# 12

# IMPLEMENTING WEB DATA AND PROCESSING IMAGES

Real-life chatbots should respond to a variety of inputs, such as questions from users on unfamiliar topics or even images sent through messaging apps. For example, chatbot app users can send not only text messages, but also photos, and the bot is supposed to react appropriately to both.

This chapter provides some examples of how to use other libraries from Python's AI ecosystem when developing a bot application. First, you'll combine spaCy with Wikipedia to find information about keywords taken from a user's question. Next, you'll obtain descriptive tags for a submitted image with the help of Clarifai, an image and video recognition tool, so your app can interpret visual content.

Then you'll put all the components together to build a Telegram bot that can generate relevant responses to text and images by extracting information from Wikipedia.

## How It Works

Figure 12-1 shows a diagram of the bot we'll build in this chapter. The bot is designed to understand text messages and pictures, and respond with text from Wikipedia.

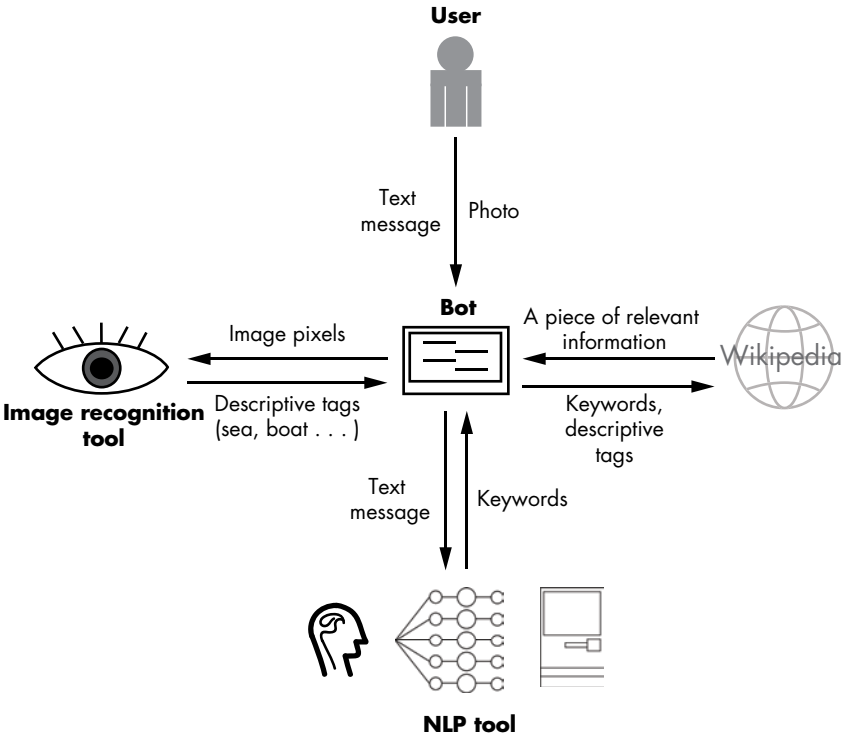


*Figure 12-1: How a bot that can process text messages and pictures works*

Using this bot, the user can post either a text message or a picture. If the post is a picture, the bot sends it to an image recognition tool for processing. This tool returns a verbal description of the picture in the form of descriptive tags. If the post is a text message, the bot uses an NLP tool like spaCy to extract a keyword or a keyphrase from it. The bot then uses either the tag or the keyphrase to find the most relevant content on Wikipedia (or somewhere else on the web) and return a piece of it to the user. You can use this scenario in chatbots you design to hold a conversation on various topics for fun, learning, or personal use.

## Making Your Bot Find Answers to Questions from Wikipedia

Let's start with a discussion of techniques that you can implement in your bot to make it interpret a wide range of text messages. Previous chapters talked about how bots used for business purposes typically ask for certain

information from the user and then use the answer to fill an order or booking request. In contrast, a bot designed to hold informal conversations should be able to answer a diverse range of user questions.

One way to help a chatbot answer user questions is to choose a keyword or keyphrase in the question that provides a clue as to what information should be included in the answer. Once you have this keyword or keyphrase, you can use it to search for the answer using tools like the Wikipedia API for Python. Wikipedia's API lets you access and parse Wikipedia content programmatically, performing a search for a keyword to retrieve content from the most relevant Wikipedia article. The following sections describe how to do this.

But before proceeding to the examples, make sure you're using one of the most recent spaCy models, because the accuracy of the dependency parsing is higher in newer versions. You can check the version of your current model using the following command:

```
nlp.meta['version']
```

Then visit the *https://explosion.ai/demos/displacy/* demo page (discussed in Chapter 7) to see the latest stable versions of spaCy models available. Alternatively, you can visit spaCy's documentation at *https://spacy.io/usage/* to check for the newest version of spaCy. Both spaCy and its models follow the same versioning scheme. Based on that information, you might want to update the model you currently use. Refer to Chapter 2 for details on how to download and install a spaCy model.

### Determining What the Question Is About

Some words in a question are more important than others when you're trying to determine what the speaker is asking about. Sometimes it's enough to look at a single word in the question, such as the noun that follows a preposition. For example, a user might use any of the following questions to ask the bot to find some information about rhinos:

```
Have you heard of rhinos? Are you familiar with rhinos? What could you tell me about rhinos?
```

Let's look at what the dependency parsing of such sentences might look like. Figure 12-2 shows a graphical representation of the parsing of the first sentence.

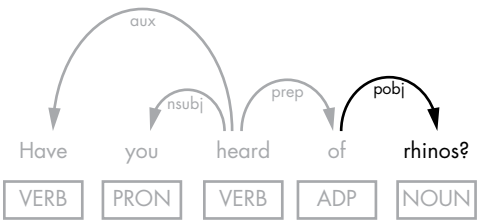


*Figure 12-2: The dependency parsing of a sentence containing an object of a preposition*

The parsing illustrates that in this kind of question you can get the word "rhinos" by extracting the object of the preposition. "Rhinos" would be the most helpful word in the question for finding an answer. The following code fragment shows how you might extract the first occurrence of an object of the preposition in the question:

```
doc = nlp(u"Have you heard of rhinos?")
for t in doc:
  if t.dep_ == 'pobj' and (t.pos_ == 'NOUN' or t.pos_ == 'PROPN'):
    phrase = (' '.join([child.text for child in t.lefts]) + ' ' + t.text).lstrip()
    break
```

In the code, we also pick up the left children of the object of the preposition, because the object might have important modifiers, as in the following example: "What can you say about wild mountain goats?" When given this question, the code should assign "wild mountain goats" to the `phrase` variable.

Notice the use of the `break` statement at the end, which guarantees that only the first object of a preposition in a sentence will be picked up. For example, in the sentence, "Tell me about the United States of America," the phrase "the United States" would be picked up, but not "America."

But this is not always desirable behavior. What if a user asked, "Tell me about the color of the sky."? This is where we need to apply more complicated logic. In particular, we might want to to pick up any prepositional object that follows the first prepositional object, provided the latter is dependent on the former.

Here is how you might implement this logic:

```
doc = nlp(u"Tell me about the color of the sky.")
for t in doc:
  if t.dep_ == 'pobj' and (t.pos_ == 'NOUN' or t.pos_ == 'PROPN'):
    phrase = (' '.join([child.text for child in t.lefts]) + ' ' + t.text).lstrip()
    if bool([prep for prep in t.rights if prep.dep_ == 'prep']):
      prep = list(t.rights)[0]
      pobj = list(prep.children)[0]
      phrase = phrase + ' ' + prep.text + ' ' + pobj.text
    break
...
```

Note that this code will process a prepositional object that is a dependent of the first prepositional object only if the former exists in the sentence. Otherwise, this code will work the same as the code shown previously.

Now let's look at another type of question in the following examples where two words, a verb and its subject, provide the best information about what a user wants in response to the questions:

```
Do you know what an elephant eats? Tell me how dolphins sleep. What is an API?
```

Figure 12-3 shows what a dependency parsing for one of these sentences might look like.

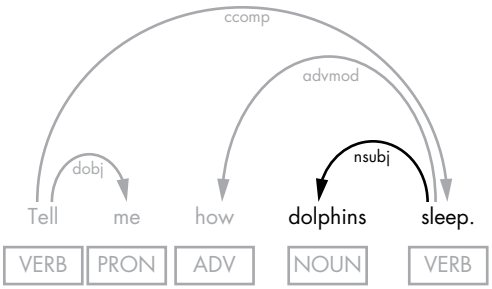


*Figure 12-3: The dependency parsing of a sentence in which a subject/verb pair is the most informative element for discovering what the speaker wants to know*

Looking through the parsing shown in the figure, notice that the subject/verb pair that occurs at the end of the sentence is the most informative when trying to determine what the speaker asks about. Programmatically, you can extract the subject and verb pair from a sentence using the following code:

```
doc = nlp(u"Do you know what an elephant eats?")
for t in reversed(doc):
  if t.dep_ == 'nsubj' and (t.pos_ == 'NOUN' or t.pos_ == 'PROPN'):
    phrase = t.text + ' ' + t.head.text
    break
```

While examining this code, notice that we loop backward from the end of the sentence using Python's `reversed()` function. The reason is that we need to pick up the last subject/verb pair in the sentence, as in this example: "Do you know what an elephant eats?" In this sentence, we're interested in the phrase "elephant eats" rather than "you know," which is also a subject/verb pair.

Additionally, in some questions, the last noun in the sentence is the direct object of a verb that matters to determine what the question is about, as in the following example:

```
How to feed a cat?
```

In this sentence, extracting the direct object "cat" wouldn't be sufficient, because we also need the word "feed" to understand the question. Ideally, we'd generate the keyphrase "feeding a cat." That is, we'd replace the infinitive "to" form of the verb with a gerund by adding "-ing," optimizing the keyphrase for an internet search. Figure 12-4 shows the dependency parsing for this sentence.

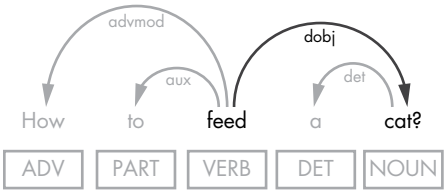


*Figure 12-4: Dependency parsing of a sentence with a verb/direct object pair as the most informative phrase*

This syntactic parsing shows that extracting the required phrase is easy, because the direct object and its transitive verb are connected with a direct link.

The code implementation for the extraction discussed here might look like this:

```
doc = nlp(u"How to feed a cat?")
for t in reversed(doc):
  if t.dep_ == 'dobj' and (t.pos_ == 'NOUN' or t.pos_ == 'PROPN'):
    phrase = t.head.lemma_ + 'ing' + ' ' + t.text
    break
```

In this case, we once again loop backward from the end of the sentence. To understand why, consider the following sentence: "Tell me something about how to feed a cat." It contains two verb/direct object pairs, but we're interested only in the one that occurs at the end of the sentence.

### *Try It Yourself*

Modify the code in the previous section that extracts the phrase "elephant eats" so the keyphrase being extracted from the sentence includes possible modifiers of the subject, excluding a possible determiner. For example, in the sentence, "Tell me how a female cheetah hunts," your script should return "female cheetah hunts" and remove the "a" determiner from the noun chunk. As an example of how you might implement this, look at the code that follows Figure 12-2. In that code, you picked up modifiers for the object of the preposition being extracted.

Also, add a check to see whether the verb included in the phrase being extracted has a direct object, and if so, append the direct object to the keyphrase. For example, the question "Do you know how many eggs a sea turtle lays?" should give you the following keyphrase: "sea turtle lays eggs."

### *Using Wikipedia to Answer User Questions*

Now that you have a keyphrase that can help you find the information needed to generate a relevant response to the user's question, you need to retrieve the information. A bot can get answers to user questions from several places, and the proper source to use depends on the application, but

Wikipedia is a good place to start. The `wikipedia` Python library (*https://pypi.org/project/wikipedia/*) allows you to access Wikipedia articles from within your Python code.

You can install the library via `pip` as follows:

```
pip install wikipedia
```

To test the newly installed library, use the following script, which relies on a code fragment from the previous section to extract a keyword from a submitted sentence. Then it uses that keyword as a Wikipedia search term.

```
import spacy
import wikipedia
nlp = spacy.load('en')
doc = nlp(u"What do you know about rhinos?")
for t in doc:
  if t.dep_ == 'pobj' and (t.pos_ == 'NOUN' or t.pos_ == 'PROPN'):
  ❶ phrase = (' '.join([child.text for child in t.lefts]) + ' ' + t.text).
    lstrip()
    break
❷ wiki_resp = wikipedia.page(phrase)
print("Article title: ", wiki_resp.title)
print("Article url: ", wiki_resp.url)
print("Article summary: ", wikipedia.summary(phrase, sentences=1))
```

In this script, we extract a keyword or keyphrase from the submitted sentence ❶ and send it to the `wikipedia.page()` function, which returns the most relevant article for the given keyword ❷. Then we simply print out the article's title, URL, and first sentence.

The output this script generates should look like this:

```
Article title:  Rhinoceros
Article url:  https://en.wikipedia.org/wiki/Rhinoceros
Article summary:  A rhinoceros (, from Greek  rhinokerōs, meaning 'nose-horned', from  rhis,
meaning 'nose', and  keras, meaning 'horn'), commonly abbreviated to rhino, is one of ...
```

### *Try It Yourself*

Enhance the script in the previous section so it can "see" the children of the first prepositional object and the dependent prepositional objects. For example, in the question, "Have you heard of fried eggs with yellow tomatoes?" it should extract the keyphrase "fried eggs with yellow tomatoes."

## Reacting to Images Sent in a Chat

In addition to text messages, users of messenger apps often post images. Other people usually respond to these with comments about what is shown in the picture. For example, a user posts a photo of grapes, to which another user leaves the following comment: "I love fruit. It contains lots of fiber and

vitamins." How can you teach a bot to do the same? One way is to generate descriptive tags for an image that the bot can use in processing. This is where you need an image recognition tool, like Clarifai, which provides built-in models trained with photos from different domains, such as apparel, travel, or celebrities.

Clarifai allows a bot to obtain a set of categories for a submitted photo, making it possible for the bot to guess what is depicted in the image. You can get useful categories for a photo in two steps. First, you use Clarifai's general image recognition model to obtain descriptive tags (objects with probabilities) that can give you a general idea of what is shown in the photo. For example, the presence of the "no person" tag indicates that no people are in the photo.

Second, after examining the tags, you can apply more specific models to the same photo, such as Clarifai's Food or Apparel models. Both are trained to recognize food and fashion-related items, respectively. This time, you'll obtain another, more granular set of tags to give you a better idea of the contents of the photo. For the entire list of Clarifai's image recognition models, visit its Models page at *https://www.clarifai.com/models/*.

### *Generating Descriptive Tags for Images Using Clarifai*

Clarifai offers a Python client to interact with its recognition API. You can install the latest stable package using `pip`:

```
pip install clarifai --upgrade
```

Before you can start using the Clarifai library, you must obtain an API key by creating an account and then clicking the **GET API KEY** button at *https://www.clarifai.com/*.

Once you have the key, you can test the Clarifai library. The following simple script passes an image to a Clarifai model and prints a list of tags expressing possible categories for the image:

```
from clarifai.rest import ClarifaiApp, client, Image
app = ClarifaiApp(api_key='YOUR_API_KEY')
❶ model = app.public_models.general_model
filename = '/your_path/grape.jpg'
❷ image = Image(file_obj=open(filename, 'rb'))
response = model.predict([image])
❸ concepts = response['outputs'][0]['data']['concepts']
for concept in concepts:
  print(concept['name'], concept['value'])
```

In this example, we call Clarifai's Predict API with the general model ❶. Clarifai takes only the pixels as input, so make sure you're opening an image file in `'rb'` mode ❷, which opens the file in binary format for reading. The

Predict API generates a list of descriptive tags, such as fruit, grape, health, and so on ❸, for the submitted photo, allowing the code to "understand" what it shows.

The *grape.jpg* file used in this example contains the photo shown in Figure 12-5.

*Figure 12-5: The photo submitted to Clarifai in the preceding script*

The list of concepts that the script generates for the photo should look as follows:

```
no person 0.9968359470367432
wine 0.9812138080596924
fruit 0.9805494546890259
juicy 0.9788177013397217
health 0.9755384922027588
grow 0.9669009447097778
grape 0.9660607576370239
...
```

Each entry represents a category and the probability that the image fits within the category. Thus, the first tag in the list tells us that the submitted photo contains no person with a probability of 0.99. Note that not all the tags will provide a direct description of the depicted content. For example,

the tag "wine" is included here, perhaps because wine is made from grapes. The presence of indirect tags in the list gives your bot more options to interpret the image.

### *Using Tags to Generate Text Responses to Images*

Now that you know how to obtain descriptive tags for an image, how can you use these tags to respond to the image? Or how can you choose the most important tags from the generated list? Think about the following general considerations:

- You might want to take into account only the tags with high likelihoods. For that, you can choose a threshold of likelihood for the tags. For example, consider only the top five or 10 tags.
- You might choose only those tags that are in the context of the current chat. Chapter 11 showed an example of how to maintain the context of the current chat in a Telegram bot using the `context.user_data` dictionary.
- You might iterate over the generated tags, searching for a particular tag. For example, you might search for the tag "fruit" or "health" to determine whether you should continue the conversation on this topic.

  The bot discussed in the next section will implement the third option.

## Putting All the Pieces Together in a Telegram Bot

In the rest of this chapter, we'll build a Telegram chatbot that uses the Wikipedia API and the Clarifai API. This bot will respond intelligently to text and images of food. Refer back to Chapter 11 for details on how to create a new bot in Telegram.

### *Importing the Libraries*

The import section of the code must include all the libraries that we'll use in the bot's code. In this example, we include the libraries required to access the Telegram Bot API, Wikipedia API, Clarifai API, and spaCy.

```
import spacy
import wikipedia
from telegram.ext import Updater, CommandHandler, MessageHandler, Filters
from clarifai.rest import ClarifaiApp, Image
```

If you've followed the instructions provided in this chapter and Chapter 11, all of these libraries should be available on your system.

### *Writing the Helper Functions*

Next, we need to implement the helper functions that will be invoked from within the bot's callback functions. The `keyphrase()` function takes a

sentence as a Doc object and tries to extract the most informative word or a phrase from it, as discussed in the earlier section "Determining What the Question Is About" on page 171. The following implementation uses the code fragments you saw in that section, adjusting them so we can use them within a single function:

```
def keyphrase(doc):
  for t in doc:
    if t.dep_ == 'pobj' and (t.pos_ == 'NOUN' or t.pos_ == 'PROPN'):
      return (' '.join([child.text for child in t.lefts]) + ' ' + t.text).
      lstrip()
  for t in reversed(doc):
    if t.dep_ == 'nsubj' and (t.pos_ == 'NOUN' or t.pos_ == 'PROPN'):
      return t.text + ' ' + t.head.text
  for t in reversed(doc):
    if t.dep_ == 'dobj' and (t.pos_ == 'NOUN' or t.pos_ == 'PROPN'):
      return t.head.text + 'ing' + ' ' + t.text
  return False
```

Note that the conditions are arranged in order of priority in this code. Thus, if the object of the preposition is found, we extract it and quit without checking for the other conditions. Of course, some complicated questions might meet multiple conditions, but checking for this would complicate the function implementation.

Like the `keyphrase()` function, the `photo_tags()` function is supposed to determine the most descriptive word for a user's input. But unlike `keyphrase()`, it analyzes a photo. It performs the analysis with the help of Clarifai, which generates a set of descriptive tags for a submitted photo. This implementation uses only two Clarifai models: the general model and the food model.

```
def photo_tags(filename):
  app = ClarifaiApp(api_key=CLARIFAI_API_KEY)
  model = app.public_models.general_model
  image = Image(file_obj=open(filename, 'rb'))
  response = model.predict([image])
  concepts = response['outputs'][0]['data']['concepts']
  for concept in concepts:
    if concept['name'] == 'food':
      food_model = app.public_models.food_model
      result = food_model.predict([image])
      first_concept = result['outputs'][0]['data']['concepts'][0]['name']
      return first_concept
  return response['outputs'][0]['data']['concepts'][1]['name']
```

This code starts by applying the general model. If the tag `'food'` is found in the generated list, it applies the food model to obtain more descriptive tags for the food items shown in the image. This implementation will use the first tag only as the keyword for the search.

Now that we have the keyword or keyphrase, determined either in the `keyphrase()` function or in the `photo_tags()` function, we need to obtain a piece of information that is closely related to this keyword or keyphrase. The following `wiki()` function does the trick:

```
def wiki(concept):
  nlp = spacy.load('en')
  wiki_resp = wikipedia.page(concept)
  doc = nlp(wiki_resp.content)
  if len(concept.split()) == 1:
    for sent in doc.sents:
      for t in sent:
        if t.text == concept and t.dep_ == 'dobj':
          return sent.text
  return list(doc.sents)[0].text
```

The algorithm we use here searches for a sentence in the retrieved content that includes the keyword as the direct object.

But this simple implementation can intelligently process only a single-word input. When a word is submitted, the algorithm we use here just extracts the first sentence from the Wikipedia article found with the help of this word.

### *Writing the Callback and main() Functions*

Next, we add the bot's callback functions. The `start()` function simply sends a greeting to the user in response to the `/start` command.

```
def start(update, context):
    update.message.reply_text('Hi! This is a conversational bot. Ask me something.')
```

The `text_msg()` function is the callback for the bot's user text messages handler.

```
def text_msg(update, context):
  msg = update.message.text
  nlp = spacy.load('en')
  doc = nlp(msg)
  concept = keyphrase(doc)
  if concept != False:
    update.message.reply_text(wiki(concept))
  else:
    update.message.reply_text('Please rephrase your question.')
```

First, we apply spaCy's pipeline to the message, converting it to a Doc object. Then we send the Doc to the `keyphrase()` function discussed earlier to extract a keyword or keyphrase from the message. The returned keyword or keyphrase is then sent to the `wiki()` function to obtain a piece of relevant information, which should be a single sentence in this implementation.

The `photo()` function shown in the following code is the callback for the bot's handler for the photos submitted by the user:

```
def photo(update, context):
  photo_file = update.message.photo[-1].get_file()
  filename = '{}.jpg'.format(photo_file.file_id)
  photo_file.download(filename)
  concept = photo_tags(filename)
  update.message.reply_text(wiki(concept))
```

The function retrieves the submitted image as a file and sends it for further processing to the helper functions discussed earlier in the section "Writing the Helper Functions."

Finally, we add the `main()` function in which we register handlers for both text messages and photos.

```
def main():
    updater = Updater("YOUR_TOKEN", use_context=True)
    disp = updater.dispatcher
    disp.add_handler(CommandHandler("start", start))
    disp.add_handler(MessageHandler(Filters.text, text_msg))
    disp.add_handler(MessageHandler(Filters.photo, photo))
    updater.start_polling()
    updater.idle()
if __name__ == '__main__':
    main()
```

The `main()` function for this Telegram bot is quite concise. We create the `Updater` and pass the bot's token to it. Then we obtain the dispatcher to register handlers. In this example, we register just three handlers. The first one is the handler for the `/start` command. The second handles text messages coming from the user. The third one handles photos posted by the user. After registering handlers, we start the bot by invoking `updater.start_polling()` and then invoking `updater.idle()` to block the script to wait for a user message or an exit shortcut (CTRL-C).

### Testing the Bot

Now that we've created the bot, it's time to test it. You can test it either on a smartphone or a computer. On a smartphone, in the Telegram app search for your bot's name followed by the @ sign, and then enter the `/start` command to start a chat. On a computer, use Telegram Web at *https://web.telegram.org*.

After receiving a greeting from the bot, send it a simple request, such as "Tell me about fruit." The bot should respond with a single sentence that it extracts from a relevant Wikipedia article. For simplicity, choose a sentence that uses the direct object from the sentence ("fruit" in this example) as the keyword.

You can also submit a photo to check which comment the bot will give in response. Figure 12-6 illustrates a screenshot of such a test.

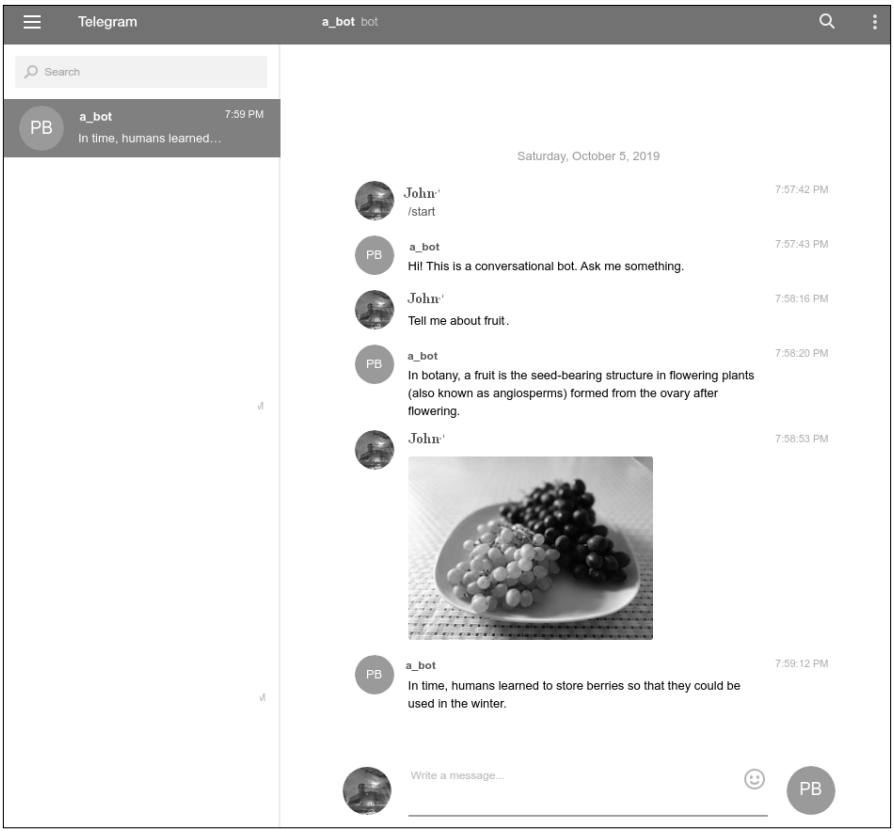


*Figure 12-6: A screenshot of the bot we created*

Remember that this implementation can properly process only photos of food.

## Try It Yourself

Note that the bot implementation provided in the preceding section can't generate smart responses to many different types of user input. The `wiki()` function we used can properly process only those requests for which `keyphrase()` returns a single word. It also works best if that keyword is a direct object. Also, the bot can only intelligently respond to images of food.

Enhance the `wiki()` function so it can process phrases instead of only one word, such as "dolphins sleep." Finding an appropriate sentence for such a phrase requires using dependency labels, because you'll need to find a subject/verb pair. In addition, you'll need to reduce the words to their lemmas. For example, "dolphins sleep" and "dolphin sleeps" should satisfy the search criteria.

You might also want to enhance the functionality of the `photo_tags()` function so it can process not only food photos, but also those that show something else—for example, apparel.

## Summary

In this chapter, you saw examples of how to use spaCy along with other libraries in Python's AI ecosystem to build an AI-powered application that can process data of different types. By using the Wikipedia and Clarifai Python APIs, we designed a chatbot that could react to images and pull text from Wikipedia, techniques that make the bot a smarter interlocutor.

After reading this book, you might want to expand and improve on what you've learned. The most natural way to enhance your knowledge is to continue to experiment with chatbots. Start by building a Telegram script with Python using the instructions provided in Chapter 11; next, enhance its functionality using instructions provided in this chapter. Then work on improving the algorithms you learned in this book to make them more suitable for your use cases.