Robots

Part one of this class was taught to Roger on Sep. 19th 2023.

This discusses the basic addition of map and player. In part two we add rocks and robots, and in part 3 we finish the game.

- Part 2
- Part 3

PyGame Console Starter

We begin with the standard PyGame console starter program:

- PyGame Terminal

Defining the World

The first step is to define the world. We start small, with baby steps. We want a map and a player which can move around. Add the following code after “self.running = True” in Class Game: init(self):

Initializing the Map

<Code:Python|Added to Class Game init(self):>

1. create map

self.gameW = 60

```
      self.gameH = 21
      self.gameMap = [[' ' for _ in range(self.gameW)] for _ in
range(self.gameH)]
```

1. add walls

for x in range(self.gameW):

```
          self.gameMap[0][x] = '#'
          self.gameMap[self.gameH-1][x] = '#'
```

```
      for y in range(self.gameH):
          self.gameMap[y][0] = '#'
          self.gameMap[y][self.gameW-1] = '#'
```

```
      self.px = 3 #random.randint(1, self.gameW-2)
```

```
      self.py = 3 #random.randint(1, self.gameH-2)
```

</Code>

I left code inside for a random position, but you can also just make the player start wherever you want for testing purposes. You will also have to set gameW and gameH based on your screen size and other factors. The numbers shown work well on my screen but yours may be different.

The two for loops will draw walls at the edges of the map.

Drawing the Map

Inside Class Game's drawMap() method we have the following code:

<Code:Python|drawMap()>

```
  def drawGame(self):
      # 1. draw map
      for x in range(self.gameW):
          for y in range(self.gameH):
              self.drawText(x, y, self.gameMap[y][x], "gray")
```

1. 2. draw player

self.drawText(self.px, self.py, “@”, “gray”) </Code>

Simply put, we draw whatever is in the gameMap to the screen, and then we draw the player. This is a really simple idea; we just draw the state of the game as-is.

Player Movement

First we have the checkEvents method, which is given hooks to move the player:

<Code:Python|checkEvents()>

```
  def checkEvents(self):
      for event in pygame.event.get():
          if event.type == pygame.QUIT:
              self.running = False
              return
```

```
          if event.type == pygame.KEYDOWN:
              # key down event, process keys.
```

```
              if event.key == pygame.K_LEFT:
                  self.movePlayer(self.px-1, self.py)
```

```
              elif event.key == pygame.K_RIGHT:
                  self.movePlayer(self.px+1, self.py)
```

```
            elif event.key == pygame.K_UP:
                self.movePlayer(self.px, self.py-1)
```

```
            elif event.key == pygame.K_DOWN:
                self.movePlayer(self.px, self.py+1)
```

```
            else:
                pass
```

</Code>

Again, keeping things light and simple. This is a feature of 'rapid prototyping', 'dynamic programming' otherwise formerly known as top-down design. You “pass the buck” by writing functions to do what you want to do in the next phase. The basic idea is that you write a custom programming language using functions. This idea should be expanded upon in at least two dedicated lessons!

The next thing we need to do is write movePlayer().

<Code:Python|movePlayer()>

```
  def movePlayer(self, to_x, to_y):
      # 1. Check map bounds.
      if to_x < 0:                to_x = self.gameW-1
      if to_y < 0:                to_y = self.gameH - 1
      if to_x >= self.gameW:      to_x = 0
      if to_y >= self.gameH:      to_y = 0
```

1. 2. Don't allow player to move onto a (blocking) wall.

if self.gameMap[to_y][to_x] == '#':

```
          return # do nothing
      # 3. Fall-through condition: move the player.
      self.px = to_x
      self.py = to_y
```

</Code>

Note that #3 could be written using else: as well; but then later if we want to introduce more checks it will be difficult to modify the code. Not difficult, just more difficult than this way. The fall-throuh condition style of coding is considered lower-level than using if-elif-else, and an even higher level is to use switch. If-else is lower level than if-else. This is based on how assembly laguage works. Although you could use a 'jump table' or a series of ifs to simulate a switch; which is why switch is a high level language concept– it doesn't exist in machine code and is a construct to bridge the gap between how we think or how we want to think when programming, and how the machine actually works. That is why higher level constructs are always slower, and why languages such as Python which are based around high level abstractions are always slower than lower level languages. The target level is different.

At this point you should have a working game which allows the player to move around, but there are some problems with the game. Yet, this so far is good enough for a small 1.5 hours class.