**#.py Source Code Companion**

Daniel McLaughlin

The purpose of this document is to provide some insight to the programming involved in the project #. Along with the source code itself, I am going to detail functions I created, what they do, and how they relate to Queer Game Design. I’m also going to discuss the ways these functions may be altered or enhanced in future iterations to provide more functionality. All the code in this project was written in Python, and also requires a “PIP” installation of python-twitter, credited to the Python-Twitter Developers <[python-twitter@googlegroups.com](mailto:python-twitter@googlegroups.com)> to work fully; though all the functions simply use return values from this package to work and would technically work with any form of data presented in textual form.



This function is used to convert text, referred to as a string, into a single number. Traditionally, games use a random number generator to produce many game decisions on the fly, such as “rolling” for a successful hit or trajectories of a bullet from a gun (by choosing a single trajectory from a ‘vector cone’). Machines are not capable of producing truly random numbers, and do so through complex algorithms that are packaged for programmers to use themselves. These algorithms vary depending on which “version” of the number generator you use, which vary between programming languages, the versions of these languages, and are sometimes influenced by the machine running the code. There’s one thing that each random function has in common – they each need a “seed” value to create a number. If this seed is a normal number, such as 10, the algorithm will ALWAYS produce the same outcome, since it’s a static function.

These algorithms typically take in the machine’s current system time in seconds since Thursday, January 1st, 1970. For example, as I’m writing this piece, the current Unix Timestamp is 1431925756. That number is how my machine knows what Month, Day, and Time it is currently. So, by calling a random number generator with that number, I am able to get any number within the range of positive integers (between 0 and 4294967295). The next step is to then take the modulus division of this result to get a number I actually want, which returns the remainder. If I want a number between 0 and 10, I would say “% 11”, which has 11 separate outcomes, the numbers 0 through 10. You can do simple addition to get ranges that don’t start at 0, or other operations for whatever you may need.

Now that we have a solid understanding of the way random numbers work, it’s time to discuss the way hashify queers this notion all together; which I use for random number generation in #. In my function, I read every single character inside the text that’s passed into the function, and take the running sum of each character after multiplied by a rather large prime number. For the English alphabet, A-Z then a-z, values begin at 65 for A and go continuously till z at 122. All characters have these values, allowing me to easily do this math to them. The max parameter of this function is used as the “ceiling” of the random number, as we see in the last line, we return this huge number modulo max. The use of the large prime number comes from a concept called “hashing” that is used for data storage techniques – but I don’t deploy these techniques, simply barrow from the theory. This is because this method allows for the possible outcome of remainders to be relatively diverse, where if the number wasn’t prime, we would have many occurrences of similar numbers despite all the multiplication through the idea of factoring.

Through all of this, I am able to take tweets from Twitter, read each character one by one, do this math to get a running sum, and then find a remainder that’s relatively random – but also heavily influenced by the data creating the number. This gives the randomness a context of whatever tweets it encounters, queering the concept of what a random number can be, because technically, the tweets are ultimately driving the number creation rather than the system time.



The above code are two separate areas, but are related. In the current iteration of #, the game draws a 5 x 5 dungeon of rooms, whose “walls” are determined by the hashify result of their message. We use the range of 0 thru 15 to determine which four walls, North, South, East, and West a single room won’t have a path through. This is because if we have 4 walls, and they can either exist or not, there are 16 total possibilities of configurations. In the definitions of northWalls, eastWalls, etc, we are telling which numbers would correlate to which wall. For instance, if the room ends up with a hashify value of 10, we have a south wall and west wall based on the fact that 10 appears in those two arrays. If we hashify to a 15, we have 4 walls and the room is impassible!

The second block of loops are demonstrating the use of the Twitter API to pull data from Twitter, using a (traditionally) random number from the “trends” variable and then getting an array of 25 tweets containing that trend. The second loop takes each of these 25 tweets and creates a new Dungeon Room object, containing the message of the tweet, and the wall value, and storing it into an array of Dungeon Rooms for our player to explore.

In this iteration, our 5 x 5 world is circular, meaning that if we have an open wall that exceeds the boundaries, it loops around (ex: if we are in cell 0,0 and we move south, we end up at 4,0 instead of -1,0). Also, each room is independent with wall creation and do not converge (ex: If I go from 2,2 to 2,3 – an eastward movement, 2,3 doesn’t necessarily have a west path to 2,2). These limitations in my code turned out to work towards a Queer experience, lending the user to explore a complex dungeon with paths that are not necessarily retractable.

The rest of the code in #.py are used as regular rules to run the game. Some helper functions, like hashtagify, ensure the game handles certain text and input in the ways I wanted it. The giant while true loop at the bottom of the code is the driver that accepts user input, prints the room information that the user is currently in, and moves the player around depending on if its “legal” or not. I also added random taglines to accompany the Twitter messages to make the experience a little guided. In future iterations, I hope to program more gaming functionality to support other modes of gameplay; which will ultimately require graphical support. It’s my hope to have several kinds of gameplay models available for these tweets to construct, so that the game can achieve poly-genre status by choosing a random genre at loading.