Introduction

As we said in a recent lecture, two important parts of making an agent that can **tell stories or have conversations** is the **interaction between text in a natural language such as English, and the program's knowledge of the world**. Today, we will work on a simple chatbot agent that has a **store of knowledge**, **uses that knowledge to decide what questions to ask**, and adds to that store of knowledge by interpreting the user's responses.

Getting Started

1. Download *chatbot.py* from the Moodle page. This is the skeleton for a chatbot program. The program has a store (a Python set) of knowledge in the form of "triples" in the form *(person, fact, value)—person* is a unique identifier for that person, the *fact* is a kind of knowledge about them (e.g. name, address, age,…), and the *value* is the value of that fact. So, an example triple is ("person1", "address", "52 Festive Road"). More formally, these three parts are known as the subject, predicate, and object.

   These triples are the starting point for the semantic web and for the knowledge graph representation that is used by Google to add context to search queries (see Manuela Nayantara Jeyaraj, Conceptualizing the Knowledge Graph Construction Pipeline, https://wso2.com/blog/research/conceptualizing-the-knowledge-graph-construction-pipeline for more details; it is not necessary to read this to do the task, perhaps read it later).

2. Run the program. It doesn't do much—it just prints the current state of the knowledge base (to help), divided into "unknown" information (where the object is a ?) and known information (at the beginning, this is the empty set, because everything is unknown). It then asks a question, and uses the *input* function to get a piece of text from the command line.

Tasks

1. **Process some facts**. Firstly, in the *if unknowns…* loop, choose an item from *unknowns* at random. Call it *currentQuery*. Set the question string to ask "What is your…" and then the fact from *currentQuery.* Remove *currentQuery* from the list of knowledge (because we are about to find out what it is). After the line beginning *reply* = *…*, add a new fact to the knowledge where the *person* and *fact* are from *currentQuery*, and the *value* is equal to *reply*.

   So, for example:

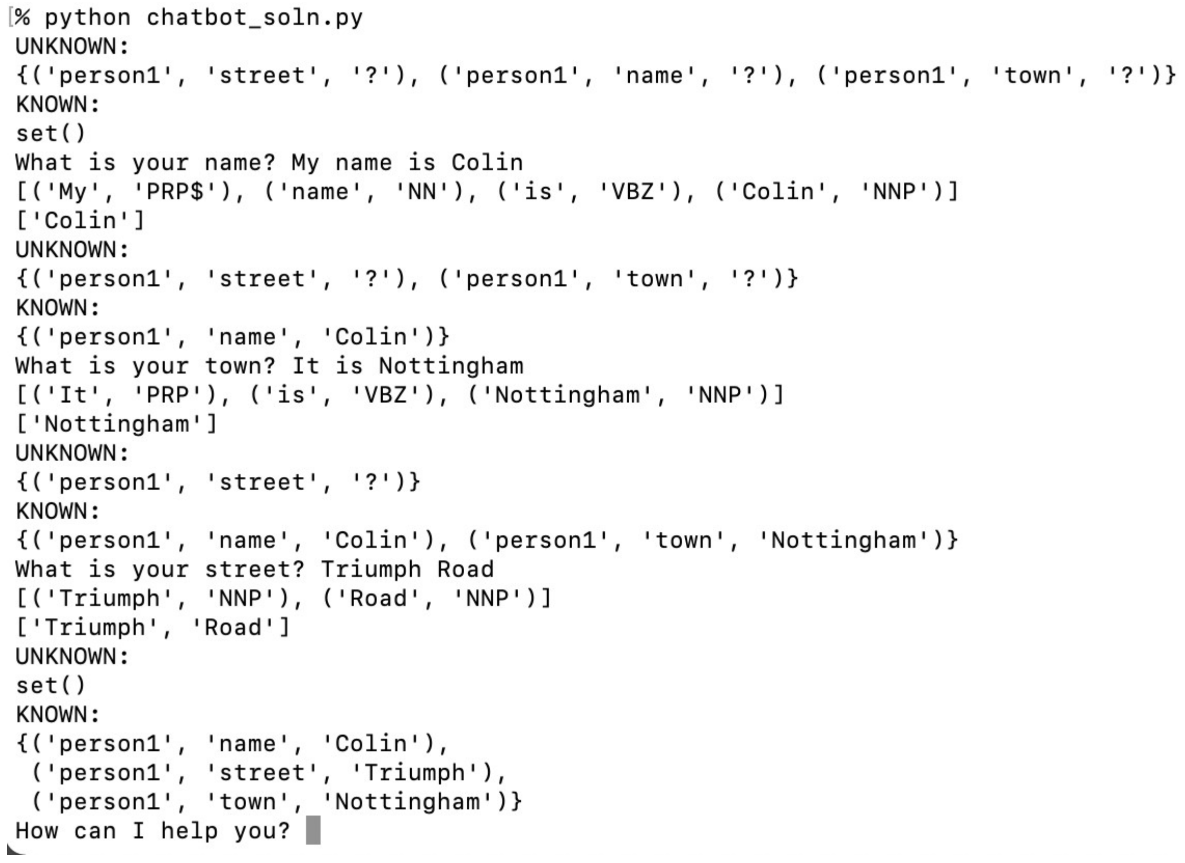
```
The fact chosen is ("person1", "town", "?")
The program asks the question "What is your town?"
The program removes ("person1", "town", "?") from its knowledge
The person replies "Nottingham"
The program adds ("person1", "town", "Nottingham") to its knowledge
```

So now, the program will continue to loop, asking questions, until its knowledge-base is empty. It then asks "Can I help you?" (We haven't written any code to deal with that yet).

2. **The processing of the reply isn't very good yet**. If the person replies with “my town is Nottingham”, then the fact becomes *("person1", "town", "my town is Nottingham").* So, we need to do some more processing on the input. So, let's use *nltk* to process the input grammatically. Import this package into the code.

   Firstly, take the *reply* to the question, and use the *nltk* code from last week's class to tokenise it and tag it. So, if someone's reply to the computer's question is tagged as an “NNP” (i.e. a “proper noun”—something that is the name of a person, place, etc.) then select that as the word to be stored as the object of the triple

   So, this should now be flexible as to how the reply is constructed; note that I have printed out the tagged sentence, and the list of proper nouns, after the knowledge:



```
% python chatbot_soln.py
UNKNOWN:
{('person1', 'street', '?'), ('person1', 'name', '?'), ('person1', 'town', '?')}
KNOWN:
set()
What is your name? My name is Colin
[('My', 'PRP$'), ('name', 'NN'), ('is', 'VBZ'), ('Colin', 'NNP')]
['Colin']
UNKNOWN:
{('person1', 'street', '?'), ('person1', 'town', '?')}
KNOWN:
{('person1', 'name', 'Colin')}
What is your town? It is Nottingham
[('It', 'PRP'), ('is', 'VBZ'), ('Nottingham', 'NNP')]
['Nottingham']
UNKNOWN:
{('person1', 'street', '?')}
KNOWN:
{('person1', 'name', 'Colin'), ('person1', 'town', 'Nottingham')}
What is your street? Triumph Road
[('Triumph', 'NNP'), ('Road', 'NNP')]
['Triumph', 'Road']
UNKNOWN:
set()
KNOWN:
{('person1', 'name', 'Colin'),
 ('person1', 'street', 'Triumph'),
 ('person1', 'town', 'Nottingham')}
How can I help you?
```

Note that this fails on the last example, because it stores the address as “Triumph” not “Triumph Road”. We will return to this, and to some similar problems, in task 5.

3. **Now, let's turn our attention to the second half of the input**; the part after *helpRequest* = we will add some code to process new requests. Again, tokenise and tag the input, and look for the verbs (words whose tags begin with “VB”; use *startswith* for this). Search that list for verbs that are to do with buying things (“buy”, “order”, “purchase”). If you don't see one of these, prompt the user (“I cannot help you with that. Can I help you to buy something?”). You can do the check by using the *any* function in python:

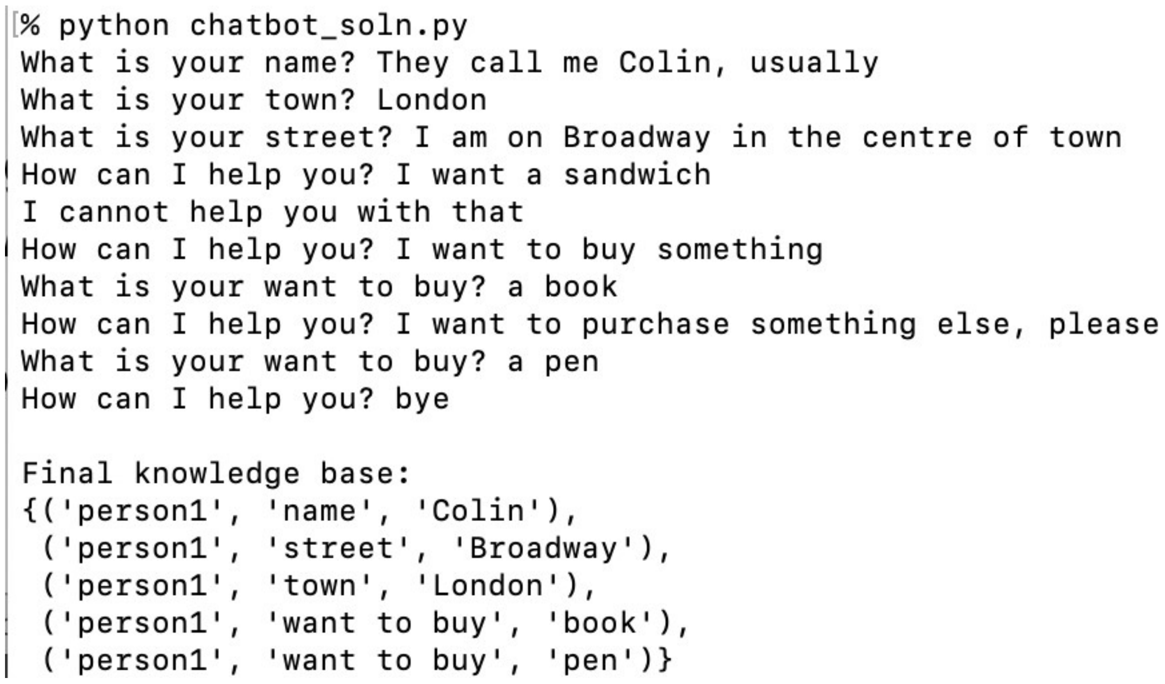
```
    if any(item in [“buy”, “order”, “purchase”] for item in verbs):
          …
    else:
          …
```

If you detect one of these “buy” words, add an additional unknown to the knowledge list, in the form of (“person1”, “want to buy”, “?”). Then, there will be some knowledge that is unknown when the code goes around the loop again, so it will ask for details.

<u>Optional:</u> You could make this more sophisticated. If someone wants to buy a book, add the title and author to the unknown knowledge. If they want to buy a pen, add the colour.

4. Add code to detect words such as “end”, “goodbye”, etc. When the user types one of these, write code that goes through the knowledge-base and makes a final list of their purchases. Then set *active* to *False* so that the program ends.

By this point, you should be able to have a conversation like this



```
% python chatbot_soln.py
What is your name? They call me Colin, usually
What is your town? London
What is your street? I am on Broadway in the centre of town
How can I help you? I want a sandwich
I cannot help you with that
How can I help you? I want to buy something
What is your want to buy? a book
How can I help you? I want to purchase something else, please
What is your want to buy? a pen
How can I help you? bye

Final knowledge base:
{('person1', 'name', 'Colin'),
 ('person1', 'street', 'Broadway'),
 ('person1', 'town', 'London'),
 ('person1', 'want to buy', 'book'),
 ('person1', 'want to buy', 'pen')}
```

(the grammar is still a bit shonky in places; think about how you might correct this, without writing too many special-purpose exceptions)

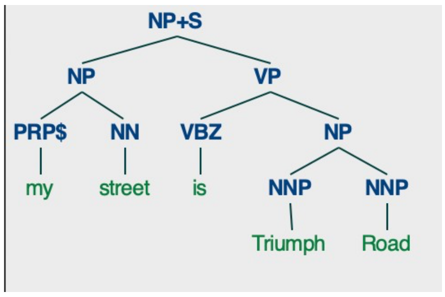
5. **(somewhat more complex)**. Install the *pyStatParser* package using

```
$~ pip install pyStatParser
```

or whatever mechanism you use in your IDE for installing packages; or, if that doesn’t work

```
$~ git clone https://github.com/emilmont/pyStatParser
$~ cd pyStatParser
$~ pip install --editable .
$~ pip install six
```

This enables you to do a more complex grammatical analysis, by taking a sentence and turning it into a hierarchical structure called a *parse tree*. Download *parserDemo.py* from the Moodle page, and run it (the first time you run it, it will download some additional data; after this it will run faster). This takes the sentence “My street is Triumph Road” and produces the following tree:



This helps us to see that “Triumph Road” is a single grammatical unit—basically, if we have asked the question “What is your street?” we can look for anything that is a “VP” (verb phrase) that has “is” someone in it, and treat the other side of that sub-tree as the answer. One approach to this is as follows.

1. Take the tree *t* and create a list by using the statement:

```
st = list(t.subtrees())
```

one of these (perhaps more than one if someone writes a particularly complex sentence) will be labelled with “VP” (you can get the label by doing *st.label()*.

2. Take this subtree (call it *vpTree*), and again extract the subtrees

```
vpSubtrees =list(vpTree.subtrees())
```

then, search for the subtree whose label begins with “is” and the label whose subtree begins with “NP” (call this one npSubtree).

3. From the latter, extract everything by using

```
" ".join(npTree.leaves())
```

to form a single string. Details of the tree syntax can be found at https://www.nltk.org/api/nltk.html#module-nltk.tree

Use this to extract better information from the user input, e.g. recognising multi-part names and addresses.

Extensions

1. Deal with multiple people. When someone logs on, you will have to identify them by asking a number of questions. Perhaps someone knows their unique identifier, in which case you can set it from just one question. But, perhaps they don’t, in which case you might have to ask questions until you narrow it down (“what’s your name?”, “Colin”; program knows facts about multiple people called “Colin”, so it still needs to ask other questions until one person in the knowledge is identified uniquely).
2. Rewrite the chatbot so that it has a different kind of conversation (a “small-talk” social chat, a quest setup in a role-playing game, a bank processing transactions…). More interestingly, rewrite the code so that the initial facts, and the follow-ups to “Can I help you?” are stored in a data file, so that the same core chatbot code can be used for multiple situations.
3. Add in code to prompt the user if they haven’t said anything after a certain number of seconds.
4. (complicated). Re-implement the “knowledge” as a database, either a traditional SQL database or something like *RDFlib* https://rdflib.readthedocs.io/en/stable/
5. (substantial but interesting) Re-work the code so that it is a game dialogue about detectives asking queries from crime suspects. Now, as well as storing and querying

direct knowledge (“What is your name?”), the code will have to deal with people’s beliefs (“doxastic knowledge”). So, triples in *knowledge* can be things like:

(“Prof. Plum”, “knows”, (“Rev. Green”, “murdered”, “Dr. Black”))

and similarly, the program will be able to answer questions about characters beliefs.