Contents

[0.30 – Kipling Update (Startscreen, Settings, DevMode, Interriors, Text Based Adventure 2](#_Toc24295069)

[Settings, Loadscreen, Devmode 2](#_Toc24295070)

[Pack-a-Punch (Bit of a Secret) 2](#_Toc24295071)

[Interiors FINALLY 3](#_Toc24295072)

[Text Based Adventure 3](#_Toc24295073)

[Auto Surrounding Descriptions (no interriors) 3](#_Toc24295074)

[Description 3](#_Toc24295075)

[Code 4](#_Toc24295076)

[Text Parser 6](#_Toc24295077)

[Description 6](#_Toc24295078)

[Code 7](#_Toc24295079)

[Colours 9](#_Toc24295080)

[0.29 – Dev Update (printT (special print), Map, Code Convention, CSV files) 9](#_Toc24295081)

[printT 9](#_Toc24295082)

[printT Code 10](#_Toc24295083)

[printT Performance 10](#_Toc24295084)

[MapDisplay 11](#_Toc24295085)

[CSV Saves 11](#_Toc24295086)

[Entities to CSV 12](#_Toc24295087)

[CSV to Entities 12](#_Toc24295088)

[CSV to Code 12](#_Toc24295089)

[0.28 – Music, settings, play testing and feedback 12](#_Toc24295090)

[0.28.1/2 – Save file problems 12](#_Toc24295091)

[0.28 – Music, Marginal Improvements, and Feedback 12](#_Toc24295092)

[0.27 - How the game works, The current structure, and the current best strategy to win 13](#_Toc24295093)

[How the game works/what I’ve learned about the Game 13](#_Toc24295094)

[The Game Structure/execution 15](#_Toc24295095)

[Game Initialization 15](#_Toc24295096)

[Game Structure 15](#_Toc24295097)

[Best Strategy to Win for speed running or otherwise 15](#_Toc24295098)

[0.26.1 – SpeedRun Update 16](#_Toc24295099)

# 0.30 – Kipling Update (Startscreen, Settings, DevMode, Interriors, Text Based Adventure

Include snippits of code and screenshots for things

## Settings, Loadscreen

The Startscreen has been improved with allowing settings to load from a settings.ini file in the base folder. This saves and loads in the GAMESETTINGS global dictionary on start. It saves settings now in a settings.ini file so whenever they are changed in the setting screen once you back out of the screen. This makes it much better so you don’t have to re-change settings every time. These settings are also saved in the game file and will automatically change over once you load them. These settings are saved in the datapath with is in “%Appdata%/roaming/EngPhysTextAdventure”. This makes it so can write to the settings even if you’re not admin. Also they’re a bit more hidden.

The loadscreen now allows loading from the start screen and loading different files other than your start name. This adds greater replay function to keep all the save files. Might later add other displays to the loadscreen which show your gear, quest progress, storyline etc like in Ocarina of Time. The other addition is in hiding the game file through the cache file being hidden. The cache/savepath is now in in “%Appdata%/roaming/EngPhysTextAdventure/cache/” which is not only more tucked away but can be written to even if they’re not an admin.

## Devmode

DevMode is a great help that disables the error catcher that runs over the whole game. This may not be the best way to do it but based on what the DevMode setting is you will through the error. This is great for development where you want to see the specific error but not for playing where you don’t want the screen to close. Dev Mode also gets rid of the starting blip and start screen. It also makes you play as the DEV character.

DEVVERBS = ['/stats', '/savegame', '/loadgame', '/restart', '/'] # lists of Verbs/keywords ONLY the developer can use  
DEVVERBS.extend(VERBS) # Combining all the normal verbs into DEVVERBS to make the extended list when in dev mode

DevMode also now has certain keywords that give information and functions that the vanilla game can’t. This includes keywords for stats, giving more information for inspecting + talking to things, as well as being able to save + load + restart + run scripts + show info from the game. The skeleton code for Creativemode has been laid with the parser being able to take in the / command then read, set, create, and remove. This will hopefully be expanded upon in future iterations of the engine.

DEVMODE Now is not a gamesetting but a separate DEVMODE.ini that the game checks for to boot you into game mode. It also has a quick disabler so if you type in n it won’t boot you into dev mode which is nice. Also is obfuscated and will refresh the screen.

DEVMODE has scripts again! Used to be able to paste log file into the command line but I think windows disabled it. NOW you can run scripts out of the ‘CWD/DEV/PlaythroughScripts’ with the ‘/script filename’ command. This will override the command input until the script is run through

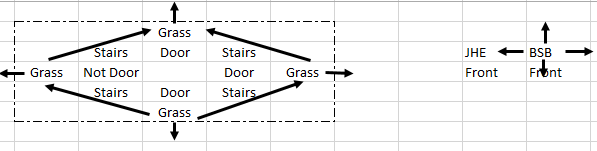
## Pack-a-Punch (Bit of a Secret)

This one is based on the time and also soon when you beat the whole game

Has a whole interface and inner works I hope is commented out

## Interiors FINALLY

This one may be a lot because it’s a lot of patchwork functionality BUT basically there’s an Overworld Map (the normal map entity files) that you



## Text Based Adventure

Based a dive into the design of the game through the feature lock had lead me to make this game much closer to an actual text based adventure. This involves taking out a lot of the RPG/MUD elements starting with the stats that display everything and break the immersion from the narrative. Based on this we are retracting the stats, map, sounds (except for start sounds), music, and Asscii art until further notice to not break the immersion. I do invision that a lot of these features could be re-implemented in a much more subtle way such as:

Stats- Instead of having explicit numbers for stats you can display certain adjectives or adverbs (which would also be cool to have random displays)

Map- This could be hidden in your journal that you see if the journal was invented.

Ascii Art – If the Ascii art was made scalable then it could still keep the immersion (also if it refreshes the screen it could be even more immersive so it’s not in your log).

## Auto Surrounding Descriptions (no interriors)

### Description

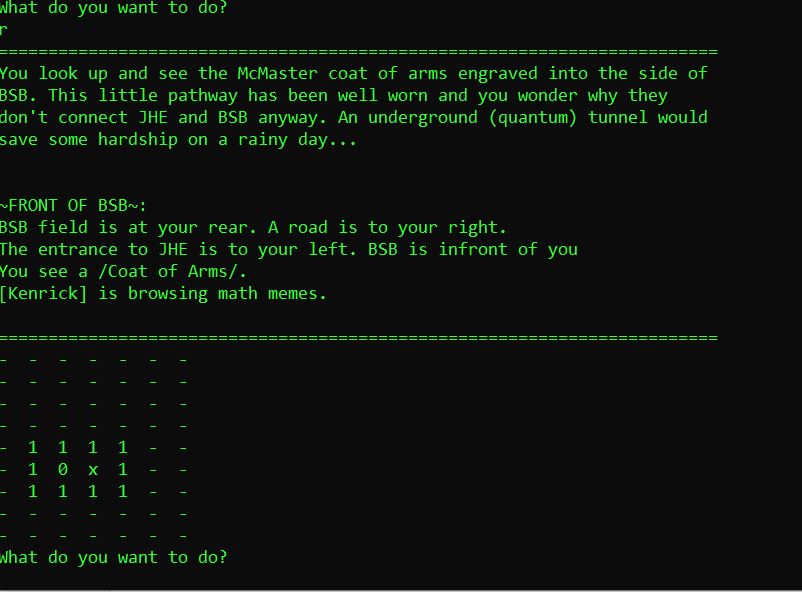
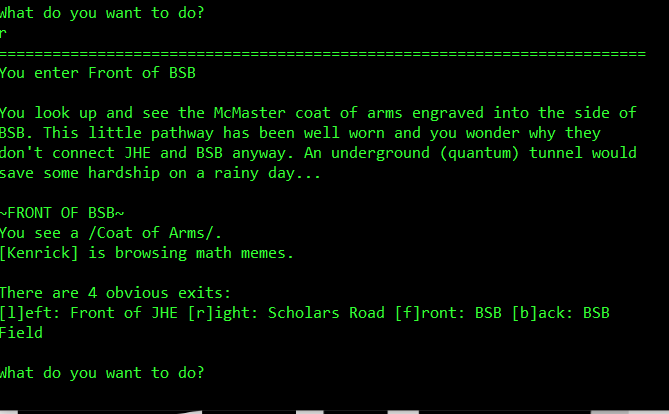
So I’m not sure why this wasn’t done before but now the descriptions of the surroundings print out automatically. The order and linebreak of the whole reading of the next is now re-arranged to fit this order:

1. IF First time visiting: Print “You enter \_” then MAP.lore

2. Now Search function prints ~MAP NAME~ + MAP.search (gives items and people and now directions)

3. What do you want to do input?

4. Linebreak

Go from this: To This:

To do this I got rid of using the map description and made the map object name the used item with the following MAP Name Convention:

Names are captalized when defined and printed initially. The name is surounded by ~ ~ and all captalized when printed through MAP.search. No delimters (the), prepositions (in, outside), or qualifying words (very). Use directions with reference to areas. Just make the name a very clear word that you would understand and want to read quickly.

ex) Front of Hatch. Behind Art Museum. JHE Lobby

NOT example) Outside Hatch. Outside Art Museum. Inside JHE

exceptions) THE Phoniex (where it's a name)

You can now access the description of the area/surroundings using look, look around, or search. Added a new word “remember” that also prints out the lore.

It should be noted that this doesn’t support interiors or links but it may be coming

### Code

“Doesn't need to be wordy, just need to know information” – Brian M.

To understand the code we need to start in the main file (EPTAvX.XX). Whenever you move or look around the MAPS[x][y][z].search method is called which returns a text description to be printed. We use the printT function to print this so the spacing is correct. This means that also we can’t use newline characters \n but need to use (\S) to let printT know we want a new line.

print "You enter " + place.name + "\n"  
printT(place.lore)  
printT("(\S) ~"+place.name.upper() + "~ (\S)" + place.search(MAPS)) # (\S) used for printT newline  
place.travelled = 0

^ from the move function in GameFunctions.py

The .search method is in the GameClasses and is quite extensive. This can also make it quite daunting but it has 3 basic parts:

1. See how many items/interacts there are at a location and add a description of them to the description variable.

2. For each enemy print that they are there and give a random description or death description and add each to the description variable.

2b) If there is nothing in the description variable there is nothing there.

3. NEW Search for the surrounding areas and put them in an ordered list.

location = self.location # gets coordinates tuple  
letterdirections = ['l','r','f','b','u','d'] # letter based list of directions to check against walls  
tupledirections = [(-1,0,0,0),(1,0,0,0),(0,1,0,0),(0,-1,0,0),(0,0,1,0),(0,0,-1,0)] # tuple based list of directions to add to current location  
surroundings = [None] \* 6 # Name storage, defaulted to none. Order of: Left, right, Front, Back, Up, Down  
i = 0 # Counter for direction indexing  
for direction in letterdirections: # Looping through all the directions  
 if direction not in self.walls: # seeing if the way you can go is in the walls  
 # Gets tuple of requested adjacent spot by adding the direction in the right order  
 dx, dy, dz, dim = tuple(map(operator.add,location,tupledirections[i]))  
 if MAPS[dx][dy][dz][dim]: # if the map location exists  
 surroundings[i] = MAPS[dx][dy][dz][dim].name # store the name into the surroundings variable  
 i += 1

Based on what’s in the ordered list print out the surroundings based on:

# Short Description  
worddirections = ['[l]eft','[r]ight','[f]ront','[b]ack','[u]p','[d]own']  
description += "(\S) (\S)There are " + str(6 - surroundings.count(None)) + " obvious exits: (\S) "for i in range(6): # use index to reference direction  
 if surroundings[i]: # if the direction is seen  
 description += worddirections[i] + ": " + surroundings[i] + " " # print the word direction + name

The output should look like:

There are 6 obvious exits:

[U] 2nd Floor BSB [D] Quantum Tunnel

[F] Police Station [B] Front of BSB

[L] JHE Lobby [R] GSB

There’s future intentions to have an alternative wordy printout seen below. Also it might be cool to have a discovery mechanic (like with maps) so you need to discover the surrounding areas but we’ll see.

Some more description improvements would be to have a wordy nice description for the first and second time you travel to a location using that MAP.description field. ALSO following proper Interactive Fiction rules EVERY item that is mentioned in the lore should be intractable or it is unsatisfying.

*Potential Wordy Printout*

There are \_ obveous ways to go: Left, Right, Front, Back, Up, Down

To your left is \_. To your right is \_

Infront of you is \_. Behind you is \_.

Upstairs is \_. Downstairs is \_.

## Text Parser

### Description

It wouldn’t be an old school text adventure without a HANDMADE TEXT PARSER. Also known as a homebrew parser <http://www.ifwiki.org/index.php/Homebrew_parser> . It’s no where near a NPL, natural language processor, usually involving AI and deep learning to understand all language. Also isn’t as versatile as an infocom-type parser (used in zork and made by INFOCOM) which can interpret multiple actions and find the object, indirect object, articles, pronouns, etc in a sentence. That may all be possible with a sub-word search style parser but for now we’re just doing it for NOUNs to give some nice shortcuts.

The textparser has been moved from the main EngPhysAdventure.py file to it’s own TextParser.py to clean and functionalize the code.

The basic layout of the parser is broken down below:

Verb Declaration

Parser Function()

1 word commands

Accepted Commands

2+ word commands - usually verb noun (aka object) combination

Shortkey object replacement

Object Sub word Search

All of this is to attempt to find the right noun based on what the user types in.

Setup and Exceptions

Full word Match

Filtering Duplicates

Matching and Returning

Final Spellcheck if nothing is find

Accepted Commands

This all may same convoluted and it is so I’m sorry for the inefficient and tough parsing. There is tons of room for improvement and maybe a full rewrite if necessary but that will not be a mission of Bren007pie.

In addition, to make the parser juicier I’ve added a bunch of accepting words and some good responses to any weird combination in the game. I think it really gives some depth and shows thought into the interactions that happen in the world.

### Code

As stated the spaghetti code that is the text parser is basically an extended version of a 2 word parser. This is predicated on verb + noun/object structure to interpret commands and what they are done to. In our implantation the noun can be multiple words whereas the verb is almost always one word (with a few exceptions like long movement commands, look around, pick up, say, and creative commands which are not yet fully implemented). As such the code is segmented into single word commands (only a verb) and multi word commands (verb + noun). In my documentation however I’ll be separating the difference in how verbs are handled and objects are handled.

The first thing you need to know is that the parser splits the input text at the first space, theoretically separating the verb and the object. It then removes any excess spacing.

wordlist = command.lower().split(" ", 1) # Split at first space for verb

for i in range(len(wordlist)): # Getting rid of the spaces in strings using .strip()  
 wordlist[i] = wordlist[i].strip()

#### Verbs & 1 word commands

Section structure:

Verb spellchecking

Verb Parsing/commands

Verbs are the keywords that drive almost all the commands. While this simple 2 word parsers doesn’t make for good English it’s one of the simpler implementation to homebrew.

The VERBS list is defined up at the top of TextParser.py which state all the recognized verbs in the game. These are all the verbs the game will spell check against if you mistype something. UNLESS there’s any exceptions you must define a verb up in this list for it to be accepted in a logic statement or else it will be corrected by the spellcheck. Many verbs will be used redundantly to mean the same thing in logic. There is also a DEVVERBS list that extends the normal list to use when you’re in Dev Mode.

Before the actual parsing of the verbs there’s the spellchecking section. In the current section. If the word is longer than 1 character (not a shortcut), not the devcommand, not in VERBSHORTCUTS then it gets spellchecked. If you’re in Dev mode it gets checked against DEVVERBS, if not checked against the vanilla verbs.

# --- Verb Spellchecking ---  
if len(verb) > 1:  
 # if dev mode enabled it accepts special verbs which allows you to use special functions  
 if verb == '/420e69': pass # Does no spell checking so someone doesn't accidentally get 420e69  
 elif verb in VERBSHORTCUTS: pass # Does no spell checking if it's a shortcut  
 elif GAMESETTINGS['DevMode']: verb = SpellCheck(verb, DEVVERBS)  
 else: verb = SpellCheck(verb, VERBS)

General Parsing

As much as I wish it was something more elegant (any ideas?) the basic of the parser is the logic statements like this one.

elif verb in ['i','inventory']:  
 Inventory()

Where Inventory() comes from GameFunctions.py (you don’t need to use GameFunction.Inventory() because we use “from GameFunctions import \*”. In the future I don’t want functions like this to be in gamefunctions, because they are general they should be methods as part of gameclasses but that would be for rebuild.

A more advanced function might utilize the game dictionaries

elif verb in ['us',"use"]: # this makes it so you can use items if the interacble is in the area  
 x, y, z, dim = PLAYER.location  
 # checks all interactables in area to see if item is needed  
 for interactable in MAPS[x][y][z][dim].items: # for all itmes+interactables in the area  
 if isinstance(interactable, Interact): # if it's in interactable  
 if interactable.need == objectName:  
 print "\nYou use the " + objectName + " with the " + interactable.name + ".\n"  
 Inspect(interactable.name.lower())

This little chunks of if statements are what I refer to when I talk about accepted commands. Many of these are straightforward.

#### Objects/Nouns & 2+ word commands

Section structure:

Verb Spellchecking

Shortkey Object lookup

Object subword search and spellchecking

General parsing

The object parsing isn’t nearly as straightforward. The same verb spellchecking applies in the beginning. The shortkey object lookup basically gets your inventory in surroundings and then based on the order of the printout of the surroundings it gets that index and sets the objectName to the proper index.

## Colours

ANSI ESCAPE CODES!

Unfortunately, there is no bold like Linux but there’s still tones of options!

Colorama is a nice package from python that handles all the colouring of the foreground and background. The basic idea is that when you print one of these escape codes to the console it will set the foreground or background colour until another escape code is scene. It’s a very generic application of the colorama package so you can look it up or see the links below.

<https://pypi.org/project/colorama/>

<https://www.devdungeon.com/content/colorize-terminal-output-python>

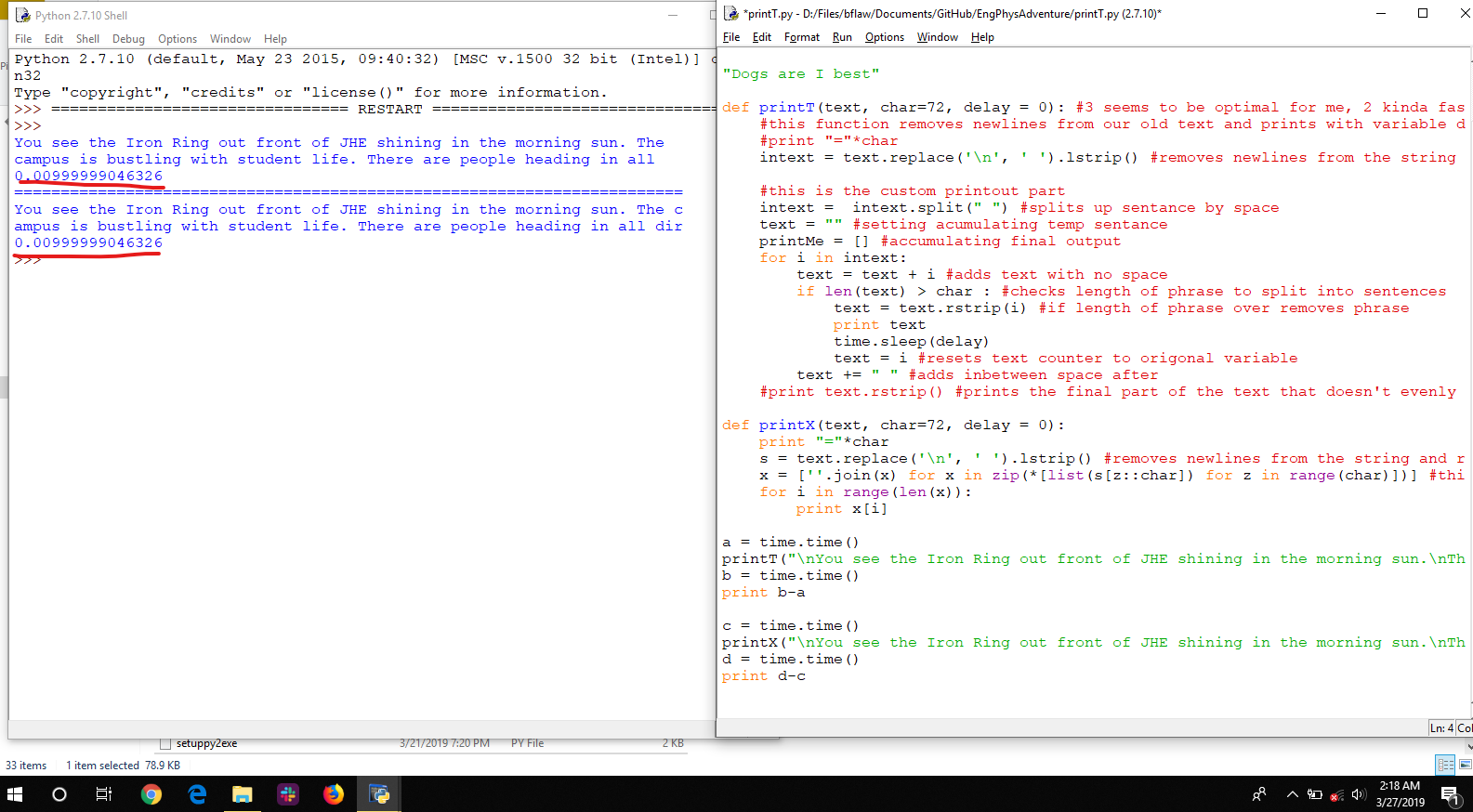
I think there’s ability to make your own console completely from ANSI escape codes (probably just like a refresh rate thing). Has tons of potential but I’m not going that far into it. Would be cool though to have autocomplete, do fun things to the user, and have more colours available though.

# 0.29 – Dev Update (printT (special print), Map, Code Convention, CSV files)

## printT

PrintT(text, character limit, delay) is a custom line printing module for the game that will allow for using old written lore with manually entering newlines or text blocks with newlines at all (as long as there’s spacing between words). This function automatically scales to any defined character width screen, can have custom delays, breaks up paragraphs automatically (as long as using 5 consistent . ! ? punctuation marks), and can be overwritten using an arbitrary delimiter (\S) to start a new line or (\S) (\S) to enter an empty line with a delay in-between. I maybe have spent more time creating this (5 hours) than total time saved in waiting but I learned a lot about python text parsing, function pass by reference/ global variables/proper non-global variable use, doc strings, and performance testing.

Although I should have done some research before creating it I simply assumed this would be too specific an algorithm to find. I did use an algorithm found online to try to parse the characters quickly but in the end gave no performance difference on the single test I performed (see below). I will talk more to the performance after.



### printT Code

The code uses a simple non-intelligent algorithm remove all new lines and replace with spaces and then to split all words in the input text into a list of “words” with no spaces. “Word” is in quotes here because it may contain punctuation or any other manner of things, its just separated by spaces so it’s key to make sure that nothing is jumbled together or else it might overflow (or at least the longest word would be the character limit. Then it basically loops through all the words and builds up phrases one word at a time until it either matches or exceeds the character limit. If it exceeds it that overflowing word is removed and added to the next phrase with all extra spaces removed. The paragraph sensor works by just counting exclamation points, question marks, and peroids in each “word”(with the exclusion of Dr.) until it reaches 5 then stops the phrase and adds an extra line. If more exclusions and specific words need to be added they can be. The (\S) parser took some time but in the end simply looks in each “word” and if it finds the (\S) in it removes it and ads the beginning of the word to the end of the phrase before printing and clearing it. IF there was a word after the (/S) it ads it to the next phrase unless it’s a space.

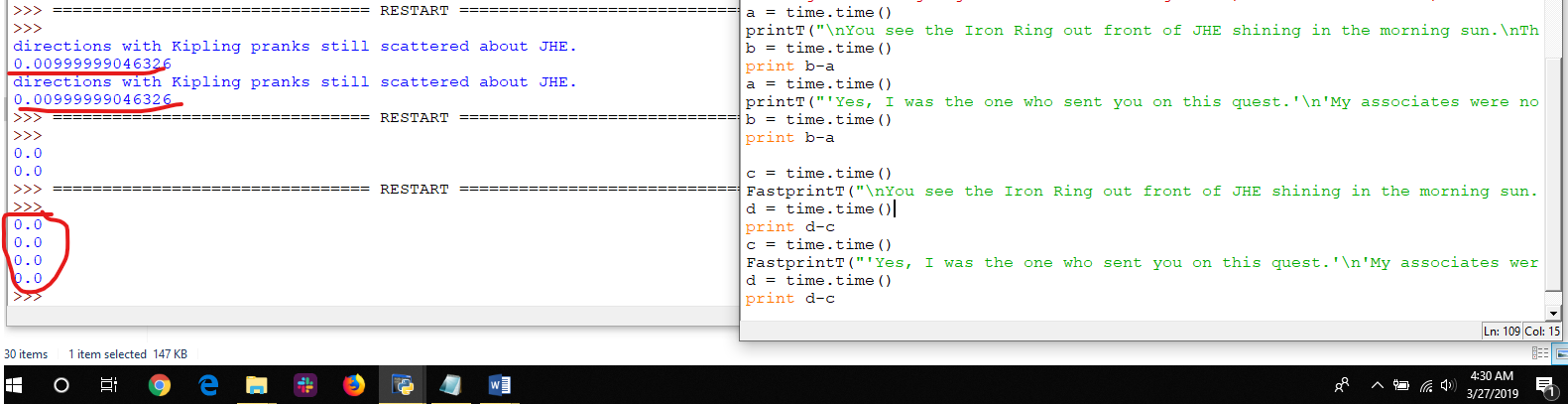
There is some tricky spacing and parsing implementation that took me some time and I coded in jank manor. So for further implementation and examples see the code.

### printT Performance

Using the time method, tested differences in the function in a tic-toc fashion before outputting them. Subject to processor load noise but ran multiple times to confirm input.

I tested the full printT and then a reduced version with less checks to look for a speed difference and saw none, even zero, within cpu noise time.

It was discovered that even on a long paragraph all the checks and looping through words took no identifiable time as seen below. The main delay was from printing output so once the printing was silenced the time for calculation went to zero. This shows that most of the algorithms (on the scale of EBTA) are probably limited by the text output so should not worry too much about scale. But this was fun and reassuring to see. Possibly there will be some calculation time over the course of the whole game text but this is hard to test.



## MapDisplay

Map Display prints out a readable “mini” map which shows areas that have been explored, not explored, and where you are. Kinda similar to the in-browers asci art maps from “A Dark Room” and I’m sure many other games. I made this to not only have a nice display but to see how people work with a map

Basically the map uses two attributes/flags in each of the map objects. The Travelled flag was used to display different lore but now is used to display if you’ve been to it or not (“0” for travelled and “1” for not travelled). The new mapped flag is used to display if the player has discovered that area (“1” or “0” is discovered where as “– “ is undiscovered or not a spot at all). When a player moves around he flips the travelled spot on the spot he’s on and discovers map locations in a radius around him. The radius of discovery (default 1) gets bigger the higher you are so basically if you go on top of a big building you can discover most of the campus. The one caveat is that you can’t discover anything in the basement or above the second floor without travelling to it. This prevents you discovery any secret locations or knowing how many floors a building has (player has to discover that).

For the code there are two main loops: 1. Does the discovery mechanic around the player. Should be explanatory.

2. Displays the mini map and prints it out. The only part to be aware of here is the order of the “if” statements which mark the priority of display. The map object may fit multiple conditions but you want it to stop and the most applicable one so it displays the way you want it to.

The rest of the particulars should be explanatory through code comments.

There is a lot more functionality that could be added such as scaling, rotating, and different types of maps but I anticipate moving to a GUI output instead of terminal so I didn’t want to go too far into development.

## CSV Saves

This was copied from the function description:

*This is a development file used to easier develop all content in the game.  
It's used to export all objects and attributes to a CSV file which is much easier to write in Excel than on an IDE.  
This will also give a nice visual display to see issues, spelling/format mistakes, and map layout.  
After writing it can be imported back on runtime or converted back into object declaration code which can be copied  
into the startup file.  
  
Side note on Why using Pickler and CSVs: If a pickler saving/loading to file is already in the game why not  
use one or the other? Ideally all data is saved in CSVs and then encrypted/de-encrypted. But for now the workflow is  
to develop using the CSVs and excel then copy the code back into startup for compilation.  
This makes it easier to develop but a bit more of a pain to pass back and forth and debug.  
Also, why not do the CSV -> python conversion in Excel? You would have to understand Python and VBA  
which some developers do but should be expected in order to change this code.  
Also saving different classes to different files because it makes it easier to read.*

### Entities to CSV

This is a fairly straightforward module that writes all the attributes of all the entities to different CSVs. There’s a different csv file for each object class. It makes use of the common Python CSV writer. The code for each CSV file is the same but the implementation for each is slightly specific so it’s commented for the first item writer and the special map.

The Special Map printer takes the map and puts it into a nicely formatting display for debugging areas and checking walls. Can also be used for map design. See the code for the particular implementation, it can be dome more efficiently but I decided to just make it work.

### CSV to Entities

This module still needs to be done to read in the code

### CSV to Code

This module takes the contents of the CSV and creates the object declarations that are seen in the startup file. It basically reads in the CSV file and then writes those attributes to an object declaration string in another file. That STARTcode.csv file can then have its contents copied to the STARTUP file.

# 0.28 – Music, settings, play testing and feedback

## 0.28.1/2 – Save file problems

After fixing the save file from moving the quest file has shown mean just how violtile/finicky this game is that part of it breaks just by me forgetting how to do dictionary’s correctly. Also how I don’t have full handle on global variables and how we should not be using them at all to make it more clear.

The main counter to making sure it works properly is play testing with different scenarios including: completing the evil mode, good mode, the completions mode, and testing save functions.

Also moving the quest mode to 0.28.1 made the end of the game break with having to call the function.

## 0.28 – Music, Marginal Improvements, and Feedback

Made a bunch of small improvements and mainly music. Music is unfortunately just a simple player that is threaded that can’t stop or play other music. The best option I see for threaded music (without making our own thread which is reasonable) is figuring out how to get pygame to compile in a reasonable way.

Moving the quest file to another function I had to shuffle around lots of thing including the imports but I did it in a similar way to the creative mode imports. This also made problems with the definition of quests global function and saving because I forgot how save files work.

Altogether not too much big structure change but a lot of minor details that added up including making the settings a global function that are defined in the opening file. Trying to keep the program modular but hoping with further help the game can be greatly improved in code structure and best practices for reading/understanding, and making changes without breaking everything. Might help to make node charts to show how everything is connected so when changing things test everything connected.

Still want to personally figure out global variables and how we can use object inheritance to our advantage. Other than that figured out how to do a mid-way custom print function that is basically just a mod overlay for the current print function. Also got a lot of good feedback to be implemented for Kipling!

Current Green console settings in R,G,B is (52,255,52)

# 0.27 - How the game works, The current structure, and the current best strategy to win

## How the game works/what I’ve learned about the Game

So while working on the game to make the save modes and restructuring the game I figured out how the game works in terms of structure (which isn’t very apparent on first glance) so I wanted to document that and my changes.

The main thing to know about the game is the variables and the structure. I’ll start with the variables which I have not changed except for adding one global dictionary called GAMEINFO.

The backend of the game was mostly written by Mitch Lemeiux and utilizes dictionaries and objects. For those unfamiliar as I was a dictionary in python is simply a list that is indexed by a key instead of it’s number.

For example

Animals = [ “Dogs”, “cats”, “frogs”] where animals is a list of items where you acesss the string “Dogs” by using x[0] where it’s the 0th element

Animals = {“playerpet” : “Dogs”, “npc pet”: “cats”, “extra pet”: “frogs”} where animals is now a dictionary with syntax { key : value}. So to access the string “Dogs” by using x [“playerpet”].

Dictionaries are valuable to use over lists because it’s more apparent in the code what information you’re trying to access. For example it’s much more clear to know what item you’re referring to if your dictionary is keyed by the item and you see ITEMS[“Eng Phys Shirt”] vs ITEMS[362].

The game also uses object oriented programming (OOP) with several game class to define the types of objects used. Hopefully the reader of this is familiar with objects and understands the advantages of using them so I will not go over them.

The main variables in the game that store all the information are also global (defined in the top of the GameFunctions.py file except for QUESTS which is at the bottom) and are in allcaps. These variables are global so they can be passed between functions by simply definining them in the function as “global PLAYER” for instance to be able to access the player’s attributes. Otherwise variables defined and used in the function are local to that function unless passed as an attribute to the function and then returned by the fucntion. The variables are acessed by the main file by using “from GameFunctions import \*” which imports all the variables, functions, and imports of the GameFunctions file.

Bellow is the definition of the main global variables in the game

1. PLAYER #The main character. player is an object instance of class character.
2. ITEMS #All the items. This a dictionary of objects of class equipment keyed by their lowcase equipment name (item.name). Remember the lowercase, may trip you up if referencing upercase version in the file.
3. MAPS #All the locations. A tuple of objects of class Map inxed by there x,y,z coordinate (MAPS[x][y][z])
4. INTERACT #All the interactables (stationary things that need something). This a dictionary of objects of class Interact keyed by their lowcase name (interact.name). Remember the lowercase, may trip you up if referencing upercase version in the file.
5. QUESTS #Quest statuses. This is a dictionary of flags (1 or 0) for the status of the quest keyed by quest name.
6. ENEMIES #All the npcs. This a dictionary of objects of class Enemy keyed by their lowcase equipment name (item.name.lower()). Remember the lowercase, may trip you up if referencing upercase version in the file.
7. GAMEINFO #Miscellaneous game info. Dictionary of all sorts of variables

You can 5 of the main variables are dictionaries which store the instance objects of all the different classes for the whole game. PLAYER is the main player object that is your player and MAPS is a special array that is indexed by it’s [x][y][z] coordinate for example MAPS[x][y][z] is one location (to be honest I’m not sure exactly how maps works but I’ll let you know when I do).

The dictionaries are filled with instance objects from each of the classes seen bellow. They all have different attributes (object variables) and methods (object functions) which a full list that can be seen in the GameClasses.py file.

1. class Equipment:
2. class Character:
3. class Enemy:
4. class Interact:
5. class Map:

Knowing these basics, seeing how it’s done in the code, and following the description in the Github readme (see copy pasted below) you should be able to follow how to add things to the game (but ask questions if you don’t).

In general try to keep this structure and put any other long ascii display or mode into another file:

EngPhysAdventure \_\_\_\_ = the setup, main loop, and ending. Run this to run the game.

GameFunctions.py = The main mechanics of the game and the quests. All non-class functions. Where the global variables are dfined

GameClasses.py = Class definitions and their coresponding methods.

Startup.py = All the map locations, items, npcs (called enemies), and interactables are defined. Also creates the dictionaries of them.

AsciiArt.py = Where all of the ascii art display files are

## The Game Structure/execution

Because the game as become much more non-linear with the ability to restart, save, continue at the end, restart at the end, and even play the game in the game I want to outline how it now runs from variable initialization to how the game runs while you’re playing it.

### Game Initialization

### Game Structure

One thing to say about the game right now is it seems very unrobust. If varaibles and definitions aren’t in the right order in the code the game will break or glitch. This makes me suspect that with increasing complexity the game will become even more finicky which may be able to be solved with better organizations/functional code. If there’s anything you see that looks bad/unrefined/can be done better please feel free to change it in a spate branch. I am by no means the most experienced at writing code, using python, or software development. In the future to make it easier manage the software development side I’d like to use better defined test cases, follow a proper python style guide, and think about code optimization

-Brendan Fallon

## Best Strategy to Win for speed running or otherwise

If doing Tyler Kashak just beat all the people. If otherwise should do the nuke quest first as it’s the easiet to get the gamma glove before having to face the tough Kenrick and Dr. Soleymani. From there best to do the silicon quest and then the optics last. Can get it down to five minutes if you want

-Brendan Fallon

# 0.26.1 – SpeedRun Update