A QA System for learning Python

Marcos Oliveira Ramos
Dpt. Informática, Centro Algoritmi
Universidade do Minho
Braga, Portugal
pg28503@alunos.uminho.pt

Maria João Varanda Pereira
Dpt. Informática e Comunicações, IPB
Centro Algoritmi, Universidade do Minho
Bragança, Portugal
mjoao@ipb.pt

Pedro Rangel Henriques
Dpt. Informática, Centro Algoritmi
Universidade do Minho
Braga, Portugal
pedrorangelhenriques@gmail.com

***Abstract*—This article proposes a Question Answering System that can automatically answer to questions presented in a natural language about the Python programming language. A system of this kind aims at the interaction with a human. Since it is natural for a human to communicate in a natural language, such as Portuguese or English, there is a need for systems that can respond to the user in the same language. When restricted to a closed or specific knowledge domain, these systems can offer satisfiable answers to the posed questions. So, it is expected that the proposed QA System can present reasonable answers to questions about Python. After surveying this emergent working area, that is growing every day, we will present the design and implementation of a Python QA system in order to prove that it is possible to adopt a systematic approach to construct this kind of systems.**

I. Introduction

With increasingly demanding users, look for answers on the WWW using standard search engines is no longer desirable having become a complicated and inefficient task.
The main goal of Question Answering (QA) Systems is to provide a new way of searching information based on natural language questions. At first, this can turn easier the user interaction and open the possibility to be used by people in general. At second, the QA System gives a concrete short answer to the user rather than a list of possible related documents where the desired answer is mixed with other kind of informations as happens in a common search engine.
A QA System can be open or closed domain but in a closed knowledge domain, it can be tuned to give more accurate answers. The system should be prepared with a set of databases resources and with a mechanism to analyse the input question. Techniques of information extraction are then used to construct the answer. Each question should be analysed based on its syntatic structure and on a set of keywords and it should be translated into a repository query. Complex data structures should be prepared to receive the query and avoid redundant, incomplete or wrong answers. At the end the given concrete answer can be complemented with a set of related documents.
PythonQAS, the QA System presented in this paper, is restricted to a closed knowledge domain: Python Programming Language. It is expected to be useful to students or to professionals that want to know more about this language. Moreover Python doesn't belong to the Informatics Engineering Degree Curricula and all the material and tools will be very useful people who wants to learn this language.

A survey of methods and tools to build Question & Answering is presented in Section 2. Our proposal to construct a closed domain QA System for Python programming language is sketched in Section 3; a block diagram to depict the system's architecture is shown, and each component is described in detail. Section 4 is dedicated to discuss the implementation based on a first glance at the questions of the FAQ for Python. In section 5 the new web interface for Pyhton QA system will be presented. Section 6 closes the paper with some conclusions and directions for future work.

II. Related work

QA Systems are mostly separated by **closed** and **open domain**. Closed-domain systems answer questions within a specific knowledge domain or to only a certain type of questions. They target precision, rather than coverage. Dealing with a restricted scope of knowledge allows QA systems to resort to smaller amounts of information, usually structured data such as ontologies. With such limited, formalized and structured data, systems try to take great advantages of natural language processing techniques in order to be the most accurate possible in their answers. Open-domain systems answer questions about almost everything. These system rely on much larger amounts of data than closed-domain QA systems, using mostly unstructured data and general ontologies. Their primarily goal is to provide factoid answers to questions about world knowledge. While closed-domain systems aim at accuracy, open-domain systems intend to cover the greater scope of information that is possible. The ambition is to offer more than a conventional web search engine by answering the user's questions, rather than presenting a simple list of documents/web pages that match the search's query.
Concerning techniques and approaches involved, QA systems will be classified according to three different perspectives: techniques for question analysis; techniques for retrieving answers from knowledge repositories; and techniques for composing the final answer.
For question analysis, some QA systems are based on methods of natural language processing, i.e. methods that try to derive meaning from natural language input. Natural Language Processing is itself a very prominent field of computer science and artificial intelligence that involves techniques like parsing and machine learning. The process consists in converting the user's question into a database query written in a formal

language such as SQL and SPARQL. The output of the query will usually be given as an answer. BASEBALL [4] and LUNAR are closed domain QA systems and they fit perfectly into this category. START[1] is another example of a linguistic approach but it is an open domain.

Another technique for question analysis is Pattern Matching anf Tagging. The QA system analyses a question and labels it in order to find a pattern. If the pattern corresponds to the expected pattern for a certain answer, then this answer should be the right one. For example, when posed with the question, **"Who is the President of Portugal?"** the system interprets the question as **"<Person Name>is the <President of Portugal>?"** and expects the answer to be the name of a person who is president of Portugal. QACID [3] an ontology-based QA system, applies tagging algorithms in order to extract a query pattern, i.e. a query in natural language labeled with morphological information and ontological concepts. To overcome tagging difficulties, due to the complexity of natural languages and human error, QA systems might rely on, for example, synonyms and algorithms for removing stem words, stop words and vowels. The AQUA system [11], among its various steps for processing a question, divides the sentence into subject, verb, propositional phrases, adjectives and objectives. Similar to QACID, it produces a semantic representation of the query that is used by search algorithms when trying to find an answer in the knowledge base. [10] proposes a method that combines patterns with machine learning techniques in an open domain QA system. They use the machine learning technique of bootstrapping to build a tagged corpus from some examples of hand crafted question-answer pairs. These examples are passed to a search engine and from the results of the search, the system extracts patterns. The precision of each pattern is calculated for each type of question. The patterns are then employed in finding answers for new questions.

Concerning Answer Retrieval, some systems rely on structured knowledge sources and ontologies about a specific domain and they are mostly closed-domain QA systems. The idea is to make queries over a database that contains structured information about the system domain. Meaning that the information was produced before the questions were asked and that, if the knowledge source works as it should, the effort lays mainly in understanding the question rather than finding the best answer. This is only feasible when the scope of the domain is well defined and restricted, and the knowledge source is relatively small, very well defined and structured. QACID [3] is a good example of this. In this case the information, from which the system derives answers to the user's questions, is stored in textual documents written in natural language. Ontologies are mostly used to define a language in which documents and questions can be represented and exploited [9]. The most interesting feature of these systems, and what makes them perfect for working with an open-domain, is the capacity of taking advantage of the ever increasing amount of textual information, available throughout the Internet. Web search engines like Google can be used to find and retrieve these knowledge sources from the Internet. Nonetheless, not all QA systems rely on this technique, since most of the time, there is no guarantee of the correctness of the retrieved information. The system described in [2] and MULDER [7] are examples of this.

Concerning Answer formulation, a QA system can give as result a set of fragments of texts or text highlighting or just a succinct answer. Some QA systems are text based, meaning that the answer to the question posed by the user, will be a fragment of a text. By using this approach, a system becomes intricately related to other information access techniques such as "document retrieval", in which entire documents are retrieved, and "passage retrieval" in which chunks of text are returned [8] as answers. Even though, this does not offer much more than a web search engine such as Google, Yahoo, etc. it might be enough to answer a question, since it relies on the intelligence of the user to extract the exact meaning from the text. In [1], PiQASso(Pisa Question Answering System) is an example of this and in [5] is explained how QuALIM [6] answers can be supplemented with paragraphs from the Wikipedia. Contrary to the last approach, some other systems give a succinct answer that directly satisfies the user information needs. Besides giving a direct answer in a natural language, some of these systems will also provide additional informations related to the topic. Evi[2], formerly known as True Knowledge is an open-domain QA platform that translates the user's questions into a language independent query that is executed using the knowledge base and an inference system. The result of the question is a direct answer to the user's question.

III. Python QA System: our proposed solution

A closed-domain system is more adequate when wanting to built something different from what is available today. Since it deals with a smaller scope of information, it is possible to analyze the problem in greater detail, to create a system that could solve a real problem of a certain domain. Nonetheless, most QA system do not necessarily intend to replace the conventional search engines that we have become familiar with, but to complement them or to present a viable alternative.

A. Chosen domain: Python

Python has been capturing attention in recent years. Its popularity among both beginners and experienced programmers is rapidly increasing. Many people learn Python by themselves, resorting on large amounts of scattered information available throughout the Internet. The QA system being proposed could help programmers by reducing the effort necessary to find useful data among all this information.

Python is a powerful and very popular language. Nonetheless, other languages such as Java, C++ or Ruby have some or all of the same functionalities and could as well serve as the domain of the proposed system. The system should be capable

[1]http://start.csail.mit.edu

[2]https://www.evi.com

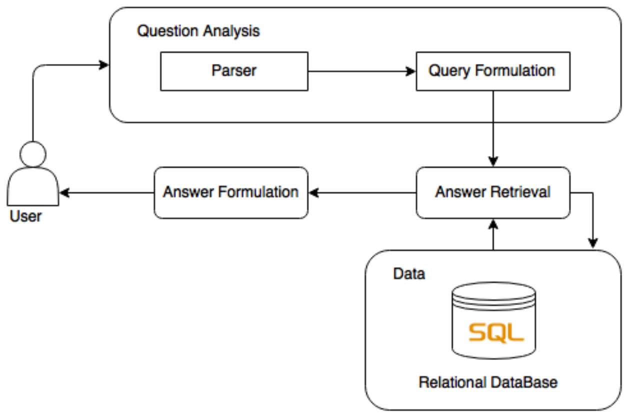

Fig. 1. Proposed System's Architecture

of being used in the domain of another programming language without the need for structural changes.

On a first phase, the Python Frequently Asked Questions (FAQ)[3], will be used as a knowledge source. Since the goal is to create a system capable of answering question in natural language about Python, the fastest way to create a functional prototype is to populate its knowledge base with answers and questions that the Python Software Foundation classifies as "frequently asked". On a second phase, more data should be added to the database in order to enhance the system capabilities.

B. System's Architecture

Figure 1 depicts the proposed system's architecture. The system starts by accepting a question from the user. Then it parses the question in order to produce a query that will be used to retrieve information from the database. This information will then be analyzed and presented to the user.

Every QA system starts by receiving a user's question. Understanding the meaning of the question is vital for the process of retrieving the correct information.

PythonQAS parses the question in order to identify certain words. Any other words are discarded leaving only the desired words and preserving the order that they had in the user's question. This 'meta-question' is then used to construct a query in order to retrieve data from a relational database. The knowledge data should be stored in a way that offers stability, safety and fast access to data. We decided to adopt a structured knowledge base such as a relational database.

When providing an answer to a user's question, we are usually confronted with a dilemma. Should the answer contain just enough information to answer the question succinctly or should the system present additional information about the context of the answer? We approach the problem by establishing a middle term between two methodologies. If a user's question makes sense within the domain of the system, then a succinct answer should be presented to the user. At the same time extracts of text or web links containing information about the topics of the question are also shown as a complement. When the system can not find a satisfiable answer, a message notifying the user of this fact appears instead of the desired result.

[3]https://docs.python.org/faq/

IV. System Implementation

The proposed system was built as a web application. This section describes the system functionalities, the used tools and the implementation approach to build the application.

A. System's Overview

To start implementing the QA system the Python FAQ (PyFAQ) was used as a source of questions and answers on this domain. That PyFAQ will be the seed to build the basic knowledge repository (KR) but in the near future it can and must be enriched with other sources. On one hand, it will support the processing of simple and literal questions; on the other hand, it will allow to provide direct answers.

First and foremost it is necessary to find a method to analyze questions and subsequently apply this method to build the knowledge base and find the respective answers.

We propose the creation of pairs of question-skeleton mapped to ($\mapsto$) the respective answer.

A question-skeleton is a template, or pattern, that describes the intention of a question. For instance, the question "*How to print out the value of a variable in Python?*" can be abstracted by the following skeleton that defines the question intention: "*HOW(print, [variable])*". So, our KR will store the information as quadruples (or four-tuples), **Question Type** $\rightarrow$ **Action** $\rightarrow$ **Keywords** $\rightarrow$ **Answer**.

Looking at the PyFAQ we concluded that the majority of the questions are posed using a small, limited kind of words, mostly adverbs, that give meaning to the phrase. We can then join questions into groups of Question Types.

In Table I there are some examples of how questions in the PyFAQ could be classified by the proposed Question Types.

TABLE I
Question Types

Question Type	Question
Why	Why are Python strings immutable? Why is there no goto?
How	How are lists implemented? How do I convert a string to a number? How do I modify a string in place?
Where	Where is the math.py source file?
What	What is a method? What is a class?
When	When can I call a method?

Usually, a question fits only one Question Type. The answers can be reused by different (but similar) questions.

Answers will be given a confidence rate. That way even if a question does not satisfy the user's needs, others with a lower confidence rate might solve the problem.

To illustrate the idea, observe the following entry in the PyFAQ:

Q3.23: Why is there no goto?

(A3.23:) *You can use exceptions to provide a "structured goto" that even works across function calls. Many feel that exceptions can conveniently emulate all reasonable uses of the "go" or "goto" constructs of C, Fortran, and other languages.*

This answer could also satisfy the question "*How can I use a goto statement?*". Even though the two questions are different, they could be both mapped to the same answer. But, for example the question "*What is a goto statement?*" can not be clarified by the answer. The Question Type is important to identify possible answers, but this does not imply that a question of some type can not be also responded by the answer to a question of a different type. It just means that the confidence of the system in the assertiveness of the answer is not as great as it would be in a different situation.

As said above, each entry in the KR(a pair <question-skeleton $\mapsto$ answer> stored in our central relational database contains the Question Type and also an Action (verb) and a set of Keywords (mostly nouns) that abstract the question intention. This triple (Type, Action, Keywords) composes the so called question-skeleton.
Table II illustrates this principle with examples taken again from PyFAQ.

TABLE II
KeyWords

Question	**Action**	**Keywords**
Why is there no goto?	not(exist)	goto
How are lists implemented?	implement	list
How do I convert a string to a number?	convert	string + number
How do I modify a string in place?	modify	string + place
What is a method?	define	method
What is a class?	define	class
When can I call a method?	call	method
Is it possible to allocate memory for strings in creation time?	allocate	memory + string + creation time

Notice that the Action and Keywords that represent a question do not need to be precisely the same words that are found in the question sentence. We resort to an English dictionary or thesaurus to lemmatize verbs and find synonyms.

This approach introduces generality in the knowledge base and it is a way to be flexible in the question interpretation. For instance, the question "*Why does not exist goto in Python ?*" will be replied with the same answer (**A3.23**) because the sentence will be recognized and represented by the same skeleton of (**Q3.23**).

To sum up, the (**Q&A3.23**) shown above is represented in our KR (contained in the central database) as a four-tuple[4] illustrated by the entry in Table III.

If the user types the question "*Why does not exist a jump statement in Python?*", the system processes the sentence with NLP tools complemented by a NL Thesaurus and a PL Ontology (to convert nouns to the singular form, verbs to infinitive form, and find synonyms, etc.), in order to identify the Question Type, Action and the Keywords that characterize

[4]Actually three elements that describe the question-skeleton plus the answer.

TABLE III
Representation of an entry in the database

Q Type	**Action**	**Kw**	**Answer**
Why	not(exist)	goto	You can use exceptions to provide a "structured goto" that even works across function calls. Many feel that exceptions can conveniently emulate all reasonable uses of the "go" or "goto" constructs of C, Fortran, and other languages.

the question skeleton. By extracting that information, our system will generate a query to send to the database aiming at retrieve the associated answer.

To be more precise, after analyzing the input sentence our system tries to find an entry with the same Question Type (*Why* in this case), and with at least one Keyword and Action. If our database contained only the entries with the set of keywords shown in Table II, the system would not find any match. But the system resorts to synonyms and similar concepts, thus it would not only look for the keyword *jump statement* but also for *goto*. A simple scoring function is then used to rank the candidate answer(s) based on the frequency of the Question Type, Action and Keywords. The answer with the highest score is then presented to the user followed by the ones above a certain score. If there is no satisfiable answer, the system returns a simple message notifying the user of this fact and advises him to rewrite the question.

B. Technology used

Since the domain of the system is a programming language (Python), we decided to use the same language for the development of the application. Python is a very popular language among the natural language processing community, thus, there are many free and good tools, packages and libraries for the language, such as NLTK.

Django was chosen as a framework to construct the web site. Using a framework such as Django allow us to construct a web site without needing to build everything from scratch. Django is free and open-source, it is written in Python and promises to be fast, secure and scalable. It officially supports various relational databases which facilitates the building of data-driven websites and comes with an integrated web server that helps to test the application faster and almost effortless. It also offers an administration application and many capabilities that intend to reduce effort and increase safety like forms, and models to build and connect with the database.

PostgreSQL was chosen as the database engine for the project. It is a free and open source object relational database that runs on most modern operative systems and can be both used for small single-machine applications and larger data warehousing.

C. Implementation approach

This section presents all the main steps necessary to build PythonQAS, along with explanations regarding the different methods adopted, the reasons behind all major decisions and the most critical obstacles that were faced.

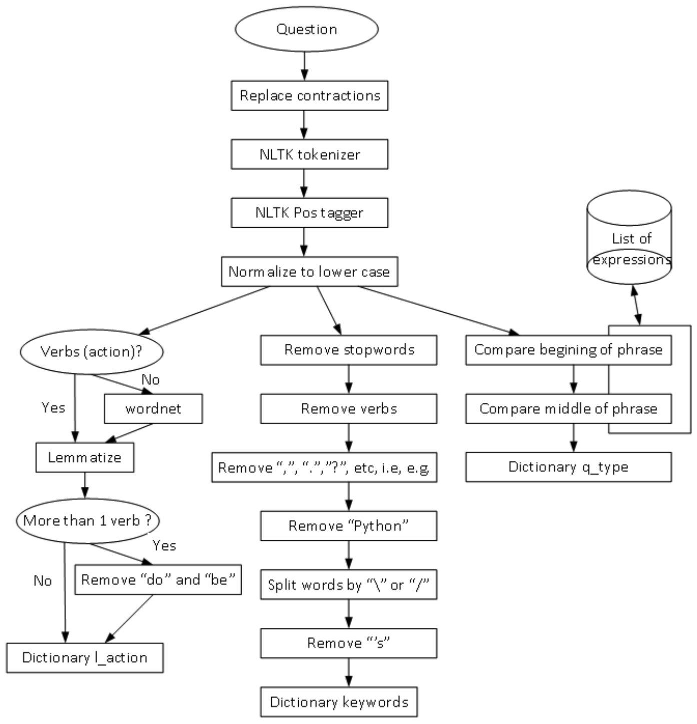

Fig. 2. Phrase Analysis

1) Phrase Analysis: A phrase or a question is always composed of many components such as adverbs, nouns, verbs, etc... To understand the meaning of a phrase it is necessary to split it into these components. Even if not in a conscientious way we, human beings, also do it. Take the example of when we are trying to learn a new language and we do not know all the vocabulary that a native uses in its daily life. Even if we do not know the word we are sometimes able to infer it's meaning by its position on the phrase. To analyze a question, the proposed QA system does a similar job by dividing a phrase into multiple components and trying to identify three different types: action, keywords and a question type.

Concerning the **Action**, usually, while using natural languages, human beings use a word that describes an action, an occurrence or a state of being. This word is obviously called a verb. But finding the right action or the main action of a question is not always an easy task. A question could have more than one verb, verbs have tenses (past, present, future), verbs change according to the personal pronoun and a verb might, for example, be no different from a noun, except for its position on the phrase. Even a human being could be mistaken about the right action of a question if not given more context or information about the it.

Figure 2 depicts the major techniques used by the system for analyzing a phrase.

In the application an Action is an object that starts by receiving a question as a simple string. Before doing anything to the question it is best to find any contractions that may exist in the phrase and if possible convert them to their full form. This may lead to ambiguous situations [12]. After some tests with different contractions it was concluded that the advantage of trying to disambiguate would not offer any significant advantages if taken to account the work that would be needed. Thus, the PythonQAS only converts contractions that are not ambiguous.

Next step, the phrase is divided into multiple substrings using the NLTK Tokenizer Package, in order to be used by the Pos (Part-of-speech) Tagger from the NLTK Tagging Package. Right after using the Pos Tagger, the words are converted to their lower form to avoid problems when comparing words. This could not be done before because it could decrease the efficacy of the Post Tagger. For example, the word "I" in a phrase has a complete different meaning from the letter "i".

The process continues by identifying all the verbs in the question. These will be saved to a dictionary where the key is a word (verb) and the object is a number from 0 to 1. This number represents the trust in the assertiveness of the verb being the right action. It will be very useful to find an answer in the database and distinguish the different results of the database queries. This dictionary will be called *l_action* along this document. If one or more verbs were found, these will be converted to their infinitive mode using the NLTK WordNetLemmatizer.

If at the beginning, no verbs were found, then a different path should be followed in order to find at least one verb. The process works by looking at the words of the phrase and trying to find any that could be a verb. But, since a word could be used both as a verb or an adjective, this process is not very reliable. So, the Trust of the verbs will be reduced.

Next, the Stopwords are removed. Stopwords are usually the most common used words in a certain language and do not contain any significant importance to the meaning of a phrase. Thus, its removal is very common when processing natural languages and usually do not compromise the success of the analysis.

The next step is to find which words could be verbs by using WordNet, a lexical database. It groups words into synonyms called synsets and provides many data about these words like number of relations, definitions and examples of usage. It might be seen as a mixture of a thesaurus and a dictionary. Next, PythonQAS resorts once again to the dictionary containing the results of the Pos Tagging process to verify if the remaining words are nouns or adjectives. If so, and if the words are positively identified by WordNet as verbs, the words are added to dictionary "l_action".

If more than one verb was found, regardless of the way the verbs were obtained, the tool will try to exclude false positive findings. First the PythonQAS removes all stopwords and words that contain characters which are not letters.

Concerning the **Keywords**, most phrases in a natural language contain a set of keywords that give the phrase meaning and context. These words are not the most frequent words used in a speech, but it would be impossible to express anything more than basic replies without them. For example, the question ”How can I populate a dictionary?” has one keyword “dictionary”. The question ”How can I populate a dictionary with lists?” has a set of two keywords, “dictionary” and “lists”. The first question relates to the context “dictionary” and the second to a context contained in the first one. Thus, the more keywords, the smaller the domain of the phrase will be.

In the Action process we knew that actions were verbs, but keywords are not as restrictive as that. So, the strategy here passes more by excluding unwanted information(keywords). PythonQAS starts by removing all stopwords and words that were identified by the Pos Tagging process as being verbs.

Next, the keywords are saved to a dictionary (“keywords”), similar to the “l_action”. The keywords are the keys of the dictionary and their objects are numbers between 0 and 1, representing the trust in the assertiveness of the classification.

It was observed in some questions that similar concepts/keywords were separated by a slash(“\” or “/”). Neither the NLTK Tokenizer or the Pos Tagger were able to separate them and they were considered to be a single word. To solve the problem, PythonQAS tries to identify any slashes or other separators inside every keywords. After the system splits the keywords using a list of separators, if the individual words contain more than one alpha character, the old word is erased from the dictionary and the new ones are added.

Concerning the **Question Type**, it was noticed that the questions of the Python FAQ are almost always formulated in similar ways. By grouping the questions by their similarity it was possible to find a set of question types. Question types are expressions such as “How”, “When”, “Where”, etc.. These types are not directly related to the keywords or even the action of the question, but its use could change the entire meaning of a question. For example the questions “How should I use a dictionary?” and “Where should I use a dictionary?” are almost the same except for the use of “How” or “Where”. The answers to both questions are also similar and we could actually find the answer to one of them in an indirect way in the answer of the other. Nonetheless, PythonQAS should be the most accurate possible and for that it tries to distinguish from the different question types.

A Question Type is, like the Keywords and Action, an object. The initial part of the procedure is shared with the Keywords and the Action classes.

After rhis, the system uses a list of lists of expressions to try to find the question type of the phrase. The list does not contain words but lists of words. Each list contains expressions that are similar. That way when searching for an answer, not only the question type of the users question can be used but also the similar expressions in the list.

The most important information produced by the object’s different parts is contained within a dictionary, called “q_type”, similar to the ones used for the Action and Keywords, whose keys are strings (question types) and the objects are numbers between 0 and 1.

To identify the expressions in the phrase the system starts by trying to match the different question types with the beginning of the question. The question type is usually at the beginning of the phrase.

The next step is to find out if the question was formulated using two phrases. The system splits the text into multiple phrases and tries to identify question types at the beginning of each phrase and then tries to identify it in the middle.

If no question types were found, the system adopts a new way to find a satisfiable question type. When comparing the expressions, or words that constitute the question type of the phrase, with the results of the NLTK PosTagger a relation was discovered between the tags and the question types. The question type, or part of it, is usually tagged with one of the following tags: “WRB”, “WP”, “MD”, “WDT” or “EX”. But even though they are usually tagged with those tags, not all words tagged in the same way constitute question types. For example, Wh-adverbs (WRB) are adverbs that start by “wh”, sometimes called interrogative words. “how” or “however” are such examples. Nonetheless “how” is a very common question type contrary to “however”.

2) Information Storage: The Python FAQ is used as the initial knowledge base of the PythonQAS. The initial database is very important because it allow us to test the system and adjust it according to the results. All the different web pages containing the questions/answers were downloaded comprising not only the text, but also their structure and HTML annotations. That way, things like colored text and web links can keep being part of the answer.

The Django Framework offers many capabilities, such as Models, that try to decrease the complexity and work associated with the building of a web application. Models allow us to simply create python classes that contain and represent information. These classes are used by Django to create all the database structure. This not only saves time by building the database, but also, because all objects in the database are treated as python classes in the application, it allows the programmer to save, delete, update and retrieve data from the database without the need to ever use SQL code. The Answer Model contains five variables: question, question type, action, a set of keywords and an answer.

3) Information Retrieval: After receiving a question from the user, the system creates an object(AnswersRetrieval) that aggregates everything necessary to process the question and search for answers. When the object is created it starts by defining variables used to store relevant information about the execution of the different methods contained by the object. These variables indicate different weights and measures to calculate the probability of an answer being right, the NLTK Stemmer that is going to be used, answers found in the database, results of analyzing the question, etc.. It also creates three objects, explained before: Question Type, Action and Keywords.

The next step will be to call three auxiliary functions to retrieve answers from the database. Each of these correspond to the components extracted from the phrase. The different methods used to retrieve the answers and the analysis of the question are used to calculate a probability value.

If the system found any actions in the phrase, then for each action it will try to find all Answer objects with that action. First, a variable is created holding the Trust in the action that is being analyzed. This value is provided by the Action object. Second, the Answer objects that contain this action are obtained from the database using a direct match.

Then the word (action) is saved to a dictionary containing pairs as objects. This dictionary, called ‘visited”, saves all the words that were already used in the search process to avoid duplicated results. The keys are the words being used in the search. The first element of the pair is the a string ”word” and the second is the actual word used to search the Answers.

Last, a dictionary “action_answ” is built containing all the answers. The key of the dictionary is a tuple where the first element is the Answer object and the second is the action that was used in the search process. The object appointed by the key is a value between 0 and 1 containing the trust in the assertiveness of the answer.

In the next step, instead of using the action to search for Answer objects, the system uses a stem word of that action using one of the Stemmers provided by the NLTK. After obtaining the stem word, the system searches for Answers with an action that starts with it. Even though a stem word does not always correspond to the beginning of the word, in most cases it does. The best option would be to apply the Stemmer to the action contained by each individual Answer, but by doing so the system would have to read and compare each Answer using only Python (we do execute any SQL code but of course Django still does). In a very small database this would not be a problem but in a medium sized to large database the time and resources necessary to do this would be very unsatisfactory. Then, the action is saved to the “visited” dictionary. Now, the key used in the dictionary is the stem word and not the action as before. This way the system saves what it really used in the search without discarding the action. This is necessary because two different words might have the same stem word and saving the action instead of the stem word could cause duplicate values. Lastly, the results obtained by search are saved to the “action_answ” dictionary.

The next step is to use synonyms by resorting to the WordNetLemmatizer. Note that only synonyms that are verbs will be used. The process used to find Answer objects using synonyms is not very different from what it was done before with the original word. Synonyms are received and for each of them a search for Answers will be performed followed by the use of Stem words.

In the next step, the system tries to find Answer objects using the keywords extracted from the phrase. The procedure is similar to the one used for finding answers with an action. First, the system declares a dictionary with the visited elements. For each keyword in the list of keywords contained by the object Keyword, a direct match will be tried .

After the first search there is a slight difference. Since the verbs/action were already in their infinitive form, nothing was done regarding this. Thus it is necessary to lemmatize the keywords using the WordNetLemmatizer. A dictionary “keywords_answ” will contain the Answers and the Trust in these same Answers, similarly to the “action_answ”. Next, the stem word of the current keyword is used followed by the use of synonyms and their stem words.

For any question types found by the questions analysis, the system searches the database for matches. Because of the nature of question types and the methods used to extract them, the search for answers based on the question type is simpler from the Action and Keywords methods. Nonetheless, the idea is still very similar.

A list is created, temporarily storing the expressions that were already visited by the search process. For each question type in the Question Type object, the PythonQAS queries the database in order to get any entries whose question type corresponds to the current expression. Any Answers found will be stored in a dictionary called “type_answ”, where the keys are tuples containing the Answer and the expression used to retrieve that answer, and the objects are values between 0 and 1 expressing the trust in that answer being the right one to the users question.

After creating the three dictionaries “action_answ”, “keywords_answ” and “type_answ” containing all the answer extracted with the actions, keywords and question types, the goal is to merge these into a new dictionary “answ_prob”. The keys of the dictionary are Answers and the objects are dictionaries containing the trust/probability of the assertiveness of the Answer regarding the action, keywords and question type of each Answer.

Regarding the Answers found with the question type, the dictionary “type_answ” will be treated in the same way as the “action_answ”. The same strategy cannot be applied regarding the keywords because an Answer can have more than one keyword. Instead of simply copying the trust, the system increments the value of the keyword of the dictionary with that trust value. This way a previous value is never lost but summed. Now, the system will divide this number by the number of keywords of the Answer Object resulting in a trust value made of the values obtained by using all keywords found in the question.

The main goal of an object of type Answer is to provide answers, thus, after everything that was explained above, the system has the dictionary “answ_prob” containing all possible answers. Still there is not yet a way from where to choose the best possible answer. The system will iterate all the Answers in the dictionary and it will calculate its probability based on the trust value of the Keywords, Action and Question Type of the respective Answer.

After, the system returns a list containing tuples (answer, probability), sorted by the probability value.

A good example would be to ask PythonQAS, ”What are the rules for local and global variables in Python?” and thus obtain a list of possible answers sorted by the probability value. This example can be consulted at http://pythonqas.

epl.di.uminho.pt/qaSystem/answer/?question_text=What+are+the+rules+for+local+and+global+variables+in+Python%3F.

V. Web application

PythonQAS uses a web interface to communicate with both users and system administrators. The website can be consulted at http://pythonqas.epl.di.uminho.pt.

Regarding the normal user's interface, the system provides a way to ask questions, view results and some information regarding the usage of PythonQAS, information about the authors and the developing methods used in its construction. When a user asks a question the system creates an Answer-Retrieval object using the question. If the object produces any answers and if these answers have a probability/trust value greater than a certain value defined in the system's programming, the answers will be presented to the user, sorted by their trust value. Only the most probable answer is shown to the user, the other answers are hidden inside collapsed divs showing only their order on the list and their trust value. Nonetheless, the user only has to click on the div to see its content.

Regarding the administrator's interface, PythonQAS provides, through the Django Administration web application a way to list all answers and keywords in the database, edit, delete, update and insert new entries, and add new users (administrators).

Even though Django offers a very good administration interface, it lacks in some functionality particular to the PythonQAS. Thus, the system was complemented in order to be able to: receive a text file containing pairs Question $\rightarrow$ Answer, parse this file, show the results to the user along with suggestions about the different components needed to insert the information into the database and a way to insert it after it is reviewed; easily insert an individual entry to the database; receive a question and creates three objects of the classes Action, Keywords and Question Type and return the results to the interface to test the system capabilities.

VI. Conclusion

The design and implementation of a computer based system capable of understanding an Human question, about a knowledge domain, delivering an appropriate answer is a major dream that is stimulating, since a long time ago, the artificial intelligence, computer science and linguistic communities to moving them up to a deeper research. In that context of Q&A (question and answering) Systems, this paper contributed, on one hand, with a survey on the work already done aimed at the classification of existing approaches and tools, and, on the other hand, as a proof of concept with the design and implementation of PythonQAS, a web-based system to answer questions set up by programmer about the language Python. Even returning satisfactory answers, the system still lacks the access to other information sources to be able to derive more answers (the more the system grows in terms of information, the better it will be able to provide the accurate answers). Another project contribution, also discussed in the paper, was the choice of Python information sources and the development of a back-end system to collect information from them and automatically populate PythonQAS knowledge repository. A first (and simple) test and evaluation of PythonQAS was performed to draw conclusions about the system outcomes (the lessons learned were pointed out) in order to understand the directions for future work (possible in a PHD context):

1) Increase the number of answers in the database, i.e. more reliable data should be added to the KR;
2) Adjust the different measures used to calculate the probability value of an answer;
3) Integrate the PythonQAS into a Question and Answer Web Site like Stackoverflow;
4) Apply this approach and technology to another Knowledge Domain, for instance Java, Perl or C# programming languages, to test its validity;
5) Apply this approach in a different Natural Language environment, for instance to Portuguese.

Acknowledgment

This work has been supported by COMPETE: POCI-01-0145-FEDER-007043 and FCT – Fundação para a Ciência e Tecnologia within the Project Scope: UID/CEC/00319/2013.

References

[1] Giuseppe Attardi, Antonio Cisternino, Francesco Formica, Maria Simi, Alessandro Tommasi, and Cesare Zavattari. Piqasso: Pisa question answering system. In *TREC*, 2001.

[2] Susan Dumais, Michele Banko, Eric Brill, Jimmy Lin, and Andrew Ng. Web question answering: Is more always better? In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 291–298. ACM, 2002.

[3] Oscar Ferrández, Rubén Izquierdo, Sergio Ferrández, and José Luis Vicedo. Addressing ontology-based question answering with collections of user queries. *Information Processing & Management*, 45(2):175–188, 2009.

[4] Bert F Green Jr, Alice K Wolf, Carol Chomsky, and Kenneth Laughery. Baseball: an automatic question-answerer. In *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*, pages 219–224. ACM, 1961.

[5] Michael Kaisser. The qualim question answering demo: Supplementing answers with paragraphs drawn from wikipedia. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Demo Session*, pages 32–35. Association for Computational Linguistics, 2008.

[6] Michael Kaisser and Tilman Becker. Question answering by searching large corpora with linguistic methods. In *TREC*, 2004.

[7] Cody Kwok, Oren Etzioni, and Daniel S Weld. Scaling question answering to the web. *ACM Transactions on Information Systems (TOIS)*, 19(3):242–262, 2001.

[8] Jimmy Lin, Dennis Quan, Vineet Sinha, Karun Bakshi, David Huynh, Boris Katz, and David R Karger. What makes a good answer? the role of context in question answering. In *Proceedings of the Ninth IFIP TC13 International Conference on Human-Computer Interaction (INTERACT 2003)*, pages 25–32, 2003.

[9] Diego Mollá and José Luis Vicedo. Question answering in restricted domains: An overview. *Computational Linguistics*, 33(1):41–61, 2007.

[10] Deepak Ravichandran and Eduard Hovy. Learning surface text patterns for a question answering system. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 41–47. Association for Computational Linguistics, 2002.

[11] Maria Vargas-Vera and Miltiadis D Lytras. Aqua: A closed-domain question answering system. *Information Systems Management*, 27(3):217–225, 2010.

[12] Martin Volk and Rico Sennrich. Disambiguation of english contractions for machine translation of tv subtitles. 2011.