2

THE TEXT-PROCESSING PIPELINE

Now that you understand the structure of an NLP application, it's time to see these underlying concepts in action. In this chapter, you'll install spaCy and set up your working environment. Then you'll learn about the *text-processing pipeline*, a series of basic NLP operations you'll use to determine the meaning and intent of a discourse. These operations include tokenization, lemmatization, part-of-speech tagging, syntactic dependency parsing, and named entity recognition.

Setting Up Your Working Environment

Before you start using spaCy, you need to set up a working environment by installing the following software components on your machine:

- Python 2.7 or later, or 3.4 or later
- The spaCy library
- A statistical model for spaCy

You'll need Python 2.7 or later, or 3.4 or later to use spaCy v2.0. Download it at *https://www.python.org/downloads/* and follow the instructions to set up a Python environment. Next, install spaCy in your Python environment using `pip` by running the following command:

```
$ pip install spacy
```

If you have more than one Python installation on your system, select the `pip` executable associated with the Python installation you want to use. For instance, if you want to use spaCy with Python 3.5, you'd run the following command:

```
$ pip3.5 install spacy
```

If you already have spaCy installed on your system, you might want to upgrade it to a new release. The examples in this book assume you use spaCy v2.0.x or later. You can verify which version of spaCy you have installed with the following command:

```
$ python -m spacy info
```

Once again, you might need to replace the `python` command with the command for the python executable used in your particular environment, say, `python3.5`. From now on, we'll use `python` and `pip` regardless of the executables your system uses.

If you decide to upgrade your installed spaCy package to the latest version, you can do this using the following `pip` command:

```
$ pip install -U spacy
```

Installing Statistical Models for spaCy

The spaCy installation doesn't include statistical models that you'll need when you start using the library. The statistical models contain knowledge collected about the particular language from a set of sources. You must separately download and install each model you want to use.

Several pretrained statistical models are available for different languages. For English, for example, the following models are available for

download from spaCy's website: `en_core_web_sm`, `en_core_web_md`, `en_core_web_lg`, and `en_vectors_web_lg`. The models use the following naming convention: *lang_type_genre_size*. Lang specifies the language. Type indicates the model's capabilities (for example, core is a general-purpose model that includes vocabulary, syntax, entities, and vectors). Genre indicates the type of text the model has been trained on: web (such as Wikipedia or similar media resources) or news (news articles). Size indicates how large the model is: `lg` is large, `md` is medium, and `sm` is small. The larger the model is, the more disk space it requires. For example, the `en_vectors_web_lg-2.1.0` model takes 631MB, whereas `en_core_web_sm-2.1.0` takes only 10MB.

To follow along with the examples provided in this book, `en_core_web_sm` (the most lightweight model) will work fine. spaCy will choose it by default when you use spaCy's download command:

```
$ python -m spacy download en
```

The `en` shortcut link in the command instructs spaCy to download and install the best-matching default model for the English language. The best-matching model, in this context, means the one that is generated for the specified language (English in this example), a general purpose model, and the most lightweight.

To download a specific model, you must specify its name, like this:

```
$ python -m spacy download en_core_web_md
```

Once installed, you can load the model using this same shortcut you specified during the installation:

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

Basic NLP Operations with spaCy

Let's begin by performing a chain of basic NLP operations that we call a processing pipeline. spaCy does all these operations for you behind the scenes, allowing you to concentrate on your application's specific logic. Figure 2-1 provides a simplified depiction of this process.

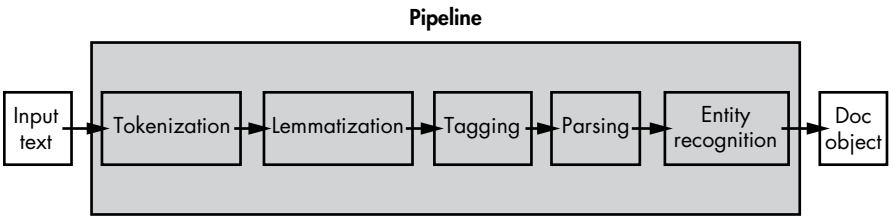

Figure 2-1: A high-level view of the processing pipeline

The processing pipeline typically includes tokenization, lemmatization, part-of-speech tagging, syntactic dependency parsing, and named entity recognition. We'll introduce each of these tasks in this section.

Tokenization

The very first action any NLP application typically performs on a text is parsing that text into *tokens*, which can be words, numbers, or punctuation marks. Tokenization is the first operation because all the other operations require you to have tokens already in place.

The following code shows the tokenization process:

```
❶ import spacy
❷ nlp = spacy.load('en')
❸ doc = nlp(u'I am flying to Frisco')
❹ print([w.text for w in doc])
```

We start by importing the spaCy library ❶ to gain access to its functionality. Then, we load a model package using the `en` shortcut link ❷ to create an instance of spaCy's Language class. A Language object contains the language's vocabulary and other data from the statistical model. We call the Language object `nlp`.

Next, we apply the object just created ❸ to a sample sentence, creating a *Doc object* instance. A Doc object is a container for a sequence of Token objects. spaCy generates it implicitly based on the text you provide it.

At this point, with just three lines of code, spaCy has generated the grammatical structure for the sample sentence. How you'll use it is entirely up to you. In this very simple example, you just print out the *text content* of each token from the sample sentence ❹.

The script outputs the sample sentence's tokens as a list:

```
['I', 'am', 'flying', 'to', 'Frisco']
```

The *text content*—the group of characters that compose the token, such as the letters "a" and "m" in the token "am"—is just one of many properties of a Token object. You can also extract various linguistic features assigned to a token, as you'll see in the following examples.

Lemmatization

A *lemma* is the base form of a token. You can think of it as the form in which the token would appear if it were listed in a dictionary. For example, the lemma for the token "flying" is "fly." *Lemmatization* is the process of reducing word forms to their lemma. The following script provides a simple example of how to do lemmatization with spaCy:

```
import spacy
nlp = spacy.load('en')
```

```
doc = nlp(u'this product integrates both libraries for downloading and
applying patches')
for token in doc:
  print(❶token.text, ❷token.lemma_)
```

The first three lines in the script are the same as those in the previous script. Recall that they import the spaCy library, load an English model using the `en` shortcut and create a text-processing pipeline, and apply the pipeline to a sample sentence—creating a Doc object through which you can access the grammatical structure of the sentence.

NOTE *In grammar, sentence structure is the arrangement of individual words, as well as phrases and clauses in a sentence. The grammatical meaning of a sentence depends on this structural organization.*

Once you have a Doc object containing the tokens from your example sentence, you iterate over those tokens in a loop, and then print out a token's text content ❶ along with its corresponding lemma ❷. This script produces the following output (I've tabulated it to make it more readable):

```
this        this
product     product
integrates  integrate
both        both
libraries   library
for         for
downloading download
and         and
applying    apply
patches     patch
```

The column on the left contains the tokens, and the column on the right contains their lemmas.

Applying Lemmatization for Meaning Recognition

Lemmatization is an important step in the task of meaning recognition. To see how, let's return to the sample sentence from the previous section:

```
I am flying to Frisco.
```

Suppose this sentence was submitted to an NLP application interacting with an online system that provides an API for booking tickets for trips. The application processes a customer's request, extracting necessary information from it and then passing on that information to the underlying API. This design might look like the one depicted in Figure 2-2.

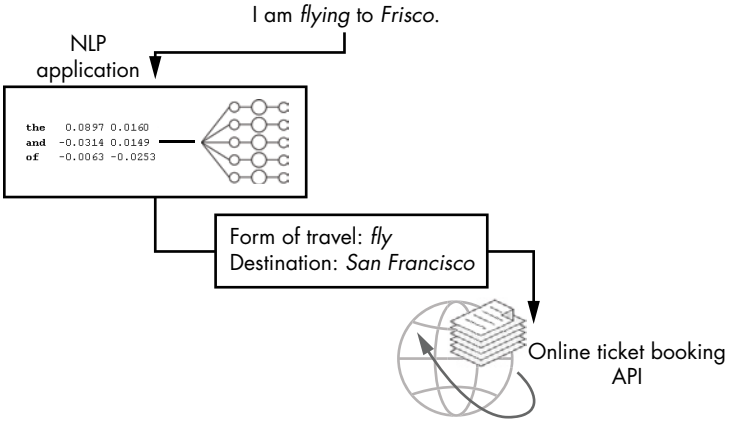

Figure 2-2: Using lemmatization in the process of extracting necessary information from a customer's request

The NLP application tries to get the following information from a customer's request: a form of travel (plane, rail, bus, and so on) and a destination. The application needs to first determine whether the customer wants an air ticket, a railway ticket, or a bus ticket. To determine this, the application searches for a word that matches one of the keywords in the predefined list. An easy way to simplify the search for these keywords is to first convert all the words in a sentence being processed to their lemmas. In that case, the predefined list of keywords will be much shorter and clearer. For example, you won't need to include all the word forms of the word fly (such as "fly," "flying," "flew," and "flown") to serve as an indicator that the customer wants an air ticket, reducing all possible variants to the base form of the word—that is, "fly."

Lemmatization also comes in handy when the application tries to determine a destination from a submitted request. There are a lot of nicknames for the globe's cities. But the system that books the tickets requires official names. Of course, the default Tokenizer that performs lemmatization won't know the difference between nicknames and official names for cities, countries, and so on. To solve this problem, you can add special case rules to an existing Tokenizer instance.

The following script illustrates how you might implement lemmatization for the destination cities example. It prints out the lemmas of the words composing the sentence.

```
import spacy
from spacy.symbols import ORTH, LEMMA
nlp = spacy.load('en')
doc = nlp(u'I am flying to Frisco')
print([w.text for w in doc])
❶ special_case = [{ORTH: u'Frisco', LEMMA: u'San Francisco'}]
❷ nlp.tokenizer.add_special_case(u'Frisco', special_case)
❸ print([w.lemma_ for w in nlp(u'I am flying to Frisco')])
```

You define a *special case* for the word `Frisco` ❶ by replacing its default lemma with `San Francisco`. Then you add this special case to the Tokenizer instance ❷. Once added, the Tokenizer instance will use this special case each time it's asked for the lemma of `Frisco`. To make sure that everything works as expected, you print out the lemmas of the words in the sentence ❸.

The script generates the following output:

```
['I', 'am', 'flying', 'to', 'Frisco']
['-PRON-', 'be', 'fly', 'to', 'San Francisco']
```

The output lists the lemmas for all words occurring in the sentence with the exception of `Frisco`, for which it lists `San Francisco`.

Part-of-Speech Tagging

A *part-of-speech tag* tells you the part-of-speech (noun, verb, and so on) of a given word in a given sentence. (Recall from Chapter 1 that a word can act as more than one part of speech depending on the context in which it appears.)

In spaCy, part-of-speech tags can include detailed information about a token. In the case of verbs, they might tell you the following features: tense (past, present, or future), aspect (simple, progressive, or perfect), person (1st, 2nd, or 3rd), and number (singular or plural).

Extracting these verb part-of-speech tags can help identify a user's intent when tokenization and lemmatization alone aren't sufficient. For instance, the lemmatization script for the ticket booking application in the preceding section won't decide how the NLP application chooses words in a sentence to compose a request to the underlying API. In a real situation, doing so might be quite complicated. For example, a customer's request might consist of more than one sentence:

```
I have flown to LA. Now I am flying to Frisco.
```

For these sentences, the results of lemmatization would be as follows:

```
['-PRON-', 'have', 'fly', 'to', 'LA', '.', 'now', '-PRON-', 'be', 'fly', 'to',
'San Francisco', '.']
```

Performing lemmatization alone isn't enough here; the application might consider the lemmas "fly" and "LA" from the first sentence as the keywords, indicating that the customer intends to fly to LA when in fact the customer intends to fly to San Francisco. Part of the problem is that lemmatization changes verbs to their infinitive forms, making it hard to know the role they play in a sentence.

This is where part-of-speech tags come into play. In English, the core parts of speech include noun, pronoun, determiner, adjective, verb, adverb, preposition, conjunction, and interjection. (See the linguistic primer in the Appendix for more information about these parts of speech.) In spaCy, these same categories—plus some additional ones for symbols, punctuation

marks, and others—are called *coarse-grained parts of speech* and are available as a fixed set of tags through the `Token.pos` (int) and `Token.pos_` (unicode) attributes.

Also, spaCy offers *fine-grained parts of speech* tags that provide more detailed information about a token, covering morphological features, such as verb tenses and types of pronouns. Naturally, the list of fine-grained parts of speech contains many more tags than the coarse-grained list. The fine-grained part-of-speech tags are available as the `Token.tag` (int) and `Token.tag_` (unicode) attributes.

Table 2-1 lists some of the common part-of-speech tags used in spaCy for English models.

Table 2-1: Some Common spaCy Part-of-Speech Tags

TAG (fine-grained part of speech)	POS (coarse-grained part of speech)	Morphology	Description
`NN`	NOUN	Number=sing	Noun, singular
`NNS`	NOUN	Number=plur	Noun, plural
`PRP`	PRON	PronType=prs	Pronoun, personal
`PRP$`	PRON	PronType=prs Poss=yes	Pronoun, possessive
`VB`	VERB	VerbForm=inf	Verb, base form
`VBD`	VERB	VerbForm=fin Tense=past	Verb, past tense
`VBG`	VERB	VerbForm=part Tense=pres Aspect=prog	Verb, gerund, or present participle
`JJ`	ADJ	Degree=pos	Adjective

NOTE *You can find the entire list of the fine-grained part-of-speech tags used in spaCy in the "Part-of-Speech Tagging" section in the Annotation Specifications manual at* https://spacy.io/api/annotation#pos-tagging.

Tense and aspect are perhaps the most interesting properties of verbs for NLP applications. Together, they indicate a verb's reference to a position in time. For example, we use the *present tense progressive aspect* form of a verb to describe what is happening right now or what will happen in the near future. To form the present tense progressive aspect verb, you add the present tense form of the verb "to be" before an -ing verb. For example, in the sentence "I am looking into it," you add "am"—the form of the verb "to be" in the first person, present tense—before the -ing verb "looking." In this example, "am" indicates the present tense and "looking" points to the progressive aspect.

Using Part-of-Speech Tags to Find Relevant Verbs

The ticket booking application could use the fine-grained part-of speech tags available in spaCy to filter the verbs in the discourse, choosing only those that could be key to determining the customer's intent.

Before moving onto the code for this process, let's try to figure out what kind of utterances a customer might use to express their intention to book a plane ticket to, say, LA. We could start by looking at some sentences that contain the following combination of lemmas: "fly", "to", and "LA". Here are some simple options:

```
I flew to LA.
I have flown to LA.
I need to fly to LA.
I am flying to LA.
I will fly to LA.
```

Notice that although all of these sentences would include the "fly to LA" combination if reduced to lemmas, only some of them imply the customer's intent to book a plane ticket to LA. The first two definitely aren't suitable.

A quick analysis reveals that the past and past perfect forms of the verb "fly"—the tenses used in the first two sentences—don't imply the intent we're looking for. Only the infinitive and present progressive forms are suitable. The following script illustrates how to find those forms in the sample discourse:

```
import spacy
nlp = spacy.load('en')
doc = nlp(u'I have flown to LA. Now I am flying to Frisco.')
print([w.text for w in doc if ❶w.tag_== ❷'VBG' or w.tag_== ❸'VB'])
```

The `tag_` property ❶ of a Token object contains the fine-grained part-of-speech attribute assigned to that object. You use a loop performed over the tokens composing the discourse to check whether the fine-grained part-of-speech tag assigned to a token is `VB` (a verb in the base, or infinitive, form) ❸ or `VBG` (a verb in the present progressive form) ❷.

In the sample discourse, only the verb "flying" in the second sentence meets the specified condition. So you should see the following output:

```
['flying']
```

Of course, fine-grained part-of-speech tags aren't only assigned to verbs; they're also assigned to the other parts of speech in a sentence. For example, spaCy would recognize LA and Frisco as proper nouns—nouns that are the names of individuals, places, objects, or organizations—and tag them with `PROPN`. If you wanted, you could add the following line of code to the previous script:

```
print([w.text for w in doc if w.pos_ == 'PROPN'])
```

Adding that code should output the following list:

```
['LA', 'Frisco']
```

The proper nouns from both sentences of the sample discourse are in the list.

Context Is Important

Fine-grained part-of-speech tags might not always be enough to determine an utterance's meaning. For this, you might still need to rely on context. As an example, consider the following utterance: "I am flying to LA." The part-of-speech tagger will assign the `VBG` tag to the verb "flying" in this example, because it's in the present progressive form. But because we use this verb form to describe either what is happening right now or what will happen in the near future, the utterance might mean either "I'm already in the sky, flying to LA." or "I'm going to fly to LA." When submitted to the ticket booking NLP application, the application should interpret only one of these sentences as "I need an air ticket to LA." Similarly, consider the following discourse: "I am flying to LA. In the evening, I have to be back in Frisco." This most likely implies that the speaker wants an air ticket from LA to Frisco for an evening flight. You'll find more examples about recognizing meaning based on context in "Using Context to Improve the Ticket-Booking Chatbot" on page 91.

Syntactic Relations

Now let's combine the proper nouns with the verb that the part-of-speech tagger selected earlier. Recall that the list of verbs you could potentially use to identify the intent of the discourse contains only the verb "flying" in the second sentence. How can you get the verb/proper noun pair that best describes the intent behind the discourse? A human would obviously compose the verb/proper noun pairs from words found in the same sentence. Because the verb "flown" in the first sentence doesn't meet the condition specified (remember that only infinitive and present progressive forms meet the condition), you'd be able to compose such a pair for the second sentence only: "flying, Frisco."

To handle these situations programmatically, spaCy features a *syntactic dependency* parser that discovers syntactic relations between individual tokens in a sentence and connects syntactically related pairs of words with a single arc.

Like lemmas and part-of-speech tags discussed in the previous sections, *syntactic dependency labels* are linguistic features that spaCy assigns to the Token objects that make up a text contained in a Doc object. For example, the dependency label `dobj` stands for "direct object." We could illustrate the syntactic relation it represents as an arrow arc, as shown in Figure 2-3.

HEAD AND CHILD

A syntactic dependency label describes the type of syntactic relation between two words in a sentence. In such a pair, one word is the syntactic governor (also called the head or parent) and the other is the dependent (also called the child). spaCy assigns a syntactic dependency label to the pair's dependent. For example, in the pair "need, ticket," extracted from the sentence "I need a plane ticket," the word "ticket" is the child and word "need" is the head, because "need" is the verb in what's called a verb phrase. In this same sentence, "a plane ticket" is a noun phrase: the noun "ticket" is the head, and "a" and "plane" are its children. To learn more, consult the linguistic primer in "Dependency Grammars vs. Phrase Structure Grammars" on page 185.

Each word in a sentence has exactly one head. Consequently, a word can be a child only to one head. The opposite is not always the case. The same word can act as a head in none, one, or several pairs. The latter means that the head has several children. This explains why a dependency label is always assigned to the child.

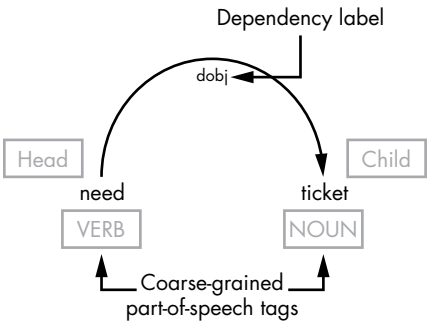

Figure 2-3: A graphical representation of a syntactic dependency arc

The `dobj` label is assigned to the word "ticket" because it's the child of the relation. A dependency label is always assigned to the child. In your script, you can determine the head of a relation using the `Token.head` attribute.

You might also want to look at the other head/child relations in the sentence, like the ones shown in Figure 2-4.

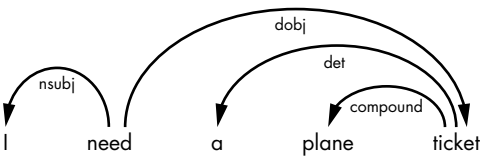

Figure 2-4: Head/child relations in an entire sentence

As you can see, the same word in a sentence can participate in several syntactic relations. Table 2-2 lists some of the most commonly used English dependency labels.

Table 2-2: Some Common Dependency Labels

Dependency label	Description
`acomp`	Adjectival complement
`amod`	Adjectival modifier
`aux`	Auxiliary
`compound`	Compound
`dative`	Dative
`det`	Determiner
`dobj`	Direct object
`nsubj`	Nominal subject
`pobj`	Object of preposition
`ROOT`	Root

The `ROOT` label marks the token whose head is itself. Typically, spaCy assigns it to the main verb of the sentence (the verb that is at the heart of the predicate). Every complete sentence should have a verb with the `ROOT` tag and a subject with the `nsubj` tag. The other elements are optional.

NOTE *Most of the examples in this book will assume that the submitted text is a complete sentence and use the `ROOT` tag to locate the sentence's main verb. Keep in mind that this won't work for every possible input.*

The following script illustrates how to access the syntactic dependency labels of the tokens in the discourse from the example in the "Part-of-Speech Tagging" on page 21:

```
import spacy
nlp = spacy.load('en')
doc = nlp(u'I have flown to LA. Now I am flying to Frisco.')
for token in doc:
  print(token.text, ❶token.pos_, ❷token.dep_)
```

The script outputs the coarse-grained part-of-speech tags ❶ (see Table 2-1) and dependency labels assigned to the tokens ❷ composing the sample discourse:

```
I      PRON  nsubj
have   VERB  aux
flown  VERB  ROOT
to     ADP   prep
```

```
LA     PROPN pobj
.      PUNCT punct
Now    ADV   advmod
I      PRON  nsubj
am     VERB  aux
flying VERB  ROOT
to     ADP   prep
Frisco PROPN pobj
.      PUNCT punct
```

But what it doesn't show you is how words are related to each other in a sentence by means of the commonly called *dependency arcs* explained at the beginning of this section. To look at the dependency arcs in the sample discourse, replace the loop in the preceding script with the following one:

```
for token in doc:
  print(❶token.head.text, token.dep_, token.text)
```

The head property of a token object ❶ refers to the syntactic head of this token. When you print this line, you'll see how words in the discourse sentences are connected to each other by syntactic dependencies. If they were presented graphically, you would see an arc for each line in the following output, except for the `ROOT` relation. The reason is that the word to which this label is assigned is the only word in a sentence that doesn't have a head:

```
flown   nsubj  I
flown   aux    have
flown   ROOT   flown
flown   prep   to
to      pobj   LA
flown   punct  .
flying  advmod Now
flying  nsubj  I
flying  aux    am
flying  ROOT   flying
flying  prep   to
to      pobj   Frisco
flying  punct  .
```

Looking at the earlier list of syntactic dependencies, let's try to figure out what labels point to the tokens that could potentially best describe the customer's intent: in other words, you need to find a pair that would alone appropriately describe the customer's intent.

You might be interested in the tokens marked with the `ROOT` and `pobj` dependency labels, because in this example they're key in intent recognition. As stated earlier, the `ROOT` label marks the main verb of the sentence, and `pobj`, in this example, marks the entity that—in conjunction with the verb—summarizes the meaning of the entire utterance.

The following script locates words that are assigned those two dependency labels:

```
import spacy
nlp = spacy.load('en')
doc = nlp(u'I have flown to LA. Now I am flying to Frisco.')
❶ for sent in doc.sents:
❷  print([w.text for w in sent ❸if w.dep_ == 'ROOT' or w.dep_ == 'pobj'])
```

In this script, you *shred the discourse* ❶ to separate the sentences with the `doc.sents` property, which iterates over the sentences in the document. Shredding a text into separate sentences can be useful when you need to find, for example, certain parts of speech in each sentence of the discourse. (We'll discuss `doc.sents` in the next chapter, where you'll see an example of how to refer to the tokens in a document with sentence-level indices.) This allows you to create a list of potential keywords for each sentence based on specific dependency labels assigned to the tokens ❷. The filter conditions used in this example are chosen based on the examination of the syntactically related pairs generated by the previous script. In particular, you pick up the tokens with `ROOT` and `pobj` dependency labels ❸, because these tokens form the pairs you're interested in.

The script's output should look as follows:

```
['flown', 'LA']
['flying', 'Frisco']
```

In both sentence pairs, the output nouns are the ones labeled as `pobj`. You could use this in your ticket booking application to choose the noun that best belongs with the verb. In this case, that would be "flying," which goes with "Frisco."

This is a simplified example of information extraction using dependency labels. In the following chapters, you'll be given more sophisticated examples of how to iterate over the dependency tree of a sentence or even an entire discourse, extracting necessary pieces of information.

Try It Yourself

Now that you know how to take advantage of lemmatization, part-of-speech tags, and syntactic dependency labels, you can put them all together to do something useful. Try combining the examples from the preceding sections into a single script that correctly identifies a speaker's intent to fly to San Francisco.

Your script should generate the following output:

```
['fly', 'San Francisco']
```

To achieve this, start with the latest script from this section and enhance the conditional clause in the loop, adding the conditions to

account for fine-grained part-of-speech tags, as discussed in "Part-of-Speech Tagging" on page 21. Then add the lemmatization functionality to your script, as discussed in "Lemmatization" on page 18.

Named Entity Recognition

A *named entity* is a real object that you can refer to by a proper name. It can be a person, organization, location, or other entity. Named entities are important in NLP because they reveal the place or organization the user is talking about. The following script finds named entities in the sample discourse used in the previous examples:

```
import spacy
nlp = spacy.load('en')
doc = nlp(u'I have flown to LA. Now I am flying to Frisco.')
for token in doc:
❶ if token.ent_type != 0:
    print(token.text, ❷token.ent_type_)
```

If the `ent_type` attribute of a token is not set to 0 ❶, then the token is a named entity. If so, you print the `ent_type_` attribute of a token ❷, which contains the type of named entity in unicode. As a result, the script should output the following:

```
LA      GPE
Frisco  GPE
```

Both `LA` and `Frisco` are marked as `GPE`, the acronym for "geopolitical entity" and includes countries, cities, states, and other place names.

Summary

In this chapter, you set up a working environment for using spaCy. Then you learned simple scripts that illustrate how to use spaCy's features to perform the basic NLP operations for extracting important information. These operations included tokenization, lemmatization, and identifying syntactic relations between individual tokens in a sentence. The examples provided in this chapter are simplified and don't reflect real-world scenarios. To write a more sophisticated script using spaCy, you'll need to implement an algorithm to derive the necessary tokens from a dependency tree, using the linguistic features assigned to tokens. We'll return to extracting and using linguistic features in chapter 4, and we'll cover dependency trees in detail in chapter 6.

In the next chapter, you'll look at the key objects of spaCy's API, including containers and processing pipeline components. Also, you'll learn to use spaCy's C-level data structures and interfaces to create Python modules capable of processing large amounts of text.