Chapter 2: Playing with Grammar

Grammar is one of the main building blocks of language. Each human language, and programming language for that matter, has a set of rules that every person speaking it has to follow because otherwise, they risk not being understood. These grammatical rules can be uncovered using NLP and are useful for extracting data from sentences. For example, using information about the grammatical structure of text, we can parse out subjects, objects, and relationships between different entities.

In this chapter, you will learn how to use different packages to reveal the grammatical structure of words and sentences, as well as extract certain parts of sentences. We will cover the following topics:

- Counting nouns – plural and singular nouns
- Getting the dependency parse
- Splitting sentences into clauses
- Extracting noun chunks
- Extracting entities and relations
- Extracting subjects and objects of the sentence
- Finding references – anaphora resolution

Let's get started!

Technical requirements

Follow these steps to install the packages and models required for this chapter:

```
pip install inflect
python -m spacy download en_core_web_md
pip install textacy
```

For the *Finding references: anaphora resolution* recipe, we have to install the **`neuralcoref`** package. To install this package, use the following command:

```
pip install neuralcoref
```

In case, when running the code, you encounter errors that mention **spacy.strings.StringStore size changed**, you might need to install **neuralcoref** from the source:

```
pip uninstall neuralcoref
git clone https://github.com/huggingface/
neuralcoref.git
cd neuralcoref
pip install -r requirements.txt
pip install -e
```

For more information about installation and usage, see **https://github.com/huggingface/neuralcoref**.

Counting nouns – plural and singular nouns

In this recipe, we will do two things:

- Determine whether a noun is plural or singular
- Turn plural nouns into singular nouns and vice versa

You might need these two things in a variety of tasks: in making your chatbot speak in grammatically correct sentences, in coming up with text classification features, and so on.

Getting ready

We will be using **nltk** for this task, as well as the **inflect** module we described in *Technical requirements* section. The code for this chapter is located in the **Chapter02** directory of this book's GitHub repository. We will be working with the first part of the *Adventures of Sherlock Holmes* text, available in the **sherlock_holmes_1.txt** file.

How to do it...

We will be using code from ***Chapter 1***, *Learning NLP Basics*, to tokenize the text into words and tag them with parts of speech. Then, we will use one of two ways to determine if a noun is singular or plural, and then use the **inflect** module to change the number of the noun.

Your steps should be formatted like so:

1. Do the necessary imports:
```
import nltk
from nltk.stem import WordNetLemmatizer
import inflect
from Chapter01.pos_tagging import pos_tag_nltk
```
2. Read in the text file:
```
file = open(filename, "r", encoding="utf-8")
sherlock_holmes_text = file.read()
```
3. Remove newlines for better readability:
```
sherlock_holmes_text =
sherlock_holmes_text.replace("\n", " ")
```
4. Do part of speech tagging:
```
words_with_pos = pos_tag_nltk(sherlock_holmes_text)
```
5. Define the **`get_nouns`** function, which will filter out the nouns from all the words:
```
def get_nouns(words_with_pos):
    noun_set = ["NN", "NNS"]
    nouns = [word for word in words_with_pos if
             word[1] in noun_set]
    return nouns
```
6. Run the preceding function on the list of POS-tagged words and print it:
```
nouns = get_nouns(words_with_pos)
print(nouns)
```
The resulting list will be as follows:
```
[('woman', 'NN'), ('name', 'NN'), ('eyes', 'NNS'),
('whole', 'NN'), ('sex', 'NN'), ('emotion', 'NN'),
('akin', 'NN'), ('emotions', 'NNS'), ('cold',
'NN'), ('precise', 'NN'), ('mind', 'NN'),
('reasoning', 'NN'), ('machine', 'NN'), ('world',
'NN'), ('lover', 'NN'), ('position', 'NN'),
('passions', 'NNS'), ('gibe', 'NN'), ('sneer',
'NN'), ('things', 'NNS'), ('observer—excellent',
'NN'), ('veil', 'NN'), ('men', 'NNS'), ('motives',
'NNS'), ('actions', 'NNS'), ('reasoner', 'NN'),
('intrusions', 'NNS'), ('delicate', 'NN'),
('temperament', 'NN'), ('distracting', 'NN'),
('factor', 'NN'), ('doubt', 'NN'), ('results',
```

```
'NNS'), ('instrument', 'NN'), ('crack', 'NN'),
('high-power', 'NN'), ('lenses', 'NNS'),
('emotion', 'NN'), ('nature', 'NN'), ('woman',
'NN'), ('woman', 'NN'), ('memory', 'NN')]
```

7. To determine whether a noun is singular or plural, we have two options. The first option is to use the NLTK tags, where **`NN`** indicates a singular noun and **`NNS`** indicates a plural noun. The following function uses the NLTK tags and returns **`True`** if the input noun is plural:

```
def is_plural_nltk(noun_info):
    pos = noun_info[1]
    if (pos == "NNS"):
        return True
    else:
        return False
```

8. The other option is to use the **`WordNetLemmatizer`** class in the **`nltk.stem`** package. The following function returns **`True`** if the noun is plural:

```
def is_plural_wn(noun):
    wnl = WordNetLemmatizer()
    lemma = wnl.lemmatize(noun, 'n')
    plural = True if noun is not lemma else False
    return plural
```

9. The following function will change a singular noun into plural:

```
def get_plural(singular_noun):
    p = inflect.engine()
    return p.plural(singular_noun)
```

10. The following function will change a plural noun into singular:

```
def get_singular(plural_noun):
    p = inflect.engine()
    plural = p.singular_noun(plural_noun)
    if (plural):
        return plural
    else:
        return plural_noun
```

    We can now use the two preceding functions to return a list of nouns changed into plural or singular, depending on the original noun. The following code uses the **`is_plural_wn`** function to determine if the noun is plural. You can also use the **`is_plural_nltk`** function:

```
def plurals_wn(words_with_pos):
    other_nouns = []
    for noun_info in words_with_pos:
```

```
    for noun_info in words_with_pos:
        word = noun_info[0]
        plural = is_plural_wn(word)
        if (plural):
            singular = get_singular(word)
            other_nouns.append(singular)
        else:
            plural = get_plural(word)
            other_nouns.append(plural)
    return other_nouns
```

11. Use the preceding function to return a list of changed nouns:

```
other_nouns_wn = plurals_wn(nouns)
```

The result will be as follows:

```
['women', 'names', 'eye', 'wholes', 'sexes',
'emotions', 'akins', 'emotion', 'colds',
'precises', 'minds', 'reasonings', 'machines',
'worlds', 'lovers', 'positions', 'passion',
'gibes', 'sneers', 'thing', 'observer-excellents',
'veils', 'mens', 'motive', 'action', 'reasoners',
'intrusion', 'delicates', 'temperaments',
'distractings', 'factors', 'doubts', 'result',
'instruments', 'cracks', 'high-powers', 'lens',
'emotions', 'natures', 'women', 'women',
'memories']
```

How it works...

Number detection works in one of two ways. One is by reading the part of speech tag assigned by NLTK. If the tag is **NN**, then the noun is singular, and if it is **NNS**, then it's plural. The other way is to use the WordNet lemmatizer and to compare the lemma and the original word. The noun is singular if the lemma and the original input noun are the same, and plural otherwise.

To find the singular form of a plural noun and the plural form of a singular noun, we can use the **inflect** package. Its **plural** and **singular_noun** methods return the correct forms.

In *step 1*, we import the necessary modules and functions. You can find the **pos_tag_nltk** function in this book's GitHub repository, in the **Chapter01** module, in the **pos_tagging.py** file It uses the code we wrote for ***Chapter 1**, Learning NLP Basics*. In *step 2*, we read in the file's

wrote for *Chapter 1*, *Learning NLP Basics*. In *step 2*, we read in the file's contents into a string. In *step 3*, we remove newlines from the text; this is an optional step. In *step 4*, we use the **pos_tag_nltk** function defined in the code from the previous chapter to tag parts of speech for the words.

In *step 5*, we create the **get_nouns** function, which filters out the words that are singular or plural nouns. In this function, we use a list comprehension and keep only words that have the *NN* or *NNS* tags.

In *step 6*, we run the preceding function on the word list and print the result. As you will notice, NLTK tags several words incorrectly as nouns, such as *cold* and *precise*. These errors will propagate into the next steps, and it is something to keep in mind when working with NLP tasks.

In *steps 7* and *8*, we define two functions to determine whether a noun is singular or plural. In *step 7*, we define the **is_plural_nltk** function, which uses NLTK POS tagging information to determine if the noun is plural. In *step 8*, we define the **is_plural_wn** function, which compares the noun with its lemma, as determined by the NLTK lemmatizer. If those two forms are the same, the noun is singular, and if they are different, the noun is plural. Both functions can return incorrect results that will propagate downstream.

In *step 9*, we define the **get_plural** function, which will return the plural form of the noun by using the **inflect** package. In *step 10*, we define the **get_singular** function, which uses the same package to get the singular form of the noun. If there is no output from **inflect**, the function returns the input.

In *step 11*, we define the **plurals_wn** function, which takes in a list of words with the parts of speech that we got in *step 6* and changes plural nouns into singular and singular nouns into plural.

In *step 12*, we run the **plurals_wn** function on the nouns list. Most of the words are changed correctly; for example, *women* and *emotion*. We also see two kinds of error propagation, where either the part of speech or number of the noun were determined incorrectly. For example, the word *akins* appears here because *akin* was incorrectly labeled as a noun. On the other hand, the word *men* was incorrectly determined to be singular and resulted in the wrong output; that is, *mens*.

There's more...

The results will differ, depending on which **is_plural/is_singular** function you use. If you tag the word *men* with its part of speech, you will see that **NLTK** returns the **NNS** tag, which means that the word is plural. You can experiment with different inputs and see which function works best for you.

Getting the dependency parse

A dependency parse is a tool that shows dependencies in a sentence. For example, in the sentence *The cat wore a hat*, the root of the sentence in the verb, *wore*, and both the subject, *the cat*, and the object, *a hat*, are dependents. The dependency parse can be very useful in many NLP tasks since it shows the grammatical structure of the sentence, along with the subject, the main verb, the object, and so on. It can be then used in downstream processing.

Getting ready

We will use **spacy** to create the dependency parse. If you already downloaded it while working on the previous chapter, you do not need to do anything more. Otherwise, please follow the instructions at the beginning of ***Chapter 1***, *Learning NLP Basics*, to install the necessary packages.

How to do it...

We will take a few sentences from the **sherlock_holmes1.txt** file to illustrate the dependency parse. The steps are as follows:

1. Import **spacy**:
```
import spacy
```
2. Load the sentence to be parsed:
```
sentence = 'I have seldom heard him mention her
under any other name.'
```
3. Load the **spacy** engine:
```
nlp = spacy.load('en_core_web_sm')
```
4. Process the sentence using the **spacy** engine:
```
doc = nlp(sentence)
```
5. The dependency information will be contained in the **doc** object. We can see the dependency tags by looping through the tokens in **doc**:
```
for token in doc:
```

```
for token in doc:
    print(token.text, "\t", token.dep_, "\t",
    spacy.explain(token.dep_))
```

6. The result will be as follows. To learn what each of the tags means, use spaCy's **`explain`** function, which shows the meanings of the tags:

```
I         nsubj   nominal subject
have      aux     auxiliary
seldom    advmod          adverbial modifier
heard     ROOT    None
him       nsubj   nominal subject
mention           ccomp   clausal complement
her       dobj    direct object
under     prep    prepositional modifier
any       det     determiner
other     amod    adjectival modifier
name      pobj    object of preposition
.         punct   punctuation
```

7. To explore the dependency parse structure, we can use the attributes of the **`Token`** class. Using its **`ancestors`** and **`children`** attributes, we can get the tokens that this token depends on and the tokens that depend on it, respectively. The code to get these ancestors is as follows:

```
for token in doc:
    print(token.text)
    ancestors = [t.text for t in token.ancestors]
    print(ancestors)
```

The output will be as follows:

```
I
['heard']
have
['heard']
seldom
['heard']
heard
[]
him
['mention', 'heard']
mention
['heard']
her
['mention', 'heard']
under
```

```
['mention', 'heard']
any
['name', 'under', 'mention', 'heard']
other
['name', 'under', 'mention', 'heard']
name
['under', 'mention', 'heard']
.
['heard']
```

8. To see all the **`children token`**, use the following code:

```
for token in doc:
    print(token.text)
    children = [t.text for t in token.children]
    print(children)
```

9. The output will be as follows:

```
I
[]
have
[]
seldom
[]
heard
['I', 'have', 'seldom', 'mention', '.']
him
[]
mention
['him', 'her', 'under']
her
[]
under
['name']
any
[]
other
[]
name
['any', 'other']
.
[]
```

10. We can also see the subtree that the token is in:

```
for token in doc:
```

```
    print(token.text)
    subtree = [t.text for t in token.subtree]
    print(subtree)
```

This will produce the following output:

```
I
['I']
have
['have']
seldom
['seldom']
heard
['I', 'have', 'seldom', 'heard', 'him', 'mention',
'her', 'under', 'any', 'other', 'name', '.']
him
['him']
mention
['him', 'mention', 'her', 'under', 'any', 'other',
'name']
her
['her']
under
['under', 'any', 'other', 'name']
any
['any']
other
['other']
name
['any', 'other', 'name']
.
['.']
```

How it works...

The **spacy** NLP engine does the dependency parse as part of its overall analysis. The dependency parse tags explain the role of each word in the sentence. **ROOT** is the main word that all the other words depend on, usually the verb.

From the subtrees that each word is part of, we can see the grammatical phrases that appear in the sentence, such as the **noun phrase (NP)** *any other name* and **prepositional phrase (PP)** *under any other name*.

The dependency chain can be seen by following the ancestor links for each word. For example, if we look at the word *name*, we will see that its ancestors are *under*, *mention*, and *heard*. The immediate parent of *name* is *under*, *under's* parent is *mention*, and *mention's* parent is *heard*. A dependency chain will always lead to the root, or the main word, of the sentence.

In *step 1*, we import the **`spacy`** package. In *step 2*, we initialize the variable sentence that contains the sentence to be parsed. In *step 3*, we load the **`spacy`** engine and in *step 4*, we use the engine to process the sentence.

In *step 5*, we print out each token's dependency tag and use the **`spacy.explain`** function to see what those tags mean.

In *step 6*, we print out the ancestors of each token. The ancestors will start at the parent and go up until they reach the root. For example, the parent of *him* is *mention*, and the parent of *mention* is *heard*, so both *mention* and *heard* are listed as ancestors of *him*.

In *step 7*, we print children of each token. Some tokens, such as *have*, do not have any children, while others have several. The token that will always have children, unless the sentence consists of one word, is the root of the sentence; in this case, *heard*.

In *step 8*, we print the subtree for each token. For example, the word *under* is in the subtree *under any other name*.

See also

The dependency parse can be visualized graphically using the **`displacy`** package, which is part of **`spacy`**. Please see ***Chapter 8***, *Visualizing Text Data*, for a detailed recipe on how to perform visualization.

Splitting sentences into clauses

When we work with text, we frequently deal with compound (sentences with two parts that are equally important) and complex sentences (sentences with one part depending on another). It is sometimes useful to split these composite sentences into its component clauses for easier processing down the line. This recipe uses the dependency parse from the previ-

ous recipe.

Getting ready

You will only need the **spacy** package in this recipe.

How to do it...

We will work with two sentences, *He eats cheese, but he won't eat ice cream* and *If it rains later, we won't be able to go to the park*. Other sentences may turn out to be more complicated to deal with, and I leave it as an exercise for you to split such sentences. Follow these steps:

1. Import the **spacy** package:
```
import spacy
```
2. Load the **spacy** engine:
```
nlp = spacy.load('en_core_web_sm')
```
3. Set the sentence to **He eats cheese, but he won't eat ice cream**:
```
sentence = "He eats cheese, but he won't eat ice
cream."
```
4. Process the sentence with the **spacy** engine:
```
doc = nlp(sentence)
```
5. It is instructive to look at the structure of the input sentence by printing out the part of speech, dependency tag, ancestors, and children of each token. This can be accomplished using the following code:
```
for token in doc:
    ancestors = [t.text for t in token.ancestors]
    children = [t.text for t in token.children]
    print(token.text, "\t", token.i, "\t",
          token.pos_, "\t", token.dep_, "\t",
          ancestors, "\t", children)
```
6. We will use the following function to find the root token of the sentence, which is usually the main verb. In instances where there is a dependent clause, it is the verb of the independent clause:
```
def find_root_of_sentence(doc):
    root_token = None
    for token in doc:
        if (token.dep_ == "ROOT"):
            root_token = token
    return root_token
```

7. We will now find the root token of the sentence:

```
root_token = find_root_of_sentence(doc)
```

8. We can now use the following function to find the other verbs in the sentence:

```
def find_other_verbs(doc, root_token):
    other_verbs = []
    for token in doc:
        ancestors = list(token.ancestors)
        if (token.pos_ == "VERB" and len(ancestors)
== 1\
            and ancestors[0] == root_token):
            other_verbs.append(token)
    return other_verbs
```

9. Use the preceding function to find the remaining verbs in the sentence:

```
other_verbs = find_other_verbs(doc, root_token)
```

We will use the following function to find the token spans for each verb:

```
def get_clause_token_span_for_verb(verb, doc,
all_verbs):
    first_token_index = len(doc)
    last_token_index = 0
    this_verb_children = list(verb.children)
    for child in this_verb_children:
        if (child not in all_verbs):
            if (child.i < first_token_index):
                first_token_index = child.i
            if (child.i > last_token_index):
                last_token_index = child.i
    return(first_token_index, last_token_index)
```

10. We will put together all the verbs in one array and process each using the preceding function. This will return a tuple of start and end indices for each verb's clause:

```
token_spans = []
all_verbs = [root_token] + other_verbs
for other_verb in all_verbs:
    (first_token_index, last_token_index) = \
     get_clause_token_span_for_verb(other_verb,
                                    doc, all_verbs)
    token_spans.append((first_token_index,
                        last_token_index))
```

11. Using the start and end indices, we can now put together token spans for each clause. We sort the **sentence_clauses** list at the end so that the clauses are in the order they appear in the sentence:

```
sentence_clauses = []
for token_span in token_spans:
    start = token_span[0]
    end = token_span[1]
    if (start < end):
        clause = doc[start:end]
        sentence_clauses.append(clause)
sentence_clauses = sorted(sentence_clauses,
                          key=lambda tup: tup[0])
```

12. Now, we can print the final result of the processing for our initial sentence; that is, **He eats cheese, but he won't eat ice cream**:

```
clauses_text = [clause.text for clause in
sentence_clauses]
print(clauses_text)
```

The result is as follows:

```
['He eats cheese,', 'he won't eat ice cream']
```

IMPORTANT NOTE

The code in this section will work for some cases, but not others; I encourage you to test it out on different cases and amend the code.

How it works...

The way the code works is based on the way complex and compound sentences are structured. Each clause contains a verb, and one of the verbs is the main verb of the sentence (root). The code looks for the root verb, always marked with the **ROOT** dependency tag in spaCy processing, and then looks for the other verbs in the sentence.

The code then uses the information about each verb's children to find the left and right boundaries of the clause. Using this information, the code then constructs the text of the clauses. A step-by-step explanation follows.

In *step 1*, we import the **spaCy** package and in *step 2*, we load the **spacy** engine. In *step 3*, we set the sentence variable and in *step 4*, we process it using the **spacy** engine. In *step 5*, we print out the dependency parse information. It will help us determine how to split the sentence into clauses.

In *step 6*, we define the **find_root_of_sentence** function, which re-

turns the token that has a dependency tag of **ROOT**. In *step 7*, we find the root of the sentence we are using as an example.

In *step 8*, we define the **find_other_verbs** function, which will find other verbs in the sentence. In this function, we look for tokens that have the **VERB** part of speech tag and has the root token as its only ancestor. In *step 9*, we apply this function.

In *step 10*, we define the **get_clause_token_span_for_verb** function, which will find the beginning and ending index for the verb. The function goes through all the verb's children; the leftmost child's index is the beginning index, while the rightmost child's index is the ending index for this verb's clause.

In *step 11*, we use the preceding function to find the clause indices for each verb. The **token_spans** variable contains the list of tuples, where the first tuple element is the beginning clause index and the second tuple element is the ending clause index.

In *step 12*, we create token **Span** objects for each clause in the sentence using the list of beginning and ending index pairs we created in *step 11*. We get the **Span** object by slicing the **Doc** object and then appending the resulting **Span** objects to a list. As a final step, we sort the list to make sure that the clauses in the list are in the same order as in the sentence.

In *step 13*, we print the clauses in our sentence. You will notice that the word *but* is missing, since its parent is the root verb *eats*, although it appears in the other clause. The exercise of including *but* is left to you.

Extracting noun chunks

Noun chunks are known in linguistics as **noun phrases**. They represent nouns and any words that depend on and accompany nouns. For example, in the sentence *The big red apple fell on the scared cat*, the noun chunks are *the big red apple* and *the scared cat*. Extracting these noun chunks is instrumental to many other downstream NLP tasks, such as named entity recognition and processing entities and relationships between them. In this recipe, we will explore how to extract named entities from a piece of text.

Getting ready

Getting ready

We will be using the **spacy** package, which has a function for extracting noun chunks and the text from the **sherlock_holmes_1.txt** file as an example.

In this section, we will use another spaCy language model, **en_core_web_md**. Follow the instructions in the *Technical requirements* section to learn how to download it.

How to do it...

Use the following steps to get the noun chunks from a piece of text:

1. Import the **spacy** package and the **read_text_file** from the code files of ***Chapter 1***:
   ```
   import spacy
   from Chapter01.dividing_into_sentences import
   read_text_file
   ```
   IMPORTANT NOTE
   If you are importing functions from other chapters, run it from the directory that precedes **Chapter02** *and use the* **python -m Chapter02.extract_noun_chunks** *command.*
2. Read in the **sherlock_holmes_1.txt** file:
   ```
   text = read_text_file("sherlock_holmes_1.txt")
   ```
3. Initialize the **spacy** engine and then use it to process the text:
   ```
   nlp = spacy.load('en_core_web_md')
   doc = nlp(text)
   ```
4. The noun chunks are contained in the **doc.noun_chunks** class variable. We can print out the chunks:
   ```
   for noun_chunk in doc.noun_chunks:
       print(noun_chunk.text)
   ```
   This is the partial result. See this book's GitHub repository for the full printout, which can be found in the **Chapter02/all_text_noun_chunks.txt** file:
   ```
   Sherlock Holmes
   she
   the_ woman
   I
   him
   her
   any other name
   ```

```
his eyes
she
the whole
…
```

How it works…

The spaCy **Doc** object, as we saw in the previous recipe, contains information about grammatical relationships between words in a sentence. Using this information, spaCy determines noun phrases or chunks contained in the text.

In *step 1*, we import spacy and the **read_text_file** function from the **Chapter01** module. In *step 2*, we read in the text from the **sherlock_holmes_1.txt** file.

In *step 3*, we initialize the **spacy** engine with a different model, **en_core_web_md**, which is larger and will most likely give better results. There is also the large model, **en_core_web_lg**, which is even larger. It will give better results, but the processing will be slower. After loading the engine, we run it on the text we loaded in *step 2*.

In *step 4*, we print out the noun chunks that appear in the text. As you can see, it gets the pronouns, nouns, and noun phrases that are in the text correctly.

There's more…

Noun chunks are spaCy **Span** objects and have all their properties. See the official documentation at https://spacy.io/api/token.

Let's explore some properties of noun chunks:

1. Import the **spacy** package:
```
import spacy
```
2. Load the **spacy** engine:
```
nlp = spacy.load('en_core_web_sm')
```
3. Set the sentence to *All emotions, and that one particularly, were abhorrent to his cold, precise but admirably balanced mind*:
```
sentence = "All emotions, and that one
particularly, were abhorrent to his cold, precise
```

```
but admirably balanced mind."
```

4. Process the sentence with the **spacy** engine:

```
doc = nlp(sentence)
```

5. Let's look at the noun chunks in this sentence:

```
for noun_chunk in doc.noun_chunks:
    print(noun_chunk.text)
```

6. This is the result:

```
All emotions
his cold, precise but admirably balanced mind
```

7. Some of the basic properties of noun chunks are its start and end offsets; we can print them out together with the noun chunks:

```
for noun_chunk in doc.noun_chunks:
    print(noun_chunk.text, "\t", noun_chunk.start,
"\t",
          noun_chunk.end)
```

   The result will be as follows:

```
All emotions     0       2
his cold, precise but admirably balanced
mind    11      19
```

8. We can also print out the sentence where the noun chunk belongs:

```
for noun_chunk in doc.noun_chunks:
    print(noun_chunk.text, "\t", noun_chunk.sent)
```

   Predictably, this results in the following:

```
All emotions     All emotions, and that one
particularly, were abhorrent to his cold, precise
but admirably balanced mind.
his cold, precise but admirably balanced
mind    All emotions, and that one particularly,
were abhorrent to his cold, precise but admirably
balanced mind.
```

9. Just like a sentence, any noun chunk includes a root, which is the token that all other tokens depend on. In a noun phrase, that is the noun:

```
for noun_chunk in doc.noun_chunks:
    print(noun_chunk.text, "\t",
noun_chunk.root.text)
```

10. The result will be as follows:

```
All emotions     emotions
his cold, precise but admirably balanced
mind    mind
```

11. Another very useful property of **Span** is **similarity**, which is the semantic similarity of different texts. Let's try it out. We will load an-

other noun chunk, **emotions**, and process it using **spacy**:

```
other_span = "emotions"
other_doc = nlp(other_span)
```

12. We can now compare it to the noun chunks in the sentence by using this code:

```
for noun_chunk in doc.noun_chunks:
    print(noun_chunk.similarity(other_doc))
```

This is the result:

```
UserWarning: [W007] The model you're using has no
word vectors loaded, so the result of the
Span.similarity method will be based on the tagger,
parser and NER, which may not give useful
similarity judgements. This may happen if you're
using one of the small models, e.g.
`en_core_web_sm`, which don't ship with word
vectors and only use context-sensitive tensors. You
can always add your own word vectors, or use one of
the larger models instead if available.
  print(noun_chunk.similarity(other_doc))
All emotions
0.373233604751925
his cold, precise but admirably balanced mind
0.030945358271699138
```

13. Although the result makes sense, with *all emotions* being more similar to *emotions* than to *his cold, precise but admirably balanced mind*, we get a warning. In order to fix this, we will use the medium **spacy** model, which contains vector representations for words. Substitute this line for the line in *step 2*; the rest of the code will remain the same:

```
nlp = spacy.load('en_core_web_md')
```

14. Now, when we run this code with the new model, we get this result:

```
All emotions
0.8876554549427152
that one
0.37378867755652434
his cold, precise but admirably balanced mind
0.5102475977383759
```

The result shows the similarity of **all emotions** to **emotions** being very high, 0.89, and to **his cold, precise but admirably balanced mind**, 0.51. We can also see that the larger model detects another noun chunk, *that one*.

IMPORTANT NOTE

A larger **spaCy** *model, such as* **en_core_web_md**, *takes up more space, but is more precise.*

See also

The topic of semantic similarity will be explored in more detail in ***Chapter 3***, *Representing Text: Capturing Semantics*.

Extracting entities and relations

It is possible to extract triplets of the subject entity-relation-object entity from documents, which are frequently used in knowledge graphs. These triplets can then be analyzed for further relations and inform other NLP tasks, such as searches.

Getting ready

For this recipe, we will need another Python package based on **spacy**, called **textacy**. The main advantage of this package is that it allows regular expression-like searching for tokens based on their part of speech tags. See the installation instructions in the *Technical requirements* section at the beginning of this chapter for more information.

How to do it...

We will find all verb phrases in the text, as well as all the noun phrases (see the previous section). Then, we will find the left noun phrase (subject) and the right noun phrase (object) that relate to a particular verb phrase. We will use two simple sentences, *All living things are made of cells* and *Cells have organelles*. Follow these steps:

1. Import **spaCy** and **textacy**:
```
import spacy
import textacy
from Chapter02.split_into_clauses import
find_root_of_sentence
```
2. Load the **spacy** engine:
```
nlp = spacy.load('en_core_web_sm')
```
3. We will get a list of sentences that we will be processing:

```
sentences = ["All living things are made of
cells.",
             "Cells have organelles."]
```

4. In order to find verb phrases, we will need to compile regular expression-like patterns for the part of speech combinations of the words that make up the verb phrase. If we print out parts of speech of verb phrases of the two preceding sentences, *are made of and have*, we will see that the part of speech sequences are **AUX**, **VERB**, **ADP**, and **AUX**.

```
verb_patterns = [[{"POS":"AUX"}, {"POS":"VERB"},
                 {"POS":"ADP"}],
                [{"POS":"AUX"}]]
```

5. The **contains_root** function checks if a verb phrase contains the root of the sentence:

```
def contains_root(verb_phrase, root):
    vp_start = verb_phrase.start
    vp_end = verb_phrase.end
    if (root.i >= vp_start and root.i <= vp_end):
        return True
    else:
        return False
```

6. The **get_verb_phrases** function gets the verb phrases from a spaCy **Doc** object:

```
def get_verb_phrases(doc):
    root = find_root_of_sentence(doc)
    verb_phrases = textacy.extract.matches(doc,
                                           verb_pat
terns)
    new_vps = []
    for verb_phrase in verb_phrases:
        if (contains_root(verb_phrase, root)):
            new_vps.append(verb_phrase)
    return new_vps
```

7. The **longer_verb_phrase** function finds the longest verb phrase:

```
def longer_verb_phrase(verb_phrases):
    longest_length = 0
    longest_verb_phrase = None
    for verb_phrase in verb_phrases:
        if len(verb_phrase) > longest_length:
            longest_verb_phrase = verb_phrase
    return longest_verb_phrase
```

8. The **find_noun_phrase** function will look for noun phrases either on

the left- or right-hand side of the main verb phrase:

```
def find_noun_phrase(verb_phrase, noun_phrases,
side):
    for noun_phrase in noun_phrases:
        if (side == "left" and \
            noun_phrase.start < verb_phrase.start):
            return noun_phrase
        elif (side == "right" and \
              noun_phrase.start >
verb_phrase.start):
            return noun_phrase
```

9. In this function, we will use the preceding functions to find triplets of subject-relation-object in the sentences:

```
def find_triplet(sentence):
    doc = nlp(sentence)
    verb_phrases = get_verb_phrases(doc)
    noun_phrases = doc.noun_chunks
    verb_phrase = None
    if (len(verb_phrases) > 1):
        verb_phrase = \
        longer_verb_phrase(list(verb_phrases))
    else:
        verb_phrase = verb_phrases[0]
    left_noun_phrase =
find_noun_phrase(verb_phrase,
                                          noun_phrase
s,
                                          "left")
    right_noun_phrase =
find_noun_phrase(verb_phrase,
                                           noun_phras
es,
                                           "right")
    return (left_noun_phrase, verb_phrase,
            right_noun_phrase)
```

10. We can now loop through our sentence list to find its relation triplets:

```
for sentence in sentences:
    (left_np, vp, right_np) =
find_triplet(sentence)
    print(left_np, "\t", vp, "\t", right_np)
```

11. The result will be as follows:

11. The result will be as follows:

```
All living things        are made of     cells
Cells    have    organelles
```

How it works...

The code finds triplets of subject-relation-object by looking for the root verb phrase and finding its surrounding nouns. The verb phrases are found using the **textacy** package, which provides a very useful tool for finding patterns of words of certain parts of speech. In effect, we can use it to write small grammars describing the necessary phrases.

IMPORTANT NOTE

The **textacy** *package, while very useful, is not bug-free, so use it with caution.*

Once the verb phrases have been found, we can prune through the sentence noun chunks to find those that are around the verb phrase containing the root.

A step-by-step explanation follows.

In *step 1*, we import the necessary packages and the **find_root_of_sentence** function from the previous recipe. In *step 2*, we initialize the **spacy** engine, and in *step 3*, we initialize a list with the sentences we will be using.

In *step 4*, we compile part of speech patterns that we will use for finding relations. For these two sentences, the patterns are **AUX**, **VERB**, **ADP**, and **AUX**.

In *step 5*, we create the **contains_root** function, which will make sure that a verb phrase contains the root of the sentence. It does that by checking the index of the root and making sure that it falls within the verb phrase span boundaries.

In *step 6*, we create the **get_verb_phrases** function, which extracts all the verb phrases from the **Doc** object that is passed in. It uses the part of speech patterns we created in *step 4*.

In *step 7*, we create the **longer_verb_phrase** function, which will find the longest verb phrase from a list. We do this because some verb phrases

the longest verb phrase from a list. We do this because some verb phrases might be shorter than necessary. For example, in the sentence *All living things are made of cells*, both *are* and *are made of* will be found.

In *step 8*, we create the **`find_noun_phrase`** function, which finds noun phrases on either side of the verb. We specify the side as a parameter.

In *step 9*, we create the **`find_triplet`** function, which will find triplets of subject-relation-object in a sentence. In this function, first, we process the sentence with spaCy. Then, we use the functions defined in the previous steps to find the longest verb phrase and the nouns to the left- and right-hand sides of it.

In *step 10*, we apply the **`find_triplet`** function to the two sentences we defined at the beginning. The resulting triplets are correct.

In this recipe, we made a few assumptions that will not always be correct. The first assumption is that there will only be one main verb phrase. The second assumption is that there will be a noun chunk on either side of the verb phrase. Once we start working with sentences that are complex or compound, or contain relative clauses, these assumptions no longer hold. I leave it as an exercise for you to work with more complex cases.

There's more...

Once you've parsed out the entities and relations, you might want to input them into a knowledge graph for further use. There are a variety of tools you can use to work with knowledge graphs, such as *neo4j*.

Extracting subjects and objects of the sentence

Sometimes, we might need to find the subject and direct objects of the sentence, and that can easily be accomplished with the **`spacy`** package.

Getting ready

We will be using the dependency tags from **`spacy`** to find subjects and objects.

How to do it...

We will use the **subtree** attribute of tokens to find the complete noun chunk that is the subject or direct object of the verb (see the *Getting the dependency parse* recipe for more information). Let's get started:

1. Import **spacy**:
```
import spacy
```
2. Load the **spacy** engine:
```
nlp = spacy.load('en_core_web_sm')
```
3. We will get the list of sentences we will be processing:
```
sentences=["The big black cat stared at the small
dog.",
           "Jane watched her brother in the
evenings."]
```
4. We will use two functions to find the subject and the direct object of the sentence. These functions will loop through the tokens and return the subtree that contains the token with **subj** or **dobj** in the dependency tag, respectively. Here is the subject function:
```
def get_subject_phrase(doc):
    for token in doc:
        if ("subj" in token.dep_):
            subtree = list(token.subtree)
            start = subtree[0].i
            end = subtree[-1].i + 1
            return doc[start:end]
```
5. Here is the direct object function. If the sentence does not have a direct object, it will return **None**:
```
def get_object_phrase(doc):
    for token in doc:
        if ("dobj" in token.dep_):
            subtree = list(token.subtree)
            start = subtree[0].i
            end = subtree[-1].i + 1
            return doc[start:end]
```
6. We can now loop through the sentences and print out their subjects and objects:
```
for sentence in sentences:
    doc = nlp(sentence)
    subject_phrase = get_subject_phrase(doc)
    object_phrase = get_object_phrase(doc)
    print(subject_phrase)
```

```
    print(object_phrase)
```

The result will be as follows. Since the first sentence does not have a direct object, **None** is printed out:

```
The big black cat
None
Jane
her brother
```

How it works...

The code uses the **spacy** engine to parse the sentence. Then, the subject function loops through the tokens, and if the dependency tag contains **subj**, it returns that token's subtree, which is a **Span** object. There are different subject tags, including **nsubj** for regular subjects and **nsubjpass** for subjects of passive sentences, so we want to look for both.

The object function works exactly the same as the subject function, except it looks for the token that has **dobj** (direct object) in its dependency tag. Since not all sentences have direct objects, it returns **None** in those cases.

In *step 1*, we import **spaCy**, and in *step 2*, we load the **spacy** engine. In *step 3*, we initialize a list with the sentences we will be processing.

In *step 4*, we create the **get_subject_phrase** function, which gets the subject of the sentence. It looks for the token that has a dependency tag that contains **subj** and then returns the subtree that contains that token. There are several subject dependency tags, including **nsubj** and **nsubjpass** (for a subject of a passive sentence), so we look for the most general pattern.

In *step 5*, we create the **get_object_phrase** function, which gets the direct object of the sentence. It works similarly to the **get_subject_phrase**, but looks for the *dobj* dependency tag instead of a tag that contains *"subj"*.

In *step 6*, we loop through the list of sentences we created in *step 3*, and use the preceding functions to find the subjects and direct objects in the sentences. For the sentence *The big black cat stared at the small dog*, the subject is *the big black cat*, and there is no direct object (*the small dog* is the object of the preposition *at*). For the sentence *Jane watched her brother in the evenings*, the subject is *Jane* and the direct object is *her*

brother.

There's more...

We can look for other objects; for example, the dative objects of verbs such as *give* and objects of prepositional phrases. The functions will look very similar, with the main difference being the dependency tags; that is, **`dative`** for the dative object function and **`pobj`** for the prepositional object function. The prepositional object function will return a list since there can be more than one prepositional phrase in a sentence. Let's take a look:

1. The dative object function checks the tokens for the **`dative`** tag. It returns **`None`** if there are no dative objects:
```
def get_dative_phrase(doc):
    for token in doc:
        if ("dative" in token.dep_):
            subtree = list(token.subtree)
            start = subtree[0].i
            end = subtree[-1].i + 1
            return doc[start:end]
```
2. Here is the prepositional object function. It returns a list of objects of prepositions, but will be empty if there are none:
```
def get_prepositional_phrase_objs(doc):
    prep_spans = []
    for token in doc:
        if ("pobj" in token.dep_):
            subtree = list(token.subtree)
            start = subtree[0].i
            end = subtree[-1].i + 1
            prep_spans.append(doc[start:end])
    return prep_spans
```
3. The prepositional phrase objects in the sentence *Jane watched her brother in the evenings* are as follows:
```
[the evenings]
```
4. And here is the dative object in the sentence *Laura gave Sam a very interesting book*:
```
Sam
```
It is left as an exercise for you to find the actual prepositional phrases with prepositions intact instead of just the noun phrases that are dependent on these prepositions.

Finding references – anaphora resolution

When we work on problems of extracting entities and relations from text (see the *Extracting entities and relations* recipe), we are faced with real text, and many of our entities might end up being extracted as pronouns, such as *she* or *him*. In order to tackle this issue, we need to perform **anaphora resolution**, or the process of substituting the pronouns with their referents.

Getting ready

For this task, we will be using a **spaCy** extension written by *Hugging Face* called **neuralcoref** (see **https://github.com/huggingface/neuralcoref**). As the name suggests, it uses neural networks to resolve pronouns. To install the package, use the following command:

```
pip install neuralcoref
```

How to do it...

Your steps should be formatted like so:

1. Import **spacy** and **neuralcoref**:
```
import spacy
import neuralcoref
```
2. Load the **spaCy** engine and add **neuralcoref** to its pipeline:
```
nlp = spacy.load('en_core_web_sm')
neuralcoref.add_to_pipe(nlp)
```
3. We will process the following short text:
```
text = "Earlier this year, Olga appeared on a new
song. She was featured on one of the tracks. The
singer is assuring that her next album will be
worth the wait."
```
4. Now that **neuralcoref** is part of the pipeline, we just process the text using **spaCy** and then output the result:
```
doc = nlp(text)
print(doc._.coref_resolved)
```
The output will be as follows:
```
Earlier this year, Olga appeared on a new song.
```

```
Olga was featured on one of the tracks. Olga is
assuring that Olga next album will be worth the
wait.
```

How it works...

In *step 1*, we import the necessary packages. In *step 2*, we load the **spacy** engine and then add **neuralcoref** to its pipeline. In *step 3*, we initialize the **text** variable with the short text we will be using.

In *step 4*, we use the **spacy** engine to process the text and then print out the text with the pronouns resolved. You can see that the pronouns *she* and *her*, and even the phrase **The singer**, were all correctly substituted with the name **Olga**.

The **neuralcoref** package uses custom **spacy** attributes that are set by using an underscore and the attribute name. The **coref_resolved** variable is a custom attribute that is set on a **Doc** object. To learn more about **spaCy** custom attributes, see **https://spacy.io/usage/processing-pipelines#custom-components-attributes**.

There's more...

The **neuralcoref** package did a good job of recognizing different references to **Olga** in the previous section. However, if we use an unusual name, it might not work correctly. Here, we are using an example from the Hugging Face GitHub:

1. Let's use the following short text:
```
text = "Deepika has a dog. She loves him. The movie
star has always been fond of animals."
```
2. Upon processing this text using the preceding code, we get the following output:
```
Deepika has a dog. Deepika loves Deepika. Deepika
has always been fond of animals.
```
3. Because the name *Deepika* is an unusual name, the model has trouble figuring out whether this person is a man or a woman and resolves the pronoun *him* to *Deepika*, although it is incorrect. In order to solve this problem, we can help it by characterizing who *Deepika* actually is. We will add **neuralcoref** to the **spacy** pipe, as follows:
```
neuralcoref.add_to_pipe(nlp, conv_dict={'Deepika':
```

```
    ['woman']})
```

4. Now, let's process the result, as we did previously:

```
    doc = nlp(text)
    print(doc._.coref_resolved)
```

The output will be as follows:

```
    Deepika has a dog. Deepika loves a dog. Deepika has
    always been fond of animals.
```

Once we give the coreference resolution module more information, it gives the correct output.