1. Question recall and the solution

Entity Extraction Exercise

1. Divide sentence into tokens, and then attach the tag for each token.
2. Use ne_chunk provided by NLTK. ne_chunk needs part-of-speech annotations to add NE labels to the sentence. The output of the ne_chunk is a $nltk.Tree$ object. If type of the chunk is tree, it is the NNP.

By the steps, we can finish the exercise:
Exercise1: Extract all named entities as well as its type/label and store in a directory. The key to solve this question is use the chunk $.label()$ method to get the type of entities. And use the commend $' '.join()$ to combine two strings into one for two consecutive nouns or adjectives plus nouns should be placed in one list, not separate.

Exercise2: Extract only PERSON entities or specify the type of the entities. It just add a judgment condition to the first one: $i.label() == 'PERSON'$, which can extract it.

Exercise3: Define your own grammar for noun phrase chunking with nltk.RegexpParser.

```
def np_chunking(sentence):
    entity = []
    grammer = "NP: {<JJ>*<NN.*>+} \n {<NN.*>+}"  # chunker rule(s),
```

we should define the grammar of the sentence and use $cp.parse$ to apply it to the chunks. So it so similar with the $ne.chunk()$.

Hypernym Extraction Exercise

The exercise can be divided to four steps:

1. Noun phrase chunking or named entity chunking. You can use any np chunking/named entity technique. We should also define the grammar by self with nltk.RegexpParser.

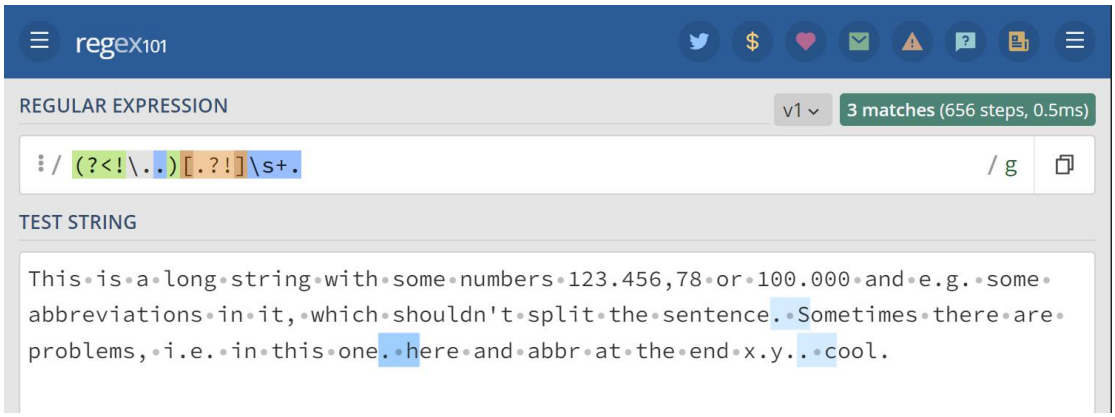
2. Chunked sentences prepare. Traverse the chunked result, if the label is NP, then merge all the words in this chunk and add a prefix NP_ (for subsequence process). And all the tokens are separated with a white space (“ ”)

3. Chunking refinement. If two or more NPs next to each other should be merged into a single NP.

a. We should find the hypernym and hyponyms on processed chunked results. And the difficult thing is writing the Hearst Patterns.

```
# Merge everything to get the final extractor
class HearstPatterns(object):
    def find_hyponyms(self, sentence):
        hearst_patterns = [("(NP_\w+ (, )?such as (NP_\w+ ?(, )?(and |or )?)+)", "first"),
                           ("((NP_\w+ ?(, )?)+(and |or )?other NP_\w+)","last"),
                           ("(NP_\w+ (, )?as (NP_\w+ ?(, )?(and |or )?)+)", "first"),
                           ("(NP_\w+ (, )?including (NP_\w+ ?(, )?(and |or )?)+)", "first"),
                           ("(NP_\w+ (, )?especially (NP_\w+ ?(, )?(and |or )?)+)", "first")]
```

the details of regular expressions were very demanding. Because if you make a mistake, it will not match the text and the result will be empty. Though lots of debugging, I found that it's just an extra **space** effect the result. We can test the result by the website: https://regex101.com/.



OpenIE Exercise

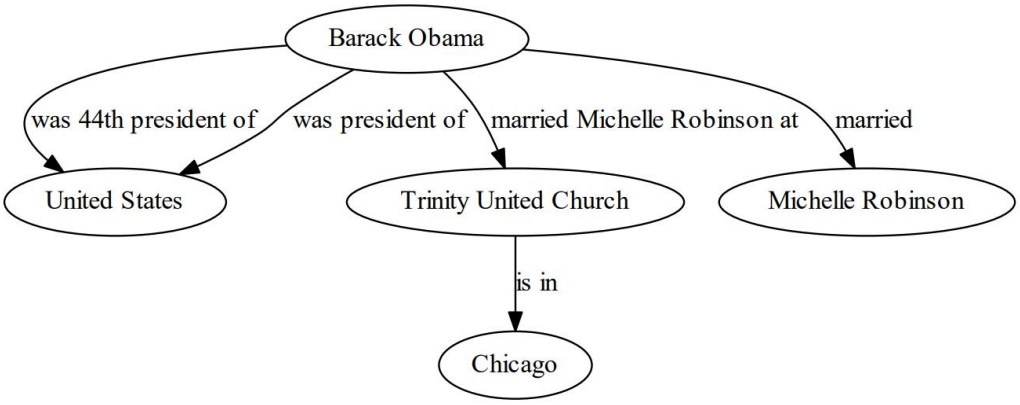
Step1: extraction of relation tuples, typically binary relations, from plain text. For the text, make use of the Entity Extraction to filter the entity int the attribute and the verb or verb phrases (chunk rule) for the relation. So the key in this step is judgement.

```
for triple in client.annotate(text):
    det = False
    if triple['subject'] in entity and triple['object'] in entity:
        for i in vb:
            if i in triple['relation']:
                det = True
                break
        if det == True:
            triples.append([triple['subject'], triple['relation'],  triple['object']])
```

Step2: Construct the KB from Triples. Given the knowledge triples, we need to index all the entities and relations. For example, get the entity set and relation set, and represent each triple using entity id and relation id. The key of this this step is .*zip*() method.

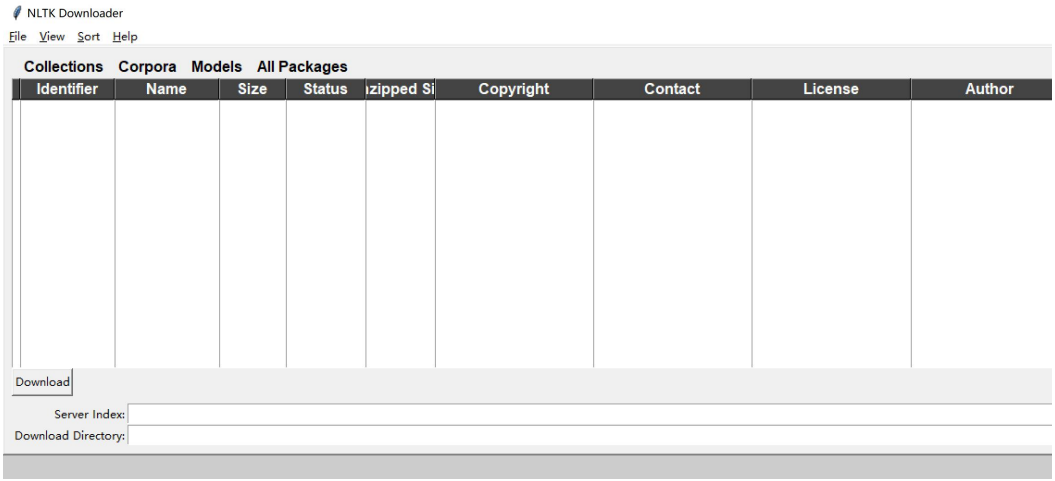
```
Entities: {0: 'Michelle Robinson', 1: 'Trinity United Church', 2: 'United States', 3: 'Barack Obama'}
Relations: {0: 'married Michelle Robinson at', 1: 'married', 2: 'was 44th president of', 3: 'was president of'}
Triples: [[3, 2, 2], [3, 3, 2], [3, 1, 0], [3, 0, 1]]
```

Step3: Visualize the KB using *graphviz*. Add the node for the attribution entities and add the relation edge between the subjects and objects. Then use the .*render*() method to show in graph way. As follow:

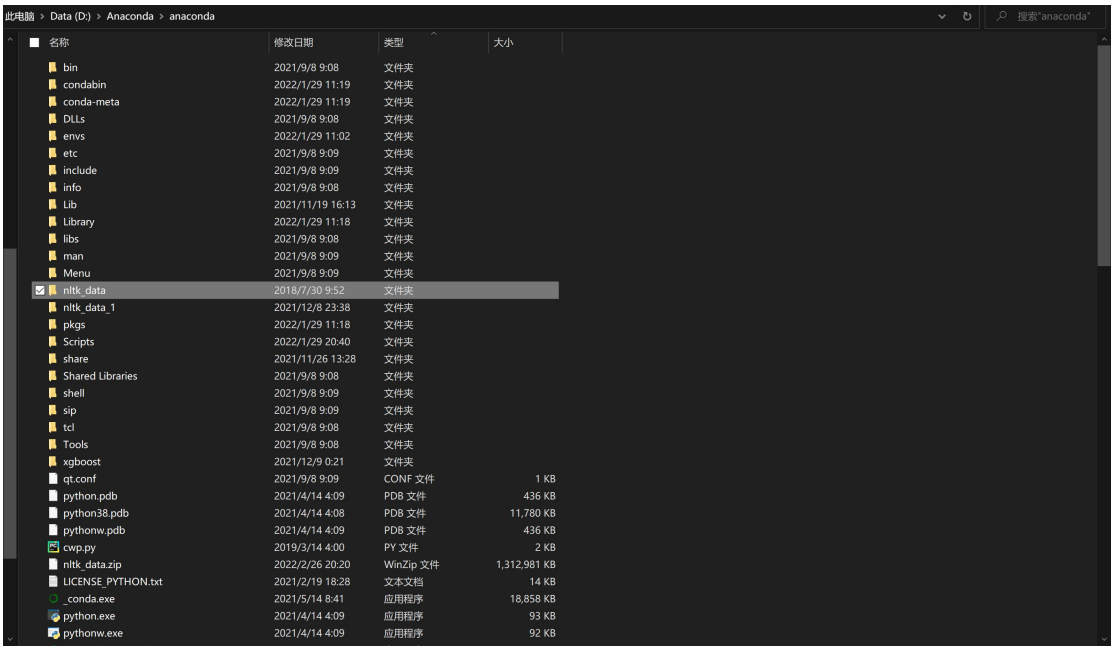


Problems:

The problem of $nltk.download$. When it comes to execute the commend $nltk.download()$, I get nothing in the dialog window. That's because the Server Index path which is invalid from GitHub.



So I can download from another Server Index: $http://www.nltk.org/nltk\_data/$. But the download speed was too slow, I just downloaded a $nltk\_data$ package directly from the Internet. And then, move the package into the home directory of the Anaconda.



And we should remember that we should unzip all the package in the all fold of *nltk_data*.

b. In the *EntityExtraction_Exercise*, I have found that the results of the run did not match expectations. The reason is that two consecutive nouns or adjectives plus nouns should be placed in one list, not separate. Just like 'Alfred' and 'Lennon' should be put in together. I ignore that and later I use the commend ' '.*join*() to combine two strings into one.

c. In the *Hypernym Relationship Extraction_Exercise*, there's one place that's been bothering me for a long time. After preparing the chunked result for subsequent Hearst pattern matching (*NP_*), we should find the hypernym and hyponyms on processed chunked results. And the difficult thing is writing the Hearst Patterns.

```
# Merge everything to get the final extractor
class HearstPatterns(object):
    def find_hyponyms(self, sentence):
        hearst_patterns = [("(NP_\w+ (, )?such as (NP_\w+ ?(, )?(and |or )?)+)", "first"),
                           ("((NP_\w+ ?(, )?)+(and |or )?other NP_\w+)","last"),
                           ("(NP_\w+ (, )?as (NP_\w+ ?(, )?(and |or )?)+)", "first"),
                           ("(NP_\w+ (, )?including (NP_\w+ ?(, )?(and |or )?)+)", "first"),
                           ("(NP_\w+ (, )?especially (NP_\w+ ?(, )?(and |or )?)+)", "first")]
```

As I was writing, I found that the details of regular expressions were very demanding. Because if you make a mistake, it will not match the text and the result will be empty. Though lots of debugging, I found that it's just an extra **space** effect the result. So we should be strictly careful for this.