5

SPEAKING APPLICATIONS

Now that you know how to make Python talk and listen, we'll create several real-world applications that utilize those skills. But before that, you'll create a local package. Since you'll use the *mysr* and *mysay* local modules in every chapter for the reminder of the book, you'll create a Python package to contain all local modules. This way, you don't need to copy and paste these modules to the folders of individual chapters. This also helps keep the code consistent throughout the book. You'll learn how a Python package works and how to create one yourself along the way.

In the first application, you'll build a Guess the Number game that takes voice commands and talks back to you in a human voice.

You'll then learn how to parse text to extract news summaries from National Public Radio (NPR) and have Python read them out to you. You'll also build a script to extract information from Wikipedia based on your voice inquiries and to speak the answers out.

Finally, you'll learn how to traverse files in a folder with your voice, with the aim of building your very own Alexa. You'll be able to say to the script, "Python, play Selena Gomez," and a song by Selena Gomez that's saved on your computer will start playing.

As usual, you can download all the code for all the scripts from *https://www.nostarch.com/make-python-talk/*. Before you begin, set up the folder */mpt/ch05/* for this chapter.

NEW SKILLS

- Learning how Python packages work
- Creating your self-made local Python package
- Parsing the source code of a news website to extract news summaries
- Extracting answers to your queries from Wikipedia and converting them to voice
- Traversing files in a folder on your computer by using the *os* module

Create Your Self-Made Local Python Package

In Chapter 3, you built a self-made local module *mysr* to contain all speech recognition–related code. Whenever you need to use the speech-recognition feature, you import `voice_to_text()` from the module. Similarly, you built a self-made local module *mysay* in Chapter 4 to contain all text-to-speech-related code. You import `print_say()` from the module whenever you use the text-to-speech feature.

You'll use these two self-made local modules in this chapter and other chapters in this book. To make these modules work, you need to put the module files (namely, *mysr.py* and *mysay.py*) in the same directory as the script that uses these two modules. This means you'd potentially have to copy and paste these files into the directory of almost every chapter in this book. You may wonder: is there a more efficient way to do this?

The answer is yes, and that's what Python packages are for.

Next, you'll first learn what a Python package is and how it works. You'll then learn how to create your self-made local package. Finally, you'll use a Python script to test and import your package.

What's a Python Package?

Many people think that Python modules and Python packages are the same. They're not.

A Python *module* is a single file with the *.py* extension. In contrast, a Python *package* is a collection of Python modules contained in a single directory. The directory must have a file named *__init__.py* to distinguish it from a directory that happens to have *.py* extension files in it.

I'll guide you through the process of creating a local package step-by-step.

Create Your Own Python Package

To create a local Python package, you need to create a separate directory for it and place all related files into it. In this section, you'll create a local package to contain both our speech recognition and text-to-speech module files—namely, *mysr.py* and *mysay.py*.

Create a Package Directory

First, you need to create a directory for the package.

In this book, you use a separate directory for each chapter. For example, all Python scripts and related files in this chapter are placed in the directory */mpt/ch05/*. Since you are creating a package to be used for all chapters in this book, you'll create a directory parallel to all chapters. Specifically, you'll use the directory */mpt/mptpkg/*, where *mptpkg* is the package name. The diagram in Figure 5-1 explains the position of the package relative to the book chapters.

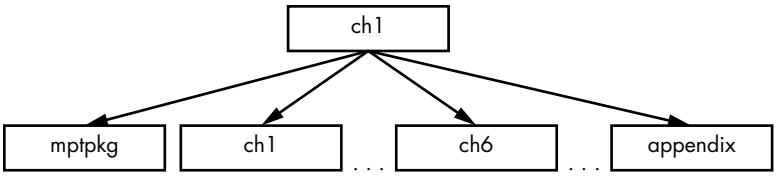

Figure 5-1: The position of the mptpkg *package relative to the chapter folders*

As you can see, the package directory is parallel to the chapter directories, which are all contained in the directory for the book, */mpt*, as in *Make Python Talk*.

Create Necessary Files for Your Package

Next, you need to create and place necessary files in the package.

First, copy and paste the two modules you created in Chapters 3 and 4, *mysr.py* and *mysay.py*, in the package directory */mpt/mptpkg/*. Do not make any changes to the two files.

Then save the following script, *__init__.py*, in the package directory */mpt/mptpkg/* (or you can download it from the book's resources):

```
from .mysr import voice_to_text
from .mysay import print_say
```

The purpose of this file is twofold: it imports `voice_to_text()` and `print_say()` so you can use those functions at the package level, and it also tells Python that the directory is a package, not a folder that happens to have Python scripts in it.

Finally, save the following script, *setup.py*, in the book directory */mpt*, one level above the package directory */mpt/mptpkg/*. The script is also available from the book's resources.

```
from setuptools import setup
setup(name='mptpkg',
version='0.1',
description='Install local package for Make Python Talk',
author='Mark Liu',
author_email='mark.liu@uky.edu',
packages=['mptpkg'],
zip_safe=False)
```

The file provides information about the package, such as the package name, author, version, descriptions, and so on.

You'll learn how to install this local package on your computer next.

Install Your Package

Because you'll modify the local package and add more features to it later in the book, it's better to install the package in editable mode.

Open your Anaconda prompt (Windows) or a terminal (Mac or Linux) and activate your virtual environment for this book, *chatting*. Run the following command:

```
pip install -e path-to-mpt
```

Replace *path-to-mpt* with the actual directory path of */mpt*. For example, the book directory */mpt* is *C:\mpt* on my office computer that runs the Windows operating system, so I installed the local package using this command:

```
pip install -e C:\mpt
```

On my Linux machine, the path to the */mpt* directory is */home/mark/Desktop/mpt*, so I installed the local package using this command:

```
pip install -e /home/mark/Desktop/mpt
```

The `-e` option tells the Python to install the package in editable mode so that you can modify the package anytime you need to.

With that, the local package is installed on your computer.

Test Your Package

Now that you have installed your self-made local package, you'll learn how to import it.

You'll write a Python script to test the package you just created.

Let's revisit the script *repeat_me1.py* from Chapter 4. Enter the following lines of code in your Spyder editor and save it as *repeat_me2.py* in your Chapter 5 directory */mpt/ch05/*:

```
# Import functions from the local package mptpkg
from mptpkg import voice_to_text
from mptpkg import print_say

while True:
    print('Python is listening...')
    inp = voice_to_text()
    if inp == "stop listening":
        print_say(f'you just said {inp}; goodbye!')
        break
    else:
        print_say(f'you just said {inp}')
        continue
```

First, import the functions `voice_to_text()` and `print_say()` from the *mptpkg* package directly. Recall that in the script *__init__.py*, you've already imported the two functions from the modules *.mysr* and *.mysay* to the package. As a result, here you can directly import the two functions from the package.

The rest of the script is the same as that in *repeat_me1.py*. It repeats what you say. If you say, "Stop listening," the script stops.

The following is an interaction with *repeat_me2.py*, with my voice input in bold:

```
Python is listening...
you just said how are you
Python is listening...
you just said I am testing a python package
Python is listening...
you just said stop listening; goodbye!
```

As you can see, the script is working properly, which means you've successfully imported functions from the local package.

More on Python Packages

Before you move on, I want to mention a couple of things about Python packages.

First, you can add more modules to your package. Later in this book, you'll add more modules to the existing local package *mptpkg*. You'll use just one local package for the whole book. This will reduce the number of directories and help organize your files.

Second, if you have an interesting package that you want to share with the rest of the world, you can easily do so. You just need to add a few more files, such as the license, a README file, and so on. For a tutorial on how to distribute your Python packages, see, for example, the Python Packaging Authority website, *https://packaging.python.org/tutorials/packaging-projects/*.

Interactive Guess the Number Game

Guess the Number is a popular game in which one player writes down a number and asks the other player to guess it in a limited number of attempts. After each guess, the first player tells whether the guess is correct, too high, or too low.

Various versions of the game are available online and in books, and we'll look at our own version to guess a number between one and nine. Start a new script and save it as *guess_hs.py*; the *hs* stands for *hear and say*.

Because the script is relatively long, I'll break it into three parts and explain them one by one. Listing 5-1 gives the first part.

```
❶ import time
  import sys

  # Import functions from the local package mptpkg
  from mptpkg import voice_to_text
  from mptpkg import print_say

  # Print and announce the rules of the game in a human voice
❷ print_say('''Think of an integer,
        bigger or equal to 1 but smaller or equal to 9,
        and write it on a piece of paper''')
  print_say("You have 5 seconds to write your number down")
  # Wait for five seconds for you to write down the number
  time.sleep(5)
  print_say('''Now let's start. I will guess a number and you can say:
      too high, that is right, or too small''')
  # The script asks in a human voice whether the number is 5
  print_say("Is it 5?")
  # The script is trying to get your response and save it as re1
  # Your response has to be 'too high', 'that is right', or 'too small'
❸ while True:
      re1 = voice_to_text()
      print_say(f"You said {re1}")
      if re1 in ("too high", "that is right", "too small"):
          break
  # If you say "that is right", game over
      if re1 == "that is right":
          print_say("Yay, lucky me!")
          sys.exit
  --snip--
```

Listing 5-1: Part 1 of the Guess the Number game

We start the script by importing needed modules ❶. We import the *time* module so we can pause the script for a period of time. We also import the *sys* module to exit the script when it is finished.

As discussed in the previous section, we import `voice_to_text()` and `print_say()` from the local package *mptpkg* to convert voice to text as well as to print out and speak the text message.

The script then speaks and prints out the rules of the game ❷. Since the instructions span several lines, we put them in triple quotation marks to make them more readable.

NOTE *When you have text that spans multiple lines and you want to print it or convert it to speech, use triple quotation marks; for example:*

```
print(''' Line 1 text,
          line 2 text,
          line 3 text''')
```

The script announces that you have five seconds to write down a number then pauses for five seconds by using `sleep()` to give you time to write your number.

The script then begins to guess; it will ask in a human voice whether the number is five. At ❸, we start an infinite loop to take your voice input. When you speak into the microphone, the computer converts your voice input into a text string variable named `re1`. The script repeats what you said back to you. Your response needs to be one of three phrases: "too high," "that is right," or "too small." If it isn't, the script will keep asking you for a response until it matches one of the phrases. This gives you a chance to have a correct response before the script moves on to the next step.

If your response is "that is right," the computer will say, "Yay, lucky me!" and exit the script. We'll enter the behavior for the response "too high" next. Listing 5-2 shows the middle part of the *guess_hs.py* script.

```
--snip--
# If you say "too high", the computer keeps guessing
elif re1 == "too high":
    # The computer guesses 3 the second round
    print_say("Is it 3?")
    # The computer is trying to get your response to the second guess
    while True:
        re2 = voice_to_text()
        print_say(f"You said {re2}")
        if re2 in ("too high", "that is right", "too small"):
            break
    # If the second guess is right, game over
    if re2 == "that is right":
        print_say("Yay, lucky me!")
        sys.exit
    # If the second guess is too small, the computer knows it's 4
    elif re2 == "too small":
        print_say("Yay, it is 4!")
        sys.exit
    # If the second guess is too high, the computer guesses the third time
    elif re2 == "too high":
        # The third guess is 1
        print_say("Is it 1?")
        # The computer is getting your response to the third guess
        while True:
            re3 = voice_to_text()
            print_say(f"You said {re3}")
```

```
            if re3 in ("too high", "that is right", "too small"):
                break
        # If the third guess is too small, the computer knows it's 2
        if re3 == "too small":
            print_say("It is 2!")
            sys.exit
        # If the third guess is right, game over
        elif re3 == "that is right":
            print_say("Yay, lucky me!")
            sys.exit
--snip--
```

Listing 5-2: The "too high" behavior

If your response is "too high," the computer will keep guessing, this time a lower number. The second guess from the computer will be three because guessing three reduces the number of attempts the computer needs to find out the answer. The script will detect and catch your response to the second guess.

Here are the options for your response to the second guess: If it's "that is right," the computer will say "Yay, lucky me!" and exit the script. If it's "too small," the computer will know that the number is four and say so. If it's "too high," the computer will make a third guess of one.

Then, the computer captures your response to the third guess. If your response is "too small," the computer will know that the number is two. If your response is "that is right," the computer will say, "Yay, lucky me!" and exit.

Now let's look at the final section of *guess_hs.py*, which handles a "too small" response to the first guess. Listing 5-3 shows the code.

```
--snip--
# If you say "too small", the computer keeps guessing
elif re1 == "too small":
    # The computer guesses 7 the second round
    print_say("Is it 7?")
    # The computer is trying to get your response to the second guess
    while True:
        re2 = voice_to_text()
        print_say(f"You said {re2}")
        if re2 in ("too high", "that is right", "too small"):
            break
    # If the second guess is right, game over
    if re2 == "that is right":
        print_say("Yay, lucky me!")
        sys.exit
    # If the second guess is too high, the computer knows it's 6
    elif re2 == "too high":
        print_say("Yay, it is 6!")
        sys.exit
    # If the second guess is too small, the computer guesses the third time
    elif re2 == "too small":
        # The third guess is 8
```

```
print_say("Is it 8?")
while True:
    re3 = voice_to_text ()
    print_say(f"You said {re3}")
    if re3 in ("too high", "that is right", "too small"):
        break
# If the third guess is too small, the computer knows it's 9
if re3 == "too small":
    print_say("It is 9!")
    sys.exit
# If the third guess is right, game over
elif re3 == "that is right":
    print_say("Yay, lucky me!")
    sys.exit
```

Listing 5-3: The "too small" behavior

The final section of the script is similar to the middle section. If you tell the computer that the first guess of five is "too small," the computer will give you a second guess of seven. The script will then catch your response to the second guess.

If you respond "that is right," the computer will say, "Yay, lucky me!" and exit the script. If you say "too high," the computer will know that the number is six. If your response is "too small," the computer will make a third guess of eight.

The computer then captures your response to the third guess. If your response is "too small," the computer will know that the number is nine. If your response is "that is right," the computer will say, "Yay, lucky me!" and exit the script.

If you have a good internet connection in a fairly quiet environment, you can have close-to-perfect communication with the computer. The internet connection is important because we use the Google Web Speech API to convert voice input into text. The *SpeechRecognition* module has an offline method called `recognize_sphinx()`, but it makes a lot of mistakes, so we use the online method.

Here's the written output from the script when my number was 8 (my voice input is in bold):

```
Please think of an integer,
bigger or equal to 1 but smaller or equal to 9,
and write on a piece of paper
You have 5 seconds to write it down
Now let's start. I will guess a number and you can say:
too high, that is right, or too small
Is it 5?
You said too small
Is it 7?
You said too small
Is it 8?
You said that is right
Yay, lucky me!
```

The script understood every word I said perfectly. This is, of course, partly because I chose certain words to avoid ambiguity. When building your own projects, you'll want to use voice commands that are unique or put the words in context to get consistently correct results. Since each voice command is usually short, the Python script may have difficulty grasping the context of your voice input and returning the right words.

For example, if you say "too large" into the microphone, the script may return "two large," which is a phrase that does make sense. That is why we use "too high" instead of "too large" in *guess_hs.py*.

Similarly, when I spoke "too low" into the microphone, the script returned "tulo" from time to time. When I use "too small," I get the correct response each time.

TRY IT OUT

Run *guess_hs.py* and play a few rounds. See if Python can understand each of your responses on the first try.

Speaking Newscast

In this project, we'll scrape the NPR News website to collect the latest news summary and have Python read it out loud. This project is split into two scripts: one to scrape and organize the news, another to handle the speech recognition and text-to-speech features. Let's start with the web scraping.

Scrape the News Summary

First, we need to scrape the information from the news site and compile it into a clean and readable format.

Different news sites arrange their content differently, so the methods for scraping are often slightly different. You can refer to Chapter 6 for the basics of web scraping. If you're interested in scraping other news sites, you'll need to adjust this code based on the features of the website. Let's first look at the site and the corresponding source code.

The news we're interested in is on the front page of the NPR News website, shown in Figure 5-2.

One handy feature of this page is the short news summaries. As you can see, the front page lists the latest news with a short summary for each news article.

You want to extract the news title and the teaser of each news article and print them out. To do this, you need to locate the corresponding tags in the HTML program.

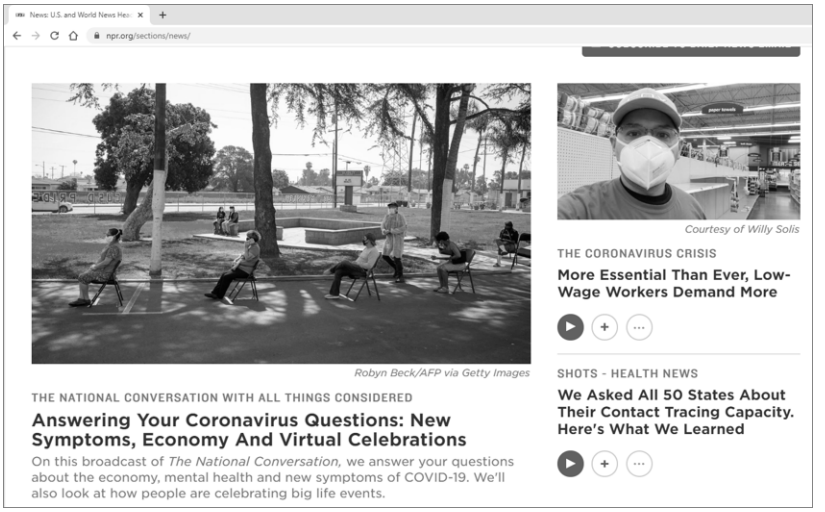

Figure 5-2: News summaries on the NPR News front page

While on the web page, press CTRL-U on your keyboard. The source code for the web page should appear. You can see that it's almost 2,000 lines long. To locate the tags you need, press CTRL-F to open a search box at the top-right corner. Because the title of the first news article starts with "Answering Your Coronavirus Questions," as shown in Figure 5-2, you should enter **`Answering Your Coronavirus Questions`** and click **Search**. Then skip to the corresponding HTML code, shown in Listing 5-4.

```
--snip--
❶ <div class="item-info">
    <div class="slug-wrap">
    <h3 class="slug">
<a href="https://www.npr.org/series/821003492/the-national-conversation-with-
all-things-considered">The National Conversation With All Things Considered
</a>
</h3>
    </div>
❷ <h2 class="title">
<a href="https://www.npr.org/2020/04/28/847585398/answering-your-coronavirus-
questions-new-symptoms-economy-and-virtual-celebratio" data-
metrics='{"action":"Click Featured Story Headline 1-
3","category":"Aggregation"}' >Answering Your Coronavirus Questions: New
Symptoms, Economy And Virtual Celebrations
</a>
</h2>
❸ <p class="teaser">
<a href="https://www.npr.org/2020/04/28/847585398/answering-your-coronavirus-
questions-new-symptoms-economy-and-virtual-celebratio"><time datetime="2020-
04-28"><span class="date">April 28, 2020 &#149; </span></time>On this
broadcast of <em>The National Conversation, </em>we answer your questions
about the economy, mental health and new symptoms of COVID-19. We'll also
look at how people are celebrating big life events.
```

```
</a>
</p>
</div>
--snip--
```

Listing 5-4: Part of the source code for the NPR News front page

Notice that all the title and teaser information are encapsulated in a parent `<div>` tag with a `class` attribute of `item-info` ❶. Information for the news title is held in a child `<h2>` tag with a `class` attribute of `title` ❷. The information for the teaser is held in a child `<p>` tag with a `class` attribute of `teaser` ❸.

We'll use these patterns to write a Python script to extract the information we need. The script *news.py* will scrape the information and organize all titles and summaries in a clean and concise way. I've added comments in places that need more detailed explanations.

The script will compile the news summary and print it out in text. Enter Listing 5-5 and save it as *news.py*.

```
# Import needed modules
import requests
import bs4

# Obtain the source code from the NPR news website
❶ res = requests.get('https://www.npr.org/sections/news/')
res.raise_for_status()
# Use beautiful soup to parse the code
soup = bs4.BeautifulSoup(res.text, 'html.parser')
# Get the div tags that contain titles and teasers
div_tags = soup.find_all('div',class_="item-info")
# Index different news
❷ news_index = 1
# Go into each div tag to retrieve the title and the teaser
❸ for div_tag in div_tags:
    # Print the news index to separate different news
    print(f'News Summary {news_index}')
    # Retrieve and print the h2 tag that contains the title
    h2tag = div_tag.find('h2', class_="title")
    print(h2tag.text)
    # Retrieve and print the p tag that contains the teaser
    ptag = div_tag.find('p', class_="teaser")
    print(ptag.text)
    # Limit to the first 10 news summaries
    news_index += 1
    if news_index>10:
        break
```

Listing 5-5: Python code to scrape the NPR News front page

We start by importing the needed modules *bs4* and *requests* (*bs4* is the newest version of the Beautiful Soup library). Follow the three steps in Chapter 2 for installing these modules if you need to.

At ❶, we obtain the source code for the NPR News front page, which is in HTML format. We then use the *bs4* module to parse HTML files. Because the information we need is encapsulated in `<div>` tags with a `class` attribute of `item-info`, we find all such tags and put them in a list called *div_tags*. To separate different news summaries, we create a variable *news_index* to mark them ❷.

We then go into each individual `<div>` tag we've collected ❸. First, we print out the news summary index to separate out individual news items. Second, we extract the `<h2>` tag that contains the news title and print it out. Third, we extract the `<p>` tag that contains the news summary and print it out. Finally, we stop if the news index exceeds 10 so that we limit the printout to 10 news summaries.

If you run *news.py*, the output will look like Listing 5-6.

```
News Summary 1
Answering Your Coronavirus Questions: New Symptoms, Economy And Virtual Celebrations
April 28, 2020 • On this broadcast of The National Conversation, we answer your questions
about the economy, mental health and new symptoms of COVID-19. We'll also look at how people
are celebrating big life events.
News Summary 2
More Essential Than Ever, Low-Wage Workers Demand More
April 28, 2020 • In this lockdown, low-wage workers have been publicly declared "essential" —
up there with doctors and nurses. But the workers say their pay, benefits and protections
don't reflect it.
News Summary 3
We Asked All 50 States About Their Contact Tracing Capacity. Here's What We Learned
April 28, 2020 • To safely reopen without risking new COVID-19 outbreaks, states need enough
staffing to do the crucial work of contact tracing. We surveyed public health agencies to
find out how much they have.
News Summary 4
Coronavirus Has Now Killed More Americans Than Vietnam War
April 28, 2020 • The number of lives taken by COVID-19 in the U.S. has reached a grim
milestone: More people have died of the disease than the 58,220 Americans who perished in the
Vietnam War.
--snip--
```

Listing 5-6: News summary scraped from the NPR News front page

Now we'll get Python to read the news to us.

Add the Text-to-Speech Features

The next step is to have the text-to-speech module convert the news summary into spoken words. Add Listing 5-7 into a new file and save it as *news_hs.py*.

```
# Import needed modules
import requests
import bs4
import sys

# Import functions from the local package mptpkg
from mptpkg import voice_to_text
```

```
from mptpkg import print_say
# Define the news_teaser() function
❶ def news_teaser():
    --snip--
      ❷ print_say(f'News Summary {news_index}')
        h2tag = div_tag.find('h2', class_="title")
        print_say(h2tag.text)
        ptag = div_tag.find('p', class_="teaser")
        print_say(ptag.text)
        --snip--
# Print and ask you if you like to hear the news summary
print_say("Would you like to hear the NPR news summary?")
# Capture your voice command
inp = voice_to_text().lower()
# If you answer yes, activate the newscast
if inp == "yes":
    news_teaser()
# Otherwise, exit the script
else:
    sys.exit
```

Listing 5-7: Python code for a voice-activated newscast

We first import the usual modules, and we import `voice_to_text()` and `print_say()` from the self-made *mptpkg* package.

We then define a function called `news_teaser()` ❶, which accomplishes whatever *news.py* does. The only exception is that instead of just printing out the news index, title, and teaser, it both prints and speaks them ❷. We then set the script to ask, "Would you like to hear the NPR news summary?" The `voice_to_text()` function captures your voice response and converts it into a string variable with all lowercase letters. If you say yes, Python will start broadcasting the news. If you answer anything other than yes, the script will exit.

TRY IT OUT

Run *news_hs.py* and hear news from NPR. To save time, modify the script so that you'll hear only the first 5 news summaries instead of 10.

Voice-Controlled Wikipedia

We'll build a talking Wikipedia in this section. Unlike with the newscaster project, we'll use the *wikipedia* module to get the information we need directly. After that, we'll get the script to understand questions you ask, retrieve the answer, and read it aloud.

Access Wikipedia

Python has a *wikipedia* module that does the work of delving into topics you want to know about, so we don't have to code that part ourselves. The module is not in the Python standard library or the Anaconda navigator. You should install it with pip. Open the Anaconda prompt (in Windows) or a terminal (in Mac or Linux) and run the following command:

```
pip install wikipedia
```

Next, run the following script as *wiki.py*:

```
import wikipedia

my_query = input("What do you want to know?\n")
answer = wikipedia.summary(my_query)
print(answer)
```

After the script is running, in the IPython console in the lower-right panel, enter the name of a topic you want to know about. The script will save your inquiry as the variable *my_query*. The `summary()` function will produce a summary answer to your question. Finally, the script prints out the answer from Wikipedia.

I entered `U.S. China trade war` and got the following result:

```
What do you want to know?
U.S. China trade war
China and the United States have been engaged in a trade war through
increasing tariffs and other measures since 2018. Hong Kong economics
professor Lawrence J. Lau argues that a major cause is the growing battle
between China and the U.S. for global economic and technological dominance.
He argues, "It is also a reflection of the rise of populism, isolationism,
nationalism and protectionism almost everywhere in the world, including in the
US."
```

This answer is relatively short. Most searches in Wikipedia will have a much longer result. If you want to limit the length of the responses to, say, the first 200 characters, you can enter `[0:200]` after `answer`.

Add Speech Recognition and Text to Speech

We'll now add the speech recognition and text-to-speech features to the script. Enter Listing 5-8 as *wiki_hs.py*.

```
import wikipedia

# Import functions from the local package mptpkg
from mptpkg import voice_to_text
from mptpkg import print_say
```

```
  # Ask what you want to know
❶ print_say("What do you want to know?")
  # Capture your voice input
❷ my_query = voice_to_text()
  print_say (f"you said {my_query}")
  # Obtain answer from Wikipedia
  ans = wikipedia.summary(my_query)
  # Say the answer in a human voice
  print_say(ans[0:200])
```

Listing 5-8: Python code for a voice-controlled talking Wikipedia

Once you start the script, a voice asks, "What do you want to know?" ❶. At ❷, the script calls `voice_to_text()` to convert your voice input into text. Then, the script retrieves the response to your question from Wikipedia, saves it as a string variable *ans*, and converts it to a human voice.

After running the script, if you say to the microphone, "US Federal Reserve Bank," you'll get a result similar to this:

```
What do you want to know?
you said U.S. federal reserve bank
The Federal Reserve System (also known as the Federal Reserve or simply the
Fed) is the central banking system of the United States of America. It was
created on December 23,
1913, with the enactment
```

I've added the `[0:200]` character limit behind the variable `ans`, so only the first 200 characters of the result are printed and spoken.

And just like that, you have your own voice-controlled talking Wikipedia. Ask away!

TRY IT OUT

Run *wiki_hs.py* and ask Wikipedia about the city you live in now (or the state if the city is not in Wikipedia). See what the output is like.

Voice-Activated Music Player

Here you'll learn how to get Python to play a certain artist or genre of music just by asking for it with a phrase like "Python, play Selena Gomez." You'll speak the name of the artist you want to listen to, and the script will receive that as keywords and then search for those keywords in a particular folder. To do this, you need to be able to traverse files and folders.

Traverse Files in a Folder

Suppose you have a subfolder *chat* in your chapter folder. If you want to list all files in the subfolder, you can use this *traverse.py* script:

```
import os

with os.scandir("./chat") as files:
    for file in files:
        print(file.name)
```

First, the script imports the *os* module. This module gives the script access to functionalities that are dependent on the operating system, such as accessing all files in a folder.

Next, you put all files in the subfolder *chat* into a list called *files*. The script goes through all items in the list, and prints out the name of each item.

The output from the preceding script is as follows after I run it on my computer:

```
book.xlsx
desk.pdf
storm.txt
graduation.pptx
--snip--
HilaryDuffSparks.mp3
country
classic
lessons.docx
SelenaGomezWolves.mp3
TheHeartWantsWhatItWantsSelenaGomez.mp3
```

As you can see, we can traverse all the files and subfolders in a folder and print out their names. Filenames include the file extension. Subfolders have no extension after the subfolder name. For example, I have two folders, *country* and *classic*, in the folder *chat*. As a result, you see `country` and `classic` in the preceding output.

Next, you'll use this feature to select a song you want to play.

Python, Play Selena Gomez

The script in Listing 5-9, *play_selena_gomez.py*, can pick out a song by whatever artist you name (for example, Selena Gomez) and play it. Either save your songs in the subfolder *chat* or replace the file path with a path to somewhere on your computer that you keep music.

```
# Import the required modules
import os
import random
from pygame import mixer
```

```
# Import functions from the local package mptpkg
from mptpkg import voice_to_text
from mptpkg import print_say

# Start an infinite loop to take your voice commands
❶ while True:
    print_say("how may I help you?")
    inp = voice_to_text()
    print_say(f"you just said {inp}")
    # Stop the script if you say 'stop listening'
    if inp == "stop listening":
        print_say("Goodbye! ")
        break
    # If 'play' is in voice command, music mode is activated
  ❷ elif "play" in inp:
        # Remove the word play from voice command
      ❸ inp = inp.replace('play ','')
        # Separate first and last names
        names = inp.split()
        # Extract the first name
        Firstname = names[0]
        # Extract the last name
        if len(names)>1:
            lastname = names[1]
        # If no last name, use the first name as last name;
        else:
            lastname = firstname
        # Create a list to contain songs
        mysongs = []
        # If either first name or last name in the file name, put in list
        with os.scandir("./chat") as files:
            for file in files:
          ❹ if (firstname in file.name or lastname in file.name) \
and "mp3" in file.name:
                mysongs.append(file.name)
        # Randomly select one from the list and play
      ❺ mysong = random.choice(mysongs)
        print_say(f"play the song {mysong} for you")
        mixer.init()
        mixer.music.load(f'./chat/{mysong}')
        mixer.music.play()
        break
```

Listing 5-9: Python code to voice activate a song by an artist on your computer

We first import the needed modules. In particular, we import the *os* module to traverse files and the *random* module to randomly select a song from a list the script will build. We use `mixer()` in the *pygame* module to play the music file.

We then start an infinite loop ❶ to put the script in standby mode to wait for your voice commands. If the script detects the word *play* in your voice command, the music mode is activated ❷. We then replace the word `play` and the whitespace behind it with an empty string ❸ so that your command "Play Selena Gomez" becomes `Selena Gomez`. The next command

separates the first name and the last name. For artists who are known by just their first names (such as Madonna, Prince, or Cher), we put their first name as a placeholder in the variable `lastname`.

We then traverse through all files in the subfolder *chat*. If a file has the *mp3* extension and contains either the first or the last name ❹, it will be added to the list *mysongs*. We use `choice()` from the *random* module to randomly select a song in the list *mysongs* ❺ and load it with `mixer.music.load()`. After that, we use `mixer.music.play()` to play it.

As a result, once you say to the script, "Play Selena Gomez," one of the two songs in the subfolder *chat*, *SelenaGomezWolves.mp3* or *TheHeartWantsWhatItWantsSelenaGomez.mp3*, will start playing.

NOTE *We use the* pygame *module to play music files in this book. Depending on which operating system you are using, other modules, such as* playsound *or* vlc, *can also play music files in Python. Alternatively, you can use* `os.system()` *to open music files in your computer's default music player, as discussed in Chapter 3.*

TRY IT OUT

Save several songs by your favorite artist, making sure that the filenames contain the artist's first and last name. Then edit and run *play_selena_gomez.py* so that when you say, "Python, play *Firstname Lastname*," one of your songs will start playing.

Python, Play a Country Song

What we'll do now is similar to interacting with the script *play_selena_gomez.py*, but here you'll learn how to access different subfolders by using the *os* module as well as a different way of playing music files.

Suppose you've organized your songs by genre. You put all classical music files in the subfolder *classic*, and all country music files in the folder *country*, and so on. You've placed these subfolders in the folder *chat* you just created.

We want to write a script so that when you say, "Python, play a country song," the script will randomly select a song from the folder *country* and play it. Enter the code in Listing 5-10 and save it as *play_genre.py*.

```
# Import needed modules
import os
import random
from pygame import mixer

# Import functions from the local package mptpkg
from mptpkg import voice_to_text
from mptpkg import print_say
```

```
while True:
    print_say("how may I help you?")
    inp = voice_to_text().lower()
    print_say(f'you just said {inp}')
    if inp == "stop listening":
        print_say('Goodbye!')
        break
    elif "play a" in inp and "song" in inp:
        # Remove 'play a' and 'song' so that only the genre name is left
      ❶ inp = inp.replace('play a ','')
      ❷ inp = inp.replace(' song','')

        # Go to the genre folder and randomly select a song
        with os.scandir(f"./chat/{inp}") as entries:
            mysongs = [entry.name for entry in entries]
        # Use pygame mixer to play the song
      ❸ mysong = random.choice(mysongs)
        print_say(f"play the song {mysong} for you")
        mixer.init()
        mixer.music.load(f"./chat/{inp}/{mysong}")
        mixer.music.play()
        break
```

Listing 5-10: Python code to voice activate a song by genre

Python checks for the terms *play a* and *song* in the voice command and activates the music mode if it finds them. The script then replaces `play a` ❶ and `song` ❷ as well as the whitespace behind them with an empty string, leaving only the genre—`country`, in this case—in the voice command. This is used as the folder for the script to search: in this case, *./chat/country*. Finally, the script randomly selects a song from the folder ❸ and plays it.

Note that we use `lower()` after `voice_to_text()` in the script so that the voice command is all lowercase. We do this because the script sometimes converts the voice command into `play A Country Song`. We can avoid mismatch due to capitalization. On the other hand, the path and filenames are not case sensitive, so even if you have capital letters in your path or filenames, there will not be any mismatch.

TRY IT OUT

Organize your music into various categories. Save a few songs in the subfolder *classic* in the *chat* folder you created. If you say, "Play a classic song," see if a song in the folder will start playing.

Summary

In this chapter, you first learned to create a Python package to contain the local text-to-speech and speech recognition modules. After that, you built several real-world applications that can understand voice commands, react, and speak.

You created a voice-controlled, talking Guess the Number game. In the game, you pick a number between one and nine and interact with the script to let it guess. Then you learned how to parse text to extract a news summary from the NPR website, adding the speech recognition and text-to-speech features to make a voice-controlled newscast.

You learned how to use the *wikipedia* module to obtain answers to your inquiries.

You traversed files in a folder on your computer by using the *os* module, and then created a script that plays a genre or artist when you ask it to.

Now that you know how to make Python talk and listen, you'll apply both features to many other interesting situations throughout the rest of the book so that you can interact with your computer via voice only.

End-of-Chapter Exercises

1. Modify *guess_hs.py* so that the third guess of the script is two instead of one.
2. Change *wiki.py* so that it prints out the first 300 characters of the result from Wikipedia.
3. Modify *play_genre.py* so that the script plays music by using the *os* module and your default music player on your computer, instead of the *pygame* module.
4. Suppose the music files on your computer are not in MP3 format but in WAV format. How can you modify *play_selena_gomez.py* so that the script still works?