blob World')
clock = pygame.time.Clock()
```

The colours are RGB tuples.

Now we're going to have the beginnings of a function that handles the drawing of the environment:

```
def draw_environment():
    game_display.fill(WHITE)
    pygame.display.update()
```

Generally, the process for a simple game is something like: run game logic/calcs, "draw" the next frame in the background without actually showing it, updating individual parts, then, when full updated and ready, push it to the screen. In our case, we're just filling the

screen with white, then using the `.update` method to push the latest frame to the screen.

Now for the `main` function and the if name statement:

```
def main():
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                quit()

        draw_environment()
        clock.tick(60)

if __name__ == '__main__':
    main()
```

In our case, this will run infinitely, unless we hit a `pygame.QUIT` event (user hits the "X" to close the window). For each loop in the `while True`, we check for a quit event, then draw the environment, and use the `clock` to control our FPS. In this case, we're setting FPS to 60. Just because we set it to 60, it doesn't mean it will actually run at 60 FPS, it just wont go any faster. PyGame runs on your CPU, and as you just learned, is going to default to running on a single core, so it's not the ideal starting environment. Converting PyGame to being a multiprocessing engine is quite a challenge, and we're not going to do that here, but we can make sure our actual game logic runs in a separate process or multiple processes to make sure PyGame at least gets a core all to itself.

All we get, however, is just the white environment. We want our blob to be here too! Let's create a `Blob` object, and then add it to our environment:

To start, we need to actually create a `Blob` object, all we've done so far is just create the class. We can do this in the `main` function:

```
def main():
    red_blob = Blob(RED)
    ...
```

In this case, we're specifying the one argument that the `Blob` class wants (`color`, as described in the __init__ method) as `RED`, so the color attribute of the `Blob` will be the red. Now let's pass the blob object when we call the `draw_environment` function:

```
        draw_environment(red_blob)
```

and:

```
def draw_environment(blob):
    game_display.fill(WHITE)
    pygame.draw.circle(game_display, blob.color, [blob.x,
blob.y], blob.size)
```

```
    pygame.display.update()
```

The `game_display.fill(WHITE)` will fill the entire screen with white, then we draw the blob as a circle, then we update the display. Notice how we can reference things like `blob.color` and `blob.size`, we're able to do that through our use of the `self` in defining the class methods.

If we run this, we get a blob on the screen:

Now what if we want this blob to dance? We just need to call `blob.move()` in the `draw_environment` function, pretty much anywhere, but we'll just add it to the end:

```
def draw_environment(blob):
    game_display.fill(WHITE)
    pygame.draw.circle(game_display, blob.color, [blob.x,
blob.y], blob.size)
    pygame.display.update()
    blob.move()
```

we're going to explore some of the properties of using our object.

To begin, we're going to take our simple class, and illustrate how we can quickly generate many objects from our class. To begin, let's add some new constants:

```
STARTING_BLUE_BLOBS = 10
STARTING_RED_BLOBS = 3
```

Next, let's consider a scenario where we want to add a bunch of blue blobs from our Blob class. We might go into our `main` function and do something like:

```
def main():
    blue_blobs = [Blob(BLUE) for i in range(STARTING_BLUE_BLOBS)]
```

Then we could modify the `draw_environment` function:

```
def draw_environment(blobs):
    game_display.fill(WHITE)
    for blob in blobs:
        pygame.draw.circle(game_display, blob.color, [blob.x,
blob.y], blob.size)
        blob.move()
    pygame.display.update()
```

Now we have many blue blobs wiggling about! That's fine, but we should probably should have some way of tracking each blob, otherwise we really don't know which blob is which. In the future, we might want to actually know blobs by some sort of id or name. If only we knew of a great way to assign a sort of id via a counter and then maybe make a dictionary out of it.So, we can use enumerate() - It returns an enumerated object, containing a pair of count/index and value.

enumerate

Thus, back in our `main` function:

```
def main():
    blue_blobs = dict(enumerate([Blob(BLUE) for i in
range(STARTING_BLUE_BLOBS)]))
```

Now we have a dictionary, where the key is an ID, and the value is the blob object. While we're at it, let's add some red_blobs:

```
def main():
    blue_blobs = dict(enumerate([Blob(BLUE) for i in
range(STARTING_BLUE_BLOBS)]))
    red_blobs = dict(enumerate([Blob(RED) for i in
range(STARTING_RED_BLOBS)]))
```

Now, what we can do is instead pass a list of dictionaries to our `draw_environments` function, doing this in the `main` function:

```
def main():
    blue_blobs = dict(enumerate([Blob(BLUE) for i in
range(STARTING_BLUE_BLOBS)]))
    red_blobs = dict(enumerate([Blob(RED) for i in
range(STARTING_RED_BLOBS)]))
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                quit()

        draw_environment([blue_blobs,red_blobs])
        clock.tick(60)
```

Now our`draw_environments` function:

```
def draw_environment(blob_list):
    game_display.fill(WHITE)

    for blob_dict in blob_list:
        for blob_id in blob_dict:
            blob = blob_dict[blob_id]
            pygame.draw.circle(game_display, blob.color, [blob.x,
blob.y], blob.size)
            blob.move()

    pygame.display.update()
```

Now we're taking this list of `blob_dicts`, where the dictionaries are key: id, value: object. We iterate through blob_dicts by id, and then draw and move the blobs as usual.

So, we have our final code - here.

Resource : pythonprogramming.net