# General description

The chatbot will be based off of “The Witcher” series, focusing mainly on “The Witcher 3: The Wild Hunt” video game. The Witcher 3 is a video game with complex lore where the primary goal is killing various monsters. These monsters will usually have a set of weaknesses which can be exploited by the player to kill that monster more easily. Some example can include certain oils that the player can craft and apply to their weapon and signs (a sign is effectively a spell) that the player can use such as Igni (Sign that produces a small burst of fire). The chatbot will allow the user to ask some questions about the game, for a description of the enemies in the game and what weakness they have that the player can exploit. The chatbot will also allow the user ask if an image contains either a bear, wolf, dog or horse. The chatbot will also generate images of bears if the user asks them.

System requirements:

* The chatbot should allow for some “small talk” such saying hello and asking how they are
* Chatbot should allow the user to tell them their name and remember it in future outputs
* The chatbot should allow the user to ask for a description all of the different monsters
* The chatbot should allow the user to ask for a list of things that a particular monster is weak to
* The chatbot should be able to answer some basic questions about The Witcher 3 and the Witcher series
* The chatbot should allow the user to exit the program by saying goodbye
* The chatbot must be able to classify an image that the user specifies as being either a bear, wolf, dog or horse
* The chatbot must be able to generate pictures of bears when the user asks

# AI techniques:

## Chatbot components

One of the AI techniques that will be employed is Rule based. I will use an AIML file to define patterns for user input. When the user input matches that pattern, there will be a defined template to decide what the output should.

Another technique that was used was TF\*IDF, using the bag of words model and Cosine similarity. This is used to calculate how similar a string is to a set of other strings. TF is term frequency. This refers to the number of times a term is present in a particular document. A document in this context being a predefined question or the input from the user. IDF is inverse document frequency. This is calculated by doing the number of documents divided by the number of documents that contain a word. The Log function is used to dampen the weighting of the IDF score. Tf and idf are represented as matrices and multiplied together. The result is a matrix containing values close to 0 which is used to determine the importance of each word.Cosine similarity is then used calculate the similarity of each document to each other document in the in the data set. Cosine similarity works by applying a formula to two matrices to work out the cosine angle between them in a multi-dimensional space. The smaller the angle, the less of a difference between the two documents. This is used to find the question with the most similarity to the users input.

## Image classification components

The main AI technique used for stage 2 is a Convolutional Neural Network which is used to create an image classifier. Neural Networks work by building a model of nodes with 3 main types of layers. An input layer, multiple hidden layers and an output layer. The network is then provided labelled data, known as training data which is reshaped to fit the nodes of input layer. Each node in each hidden layer takes a sum of the nodes at the previous layer and multiplies the value of every node by a weight to determine how influential each node at the previous layer is to the current node. An extra number known as a bias can be added to this weighted sum to influence how high this weighted sum of nodes needs to be before the current node is significantly activated. This weighted sum + bias is then put through an activation function to constrain it within certain values and decide how activated the node is**.** At the output layer the node with the highest activation is considered to be the best fit. These output nodes will usually correspond to different categories or classes that the programmer defines. During the training process, if the output does not match the label of the data, back propagation is used to alter the weights and biases of the hidden layers to alter the activation of the outputs nodes to be closer to what it should be.

Convolutional Neural Networks are Neural Networks that use image pixel data as the input layer, and convolutional layers for its hidden layers. The convolutional layers apply arbitrary kernel convolutions to manipulate the image. A kernel is a grid of numbers. Kernel convolutions work by sliding the kernel over the image, multiplying each pixel value by the value in the kernel currently covering it, calculating the sum of these values and writing the sum to a new image in the position of the pixel in the middle of the kernel.The theory is that once these kernel convolutions have been applied, it should highlight the features which are important for classifying the images.

## Image Generation

The chat bot also makes use of a Deep Convolutional Generative Adversarial Network (DCGAN) to generate new images of bears. DCGANs are formed of two Convolutional Neural Networks. A Generator and a Discriminator. A generator takes random values as its input and applies arbitrary convolutions as its hidden layers to attempt to produce a plausible looking image as an output. A Discriminator is trained to tell the difference between real and fake images, by training on real images samples, and fake image samples provided by the Generator. Iteratively, the discriminator should get better at detecting real and fake images, and the Generator should get better at fooling the Discriminator into thinking the image it has produced is real. This results in the Generator producing better quality images over time.

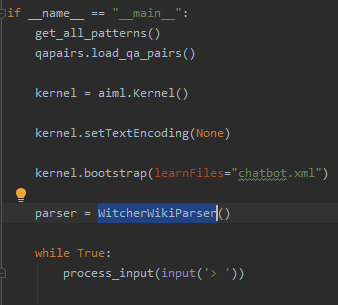
## Inference engine

An inference engine is uses an established knowledge base of facts to infer whether or not other facts are true. The chat bot uses the nltk library to create an inference engine which can load a knowledge base of First Order Logic statements to infer facts. For example, if the user queries the chatbot with a statement such as check that x is y, the chatbot will use the knowledge base to infer whether or not that fact is true, false or inconclusive according to the knowledge it has available. The nltk library uses resolution to prove if a statement is true or not. Resolution algorithms to check if a statement is true or not by using proof by contradiction. This works by adding the negation of the statement to the knowledge base and checking for any contradictions. By proving that the opposite of a statement contradicts the knowledge base, you prove that the statement must be true. This will not prove if the statement is false, only if the statement is true or cannot be satisfied. To prove that a statement is false then you must run the same proof by contradiction algorithm but starting with the inverse of the statement. If you prove that -Q is true, then that means Q is false.

# Explaining the code

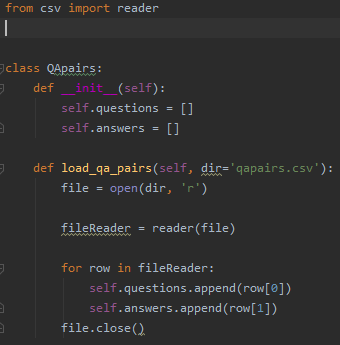
## Chatbot components

At the beginning of code being executed a few different variables are initialised.

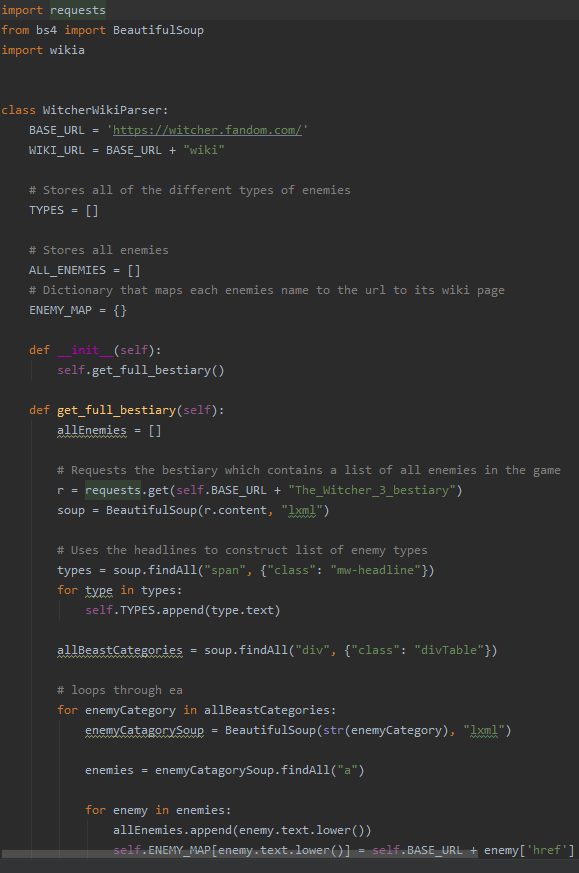


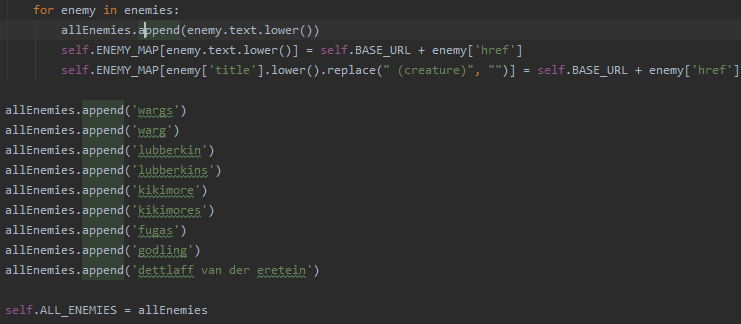
The kernel is initialised and passed the AIML file.

A function is called to populate a list two lists with questions and answers from the csv file:

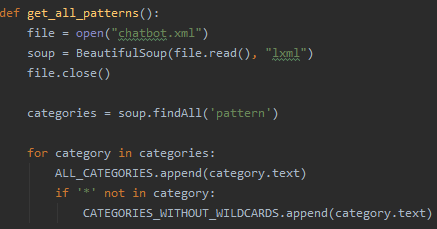


An instance of the Witcher wiki parser is created. This is used to get descriptions and weaknesses of enemies. The constructor pre loads a list of all of the different enemies, as well as a dictionary mapping each enemies name to it wiki page url.





Back in the main function, another function is called to load all of the patterns in the AIML file. This will be used later:



An infinite while loop calls the process input function, passing in the console input as a parameter



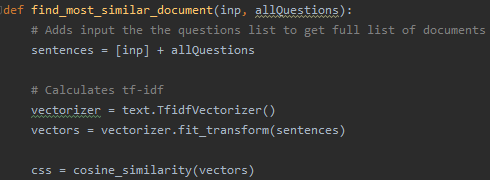
The process input function is where the chatbot logic is defined. Here the user input is stripped of any punctuation, converted to lower case, and passes the input to the kernel. The output is then processed, using a $ to signify that the input needs further processing and # followed by a number to determine which function is run what numbers and parameters is returned is determined by the AIML file:



The default reaction in the AIML file will signal the python code to do tf-ifd and cosine similarity. First the users input, questions from the csv file and patters without wild cards are passed into the function.



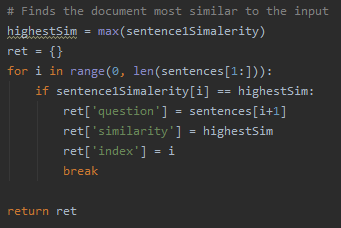
These three inputs are concatenated into one list. The Sci-kit learn package is then used to perform the TF-IDF and cosine similarity operations:



Once that has been done, the input similarity vector is extracted from the matrix. Its similarity to itself is stripped from the list



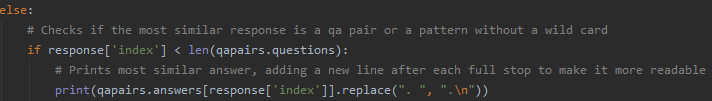
The most similar question is found. The question its self, the similarity score, and its position in the list is returned by the function in a dictionary:



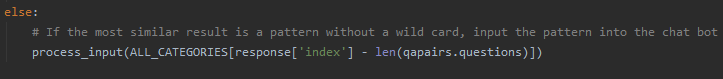
If the similarity score is 0 then the chat bot will ask the user to try and rephrase there sentence:



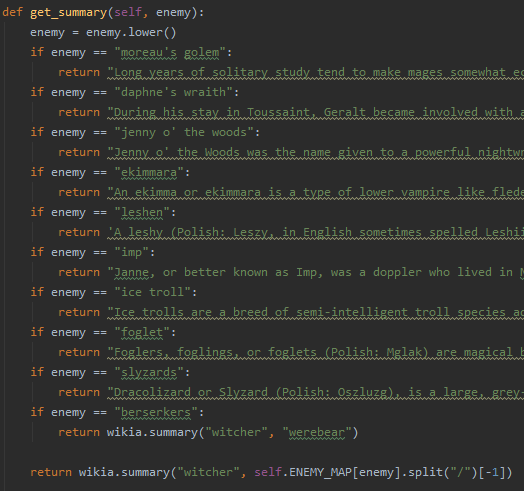
If the most similar answer was in the Q and A pairs, then the answer for that question is indexed and printed by the chat bot.



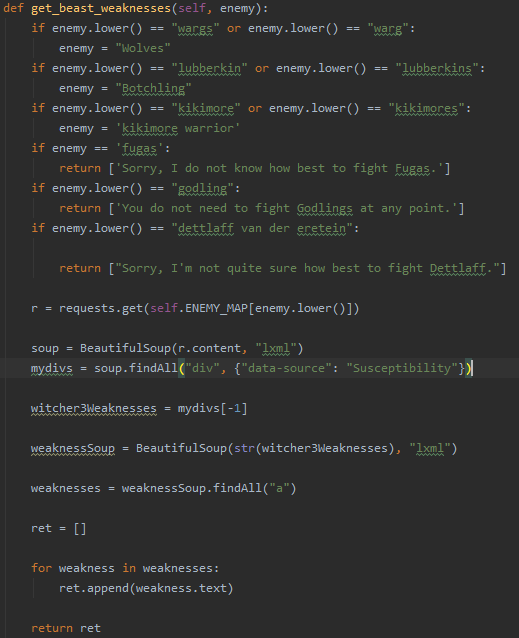
If the most similar document was one of the patterns that do not contain any wild cards, then that patter is entered into the chatbot again. This allows AIML file to be simplified and not include lots of very similar patterns where the patterns do not have any. Implementing this with patterns with wild cards would be slightly more difficult.



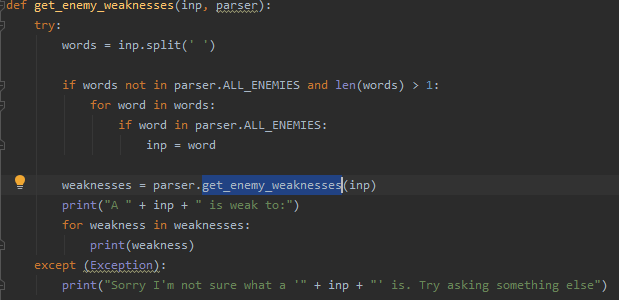
If the users requests a description of an enemy, the wikia python package is used to get this from the Witcher wikia page. This did not work with all enemies, hard coded return values where returned in these cases.



The wikia package did not have a working way that I could find to extract the weaknesses of an enemy. When the user requests the weaknesses of an enemy, wiki page for the enemy is parsed to scrape the weaknesses. Work arounds have been included for enemies that did not have susceptibility on their wiki page:



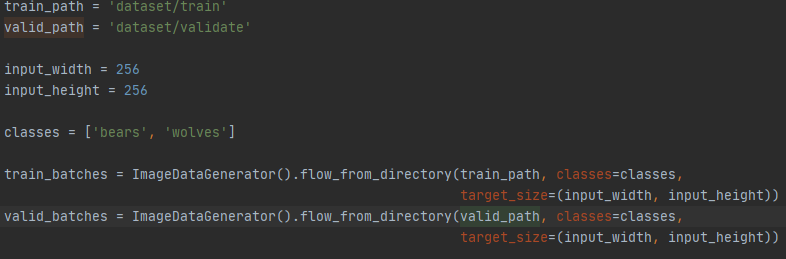
Both functions which call the get weakness and get description are very similar and use a snippet of code which tries to find the input in the list of all enemies. If it cannot be found, then it will try splitting the input into words to see if any unwanted words were also caught by the wild card:



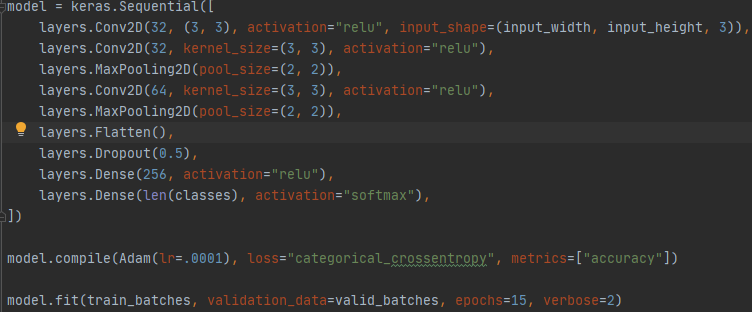
## Image classification components

The Python script which trains the classifier is in Train\_CNN.py

First the training and validation data is loaded into image data generator

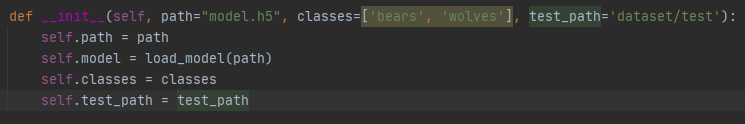


The model is constructed, compiled, and then trained on the training data

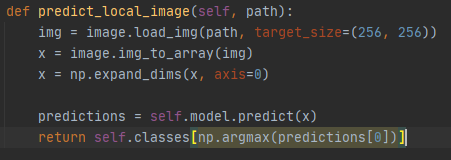


The file Trained\_CNN\_Wrapper contains a class which wraps the code required to load the trained model and make a prediction on an image.

The constructor loads the model

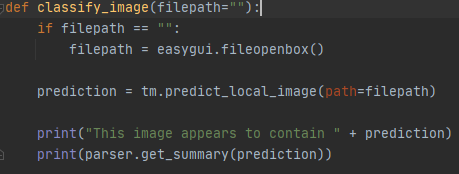


The predict\_local\_image member takes a file path as an argument, loads the specified image file, processes the image into the format that the model expects, and runs the image through the trained model, and returns the class that the model predicted.



When the user asks the chatbot what is in an image. This class is used to make a prediction.

The witcher wiki parser class is then used to fetch the summary for that creature from the Witcher wiki



## Image Generation

The code for training the GAN is in Train\_GAN.py

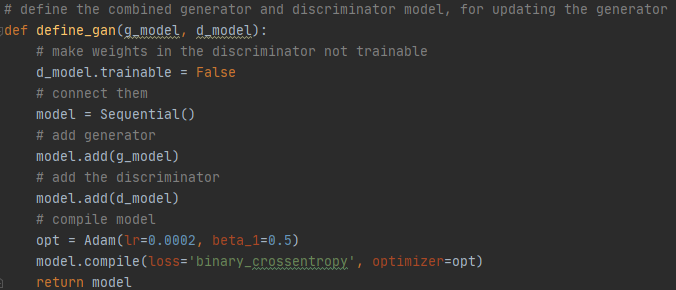
First the discriminator is built



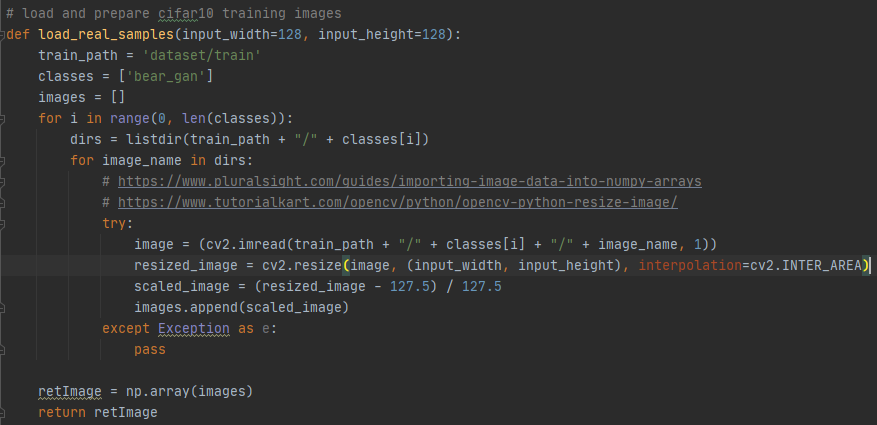
Then the generator is built:



The discriminator and generator are combined into a new model to form the GAN. The discriminator is set to not be trainable within the GAN. This is so that only the generators weights are altered when training the GAN. If the discrimination was trainable within the GAN, it would hurt its performance because it would effectively be trained to recognise the generated images as being real.

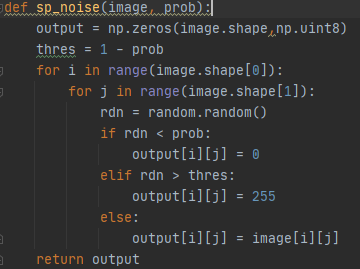


The dataset of real examples is loaded in



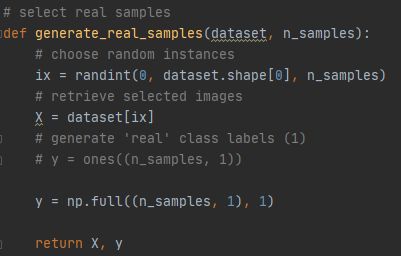
In order to compensate for a relatively small dataset, I also loaded my real samples in a second time, but added random noise to the images.

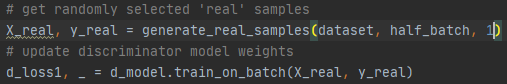




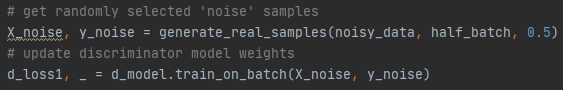
The three models are then trained iteratively.

Random samples from the real data set are collected to be used to train the discriminator

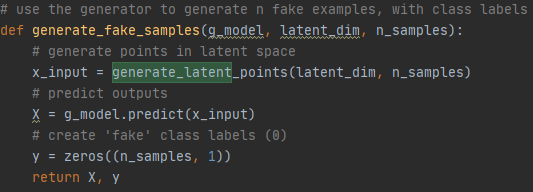




The discriminator is then trained on the real samples with noise. These images are labelled halfway between fake and real to avoid noise showing up in the generated images.

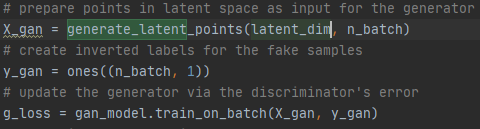


Fake samples are generated by the generator and used to train the discriminator



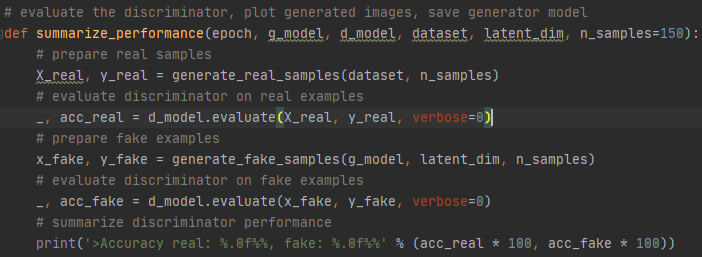


The generator is then trained on how to fool the discriminator.

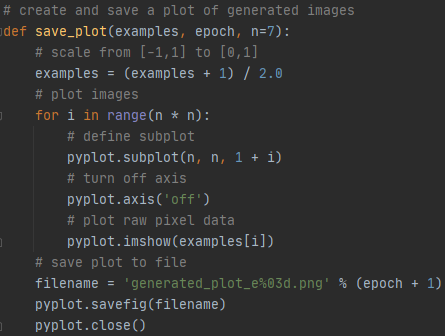


Every 10 epochs, the performance of the model is evaluated.

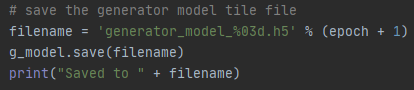
The Discriminators accuracy for detecting real samples, and detecting fake samples generated by the Generator is printed to the console



A grid of generated images is created and saved so the performance of the Generator can be reviewed



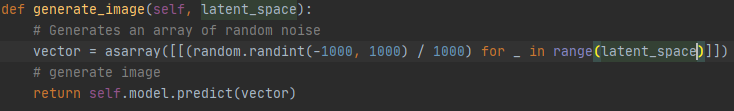
The model is saved with a name containing the current epoch



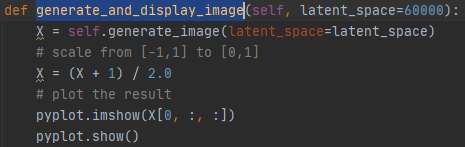
The file Trained\_GAN\_Wrapper.py contains a class which uses the Generator model to generate images. The constructor loads the model.



The function generate\_image creates an array of random noise, inputs that array into the Generator and returns a prediction



The generate\_and\_display\_image takes the Generators prediction and uses the pylot library to display the image

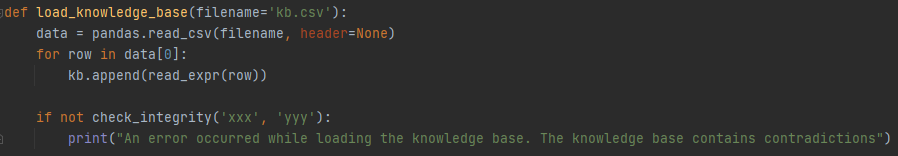


This class is used in the chatbot to Generate an image when the user requests it.



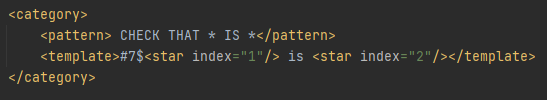
## Inference engine

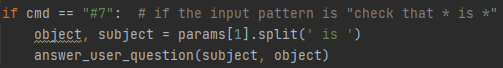
The knowledgebase is loaded and checked for any contradictions. To check for contradictions, the inference engine attempts to infer a fact that the knowledge base should have no decisive answer to. If it returns a decisive answer (correct or incorrect) then it displays an error message informing the user that there is contradictions in the knowledge base



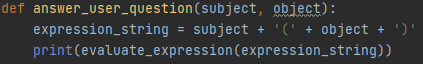


The user can query the knowledgebase by saying something like “check that x is y”

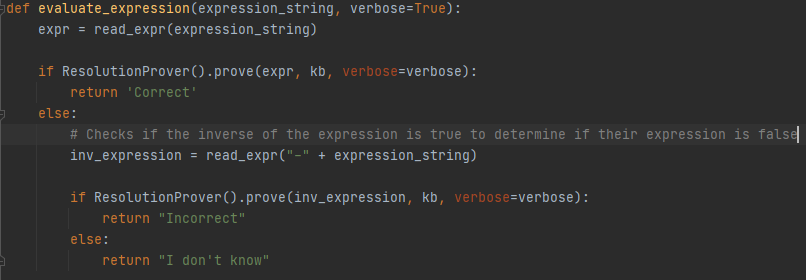




The expression string is constructed



The nltk resolution prover is used to query the knowledge base. To check if the statement is false, the inference engine checks if the inverse of the statement is true.



The user can expand the knowledge base by saying something like “I know that x is y”

