```
# Define a set named sample_set.
sample_set = {1.98, 98.9, 74.95, 2.5, 1, 16.3}
# Show the whole set
print(sample_set)

# Use len to get the length of a set.
print(len(sample_set))

# Use in to determine if the set contains a value
print(74.95 in sample_set)

# Use add() to add one item to a set.
sample_set.add(11.23)

# Use update() to add a [list] to a set.
sample_set.update([88, 123.45, 2.98])

print("\nSample set after .add() and .update()")
print(sample_set)

# Loop through the set and print each item right-aligned and formatted.
print("\nLoop through set and print each item formatted.")
for price in sample_set:
    print(f"{price:>6.2f}")
```

```
{1.98, 98.9, 2.5, 1, 74.95, 16.3}
6
True

Sample set after .add() and .update()
{1.98, 98.9, 2.5, 1, 2.98, 74.95, 11.23, 16.3, 88, 123.45}

Loop through set and print each item formatted.
  1.98
 98.90
  2.50
  1.00
  2.98
 74.95
 11.23
 16.30
 88.00
123.45
```

FIGURE 3-12: Playing about with Python sets.

IN THIS CHAPTER

- Why use data dictionaries?
- Creating a data dictionary
- Looping through a dictionary
- Data dictionary methods

Chapter 4
Cruising Massive Data with Dictionaries

Data dictionaries, also called *associative arrays* in some languages, are kind of like lists, which we discuss in Chapter 3. But instead of each item in the list being identified by its position in the list, each item is uniquely identified by a *key*. You can define the key, which can be a string or a number, yourself. All that matters is that it be unique to each item in the dictionary.

To understand why uniqueness matters, think about phone numbers, email addresses, and Social Security numbers. If two or more people had the same phone number, then whenever someone called that number, all those people would get the call. If two or more people had the same email address, then all those people would get the same email messages. If two or more people had the same Social Security number, and one of those people was a million dollars behind in their taxes, you better hope you can convince the tax folks you're not the one who owes a million dollars, even though your Social Security number is on the past-due bill.

The key in a dictionary represents one unique thing, and you can associate a *value* with that key. The value can be a number, string, list, tuple — just about anything, really. So you can think of a data dictionary as being kind of like a table where the first column contains a single item of information that's unique to that item and the second column, the value, contains information that's relevant to, and perhaps unique to, that key. In the example in Figure 4-1, the left column contains a key that's unique to each row. The second column is the value assigned to each key.

Key		Value
"htanaka"	=	"Haru Tanaka"
"ppatel"	=	"Priya Patel"
"bagarcia"	=	"Benjamin Alberto Garcia"
"zmin"	=	"Zhang Min"
"farooqi"	=	"Ayesha Farooqi"
"hajackson"	=	"Hanna Jackson"
"papatel"	=	"Pratyush Aarav Patel"
"hrjackson"	=	"Henry Jackson"

FIGURE 4-1: A data dictionary with keys in the left column, values in the right.

The left column shows an abbreviation for a person's name. Some businesses use names like these to give user accounts and email addresses to their employees.

The value for the key doesn't have to be a string or an integer. It can be a list, tuple, or set. For example, in the dictionary in Figure 4-2, the value of each key includes a name, a year (perhaps the year of hire or birth year), a number (which may be number of dependents the person claims for taxes), and a Boolean True or False value (which may have indicate whether they have a company cellphone). For now, it doesn't matter what each item of data represents. What matters is that for each key you have a list (enclosed in square brackets) that contains four pieces of information about each key.

Key		Value
"htanaka"	=	["Haru Tanaka", 2000, 0, True]
"ppatel"	=	["Priya Patel", 2015, 1, False]
"bagarcia"	=	["Benjamin Alberto Garcia", 1999, 2, True]
"zmin"	=	["Zhang Min", 2017, 0, False]
"farooqi"	=	["Ayesha Farooqi", 2001, 1, True]
"hajackson"	=	["Hanna Jackson", 1998, 0, False]
"papatel"	=	["Pratyush Aarav Patel", 2011, 2, True]
"hrjackson"	=	["Henry Jackson", 2016, 0, False]

FIGURE 4-2: A data dictionary with lists as values.

A dictionary may also consist of several different keys, each representing a piece of data. For example, rather than have a row for each item with a unique key, you may make each employee their own little dictionary, Then you can assign a key name to each unit of information. The dictionary for `htanaka`, then, may look like Figure 4-3.

'htanaka'=	'full_name'	=	'Haru Tanaka'
	'year_hired'	=	2000
	'dependents'	=	0
	'has_company_cell'	=	True

FIGURE 4-3: A data dictionary for one employee.

The dictionary for another employee may have all the same key names, `full_name`, `year_hired`, `dependents`, and `has_company_cell`, but a different value for each of those keys. (See Figure 4-4.)

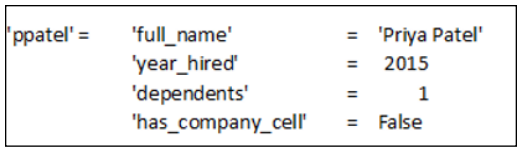


FIGURE 4-4: A data dictionary for another employee.

Each dictionary having multiple keys is common in Python, because the language makes it easy to isolate the specific item of data you want using *object.key* syntax, like this:

```
ppatel.full_name = 'Priya Patel'
ppatel.year_hired= 2015
ppatel,dependents = 1
ppatel.has_company_cell=True
```

The key name is more descriptive that using an index that's based on position, as you can see in the following example.

```
ppatel[0] = 'Priya Patel'
ppatel[1]= 2015
ppatel[2] = 1
ppatel[3]=True
```

Creating a Data Dictionary

The code for creating a data dictionary follows the basic syntax:

```
name = {key:value, key:value, key:value, key:value, ...}
```

The *name* is a name you make up and generally describes to whom or what the key-value pairs refer. The *key*:*value* pairs are enclosed in curly braces. The *key* values are usually strings enclosed in quotation marks, but you can use integers instead if you like. Each colon (:) separates the key name from the value assigned to it. The *value* is whatever you want to store for that key name, and can be a number, string, list . . . pretty much anything. The ellipsis (. . .) just means that you can have as many key-value pairs as you want. Just remember to separate the key-value pairs with commas, as shown in the syntax example.

To make the code more readable, developers often place each key-value pair on a separate line. But the syntax is still the same. The only difference is that there is a line break after each comma, as in the following:

```
name = {
    key:value,
    key:value,
    key:value,
    key:value,
     ...
}
```

If you want to try it out hands-on, open up a Jupyter notebook, a .py file, or a Python prompt, and type in the following code. Notice we created a dictionary named `people`. It contains multiple key-value pairs, each separated by a comma. The keys and values are strings so they're enclosed in quotation marks, and each key is separated from its value with a colon. It's important to keep all of that straight, otherwise the code won't work — yes, even just one missing or misplaced or mistyped quotation mark, colon, comma, or curly brace can mess the whole thing up.

```
people = {
    'htanaka': 'Haru Tanaka',
    'ppatel': 'Priya Patel',
    'bagarcia': 'Benjamin Alberto Garcia',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    'hajackson': 'Hanna Jackson',
    'papatel': 'Pratyush Aarav Patel',
    'hrjackson': 'Henry Jackson'
    }
```

Accessing dictionary data

After you have added the data in, you can work with it in a number of ways. Using `print(people)` — that is, a `print()` function with the name of the dictionary in the parentheses — you get a copy of the whole dictionary, as follows:

```
print(people)
{'htanaka': 'Haru Tanaka', 'ppatel': 'Priya Patel', 'bagarcia': 'Benjamin
    Alberto Garcia', 'zmin': 'Zhang Min', 'afarooqi': 'Ayesha Farooqi',
    'hajackson': 'Hanna Jackson', 'papatel': 'Pratyush Aarav Patel',
    'hrjackson': 'Henry Jackson'}
```

Typically this is not what you want. More often, you're looking for one specific item in the dictionary. In that case, use this syntax:

```
dictionaryname[key]
```

where *dictionaryname* is the name of the dictionary, and *key* is the key value for which you're searching. For example, if you want to know the value of the `zmin` key, you would enter

```
print(people['zmin'])
```

Think of this line as saying *print people sub zmin* where *sub* just means *the specific key*. When you do that, Python returns the value for that one person . . . the full name for `zmin`, in this example. Figure 4-5 shows that output after running the code in a Jupyter notebook cell.

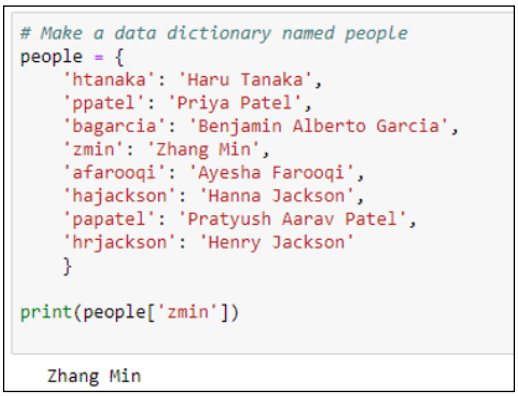


FIGURE 4-5: Printing the value of the `zmin` key in the `people` dictionary.

Notice that in the code, `zmin` is in quotation marks. Those are required because `zmin` is a string. You can use a variable name instead, so long as that variable contains a string. For example, consider the following two lines of code. The first one creates a variable named `person` and puts the string `'zmin'` into that variable. The next line, `print(people[person])` `(print people sub person)` requires no quotation makes because `person` is a variable name.

```
person = 'zmin'
print(people[person])
```

So what do you think would happen if you executed the following code?

```
person = 'hrjackson'
print(people[person])
```

You would, of course, see `Henry Jackson`, the name (value) that goes with the key `'hrjackson'`.

How about if you ran this bit of code?

```
person = 'schmeedledorp'
print(people[person])
```

Figure 4-6 shows what would happen. You get an error because nothing in the `people` dictionary has the key value `'schmeedledorp'`.

```
# Make a data dictionary named people
people = {
    'htanaka': 'Haru Tanaka',
    'ppatel': 'Priya Patel',
    'bagarcia': 'Benjamin Alberto Garcia',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    'hajackson': 'Hanna Jackson',
    'papatel': 'Pratyush Aarav Patel',
    'hrjackson': 'Henry Jackson'
    }

# Look for a person.
person = 'schmeedledorp'
print(people[person])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-16-3e728d397aa2> in <module>()
     13 # Look for a person.
     14 person = 'schmeedledorp'
---> 15 print(people[person])

KeyError: 'schmeedledorp'
```

FIGURE 4-6: Python's way of saying there is no *schmeedledorp*.

Getting the length of a dictionary

The number of items in a dictionary is considered its *length*. As with lists, you can use the `len()` statement to determine a dictionary's length. The syntax is:

```
len(dictionaryname)
```

As always, replace *`dictionaryname`* with the name of the dictionary you're checking. For example, the following code creates a dictionary, then stores its length in a variable named `howmany`:

```
people = {
    'htanaka': 'Haru Tanaka',
    'ppatel': 'Priya Patel',
    'bagarcia': 'Benjamin Alberto Garcia',
```

```
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    'hajackson': 'Hanna Jackson',
    'papatel': 'Pratyush Aarav Patel',
    'hrjackson': 'Henry Jackson'
    }
# Count the number of key:value pairs and put in a variable.
howmany = len(people)
# Show how many.
print(howmany)
```

When executed, the `print` statement shows `8`, the value of the `hominy` variable, as determined by the number of key-value pairs in the dictionary.

TIP

As you may have guessed, an empty dictionary that contains no key-value pairs has a length of zero.

Seeing whether a key exists in a dictionary

You can use the `in` keyword to see whether a key exists. If the key exists, then `in` returns `True`. If the key does not exist, `in` returns `False`. Figure 4-7 shows a simple example with two `print()` statements. The first one checks to see whether `hajackson` exists in the dictionary. The second checks to see whether `schmeedledorp` exists in the dictionary.

```
# Make a data dictionary named people
people = {
    'htanaka': 'Haru Tanaka',
    'ppatel': 'Priya Patel',
    'bagarcia': 'Benjamin Alberto Garcia',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    'hajackson': 'Hanna Jackson',
    'papatel': 'Pratyush Aarav Patel',
    'hrjackson': 'Henry Jackson'
    }

# Is there an hajackson in the people dictionary?
print('hajackson' in people)

# Is there an schmeedledorp in the people dictionary?
print('schmeedledorp' in people)
```

```
True
False
```

FIGURE 4-7: Seeing if a key exists in a dictionary.

As you can see, the first `print()` statement shows `True` because `hajackson` is in the dictionary. The second one returns `False` because `schmeedledorp` isn't in the dictionary.

Getting dictionary data with get()

Having the program kind of crash and burn when you look for something that isn't in the dictionary is a little harsh. A more elegant way to handle that is to use the `.get()` method of a data dictionary. The syntax is:

```
dictionaryname.get(key)
```

Replace *dictionaryname* with the name of the dictionary you're searching. Replace *key* with the thing you're looking for. Notice that `get()` uses parentheses, not square brackets. If you look for something that *is* in the dictionary, such as this:

```
# Look for a person.
person = 'bagarcia'
print(people.get(person))
```

. . . you get the same result as you would using square brackets.

What makes `.get()` different is what happens when you search for a non-existent name. You don't get an error, and the program doesn't just crash and burn. Instead, `get()` just gracefully returns the word `None` to let you know that no person named `schmeedledorp` is in the `people` dictionary. Figure 4-8 shows an example.

```
# Make a data dictionary named people
people = {
    'htanaka': 'Haru Tanaka',
    'ppatel': 'Priya Patel',
    'bagarcia': 'Benjamin Alberto Garcia',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    'hajackson': 'Hanna Jackson',
    'papatel': 'Pratyush Aarav Patel',
    'hrjackson': 'Henry Jackson'
    }

# Look for a person.
person = 'schmeedledorp'
print(people.get(person))
```

```
None
```

FIGURE 4-8: Python's way of saying there is no schmeedledorp.

You can actually pass two values to `get()`, the second one being what you want it to return if the `get` fails to find what you're looking for. For instance, in the following line of code we search for `schmeedledorp` again, but this time if it doesn't find that person it doesn't display the word `None`. Instead, it displays the rather more pompous message `Unbeknownst to this dictionary`.

```
print(people.get('schmeedledorp','Unbeknownst to this dictionary'))
```

Changing the value of a key

Dictionaries are mutable, which means you can change the contents of the dictionary from code (not that you can make the dictionary shut up). The syntax is simply:

```
dictionaryname[key] = newvalue
```

Replace *dictionaryname* with the name of the dictionary, *key* with the key that identifies that item, and *newvalue* with whatever you want the new value to be.

For example, supposed Hanna Jackson gets married and changes her name to Hanna Jackson-Smith. You want to keep the same key, just change the value. The line that reads `people['hajackson'] = "Hanna Jackson-Smith"` actually makes the change. The `print()` statement below that line shows the value of `hajackson` after executing that line of code. As you can see, that name has indeed been changed to `Hana Jackson-Smith`. See Figure 4-9.

```
# Print hajackson's current value.
print(people['hajackson'])

# Change the value of the hajackson key.
people['hajackson'] = "Hanna Jackson-Smith"

#Print the hajackson key to verify that the value has changed.
print(people['hajackson'])
```

```
Hanna Jackson
Hanna Jackson-Smith
```

FIGURE 4-9: Changing the value associated with a key in dictionary.

TECHNICAL STUFF

In real life, the data in the dictionary would probably also be stored in some kind of external file so it's permanent. Additional code would be required to save the dictionary changes to that external file. But you need to learn these basics before you get into all of that. So let's just forge ahead with dictionaries for now.

Adding or changing dictionary data

You can use the dictionary `update()` method to add a new item to a dictionary, or to change the value of a current key. The syntax is

```
dictionaryname.update(key, value)
```

Replace *dictionaryname* with the name of the dictionary. Replace *key* with the key of the item you want to add or change. If the key you specify doesn't already exist with the dictionary, it will be added as a new item with the *value* you specify. If the *key* you specify does exist, then nothing will be added. The value of the key will be changed to whatever you specify as the *value*.

For example, consider the following Python code that creates a data dictionary named people and put two peoples' names into it:

```
# Make a data dictionary named people.
people = {
    'papatel': 'Pratyush Aarav Patel',
    'hrjackson': 'Henry Jackson'
    }

# Change the value of the hrjackson key.
people.update({'hrjackson':'Henrietta Jackson'})
print(people)

# Update the dictionary with a new property:value pair.
people.update({'wwiggins':'Wanda Wiggins'})
```

The first update line, shown below. . .

```
people.update({'hrjackson':'Henrietta Jackson'})
```

. . . changes the value for `hrjackson` from Henry Jackson to 'Henrietta Jackson. It changes the existing name because the key `hrjackson` already exists in the data dictionary.

The second `update()` reads as follows:

```
people.update({'wwiggins':'Wanda Wiggins'})
```

There is no key `wwiggins` in the dictionary. So update() can't change the name for wwiggins. So instead that line adds a new key-value pair to the dictionary with `wwigins` as the `key` and `Wanda Wiggins` as the `value`.

The code doesn't specify whether to change or add the value. It doesn't need to because the decision is made automatically. Each key in a dictionary must be unique; you can't have two or more rows with the same key. So when you do an `update()`, it first checks to see whether the key already exists. If it does, then only the value of that key is modified; nothing new is added. If the key does not already exist in the dictionary, then there is nothing to modify so the whole new key-value is added to the dictionary. That's automatic, and the decision about which action to perform is simple:

- If the key already exists in the dictionary, then its value is updated, because no two items in a dictionary are allowed to have the same key.

» If the key does *not* already exist, then the key-value pair is added because there is nothing in the dictionary that already has that key, so the only choice is to add it.

After running the code above, the dictionary contains three items, `paptel`, `hrjackson` (with the new name), and `wwiggins`. Adding the following lines to the end of that code displays everything in the dictionary:

```
# Show what's in the data dictionary now.
for person in people.keys():
          print(person + "=" + people[person])
```

If you add that code and run it again, you get the output below, which shows the complete contents of the data dictionary at the end of that program:

```
papatel = Pratyush Aarav Patel
hrjackson = Henrietta Jackson
wwiggins = Wanda Wiggins
```

As you may have guessed, you can loop through a dictionary in much the same way you loop through lists, tuples, and sets. But there are some extra things you can do with dictionaries, so let's take a look at those next.

Looping through a Dictionary

You can loop through each item in a dictionary in much the same way you can loop through lists and tuples. But you have some extra options. If you just specify the dictionary name in the `for` loop, you get all the keys, as follows:

```
for person in people:
    print(person)

htanaka
ppatel
bagarcia
zmin
afarooqi
hajackson
papatel
hrjackson
```

If you want to see the value of each item, keep the `for` loop the same, but print *`dictionaryname`*[*`key`*] where dictionary name is the name of the dictionary (`people` in our example) and *`key`* is whatever name you use right after the `for` in the loop (`person`, in the following example).

```
for person in people:
    print(people[person])
```

Running this code against the sample `people` dictionary lists all the names, as follows:

```
Haru Tanaka
Priya Patel
Benjamin Alberto Garcia
Zhang Min
Ayesha Farooqi
Hanna Jackson
Pratyush Aarav Patel
Henry Jackson
```

You can also get all the names just by using a slightly different syntax in the `for` loop: . . . add `.values()` to the dictionary name as in the following. Then you can just print the variable name (`person`), and you will still see the value before you're looping through the values.

```
for person in people.values():
    print(person)
```

Lastly, you can loop through the keys and values at the same time by using `.items()` after the dictionary name in the `for` loop. But you will need two variables after the `for` as well, one to reference the key, the other to reference the value. If you want so see both as you're looking through, then you'll need to use those same names inside the parentheses of the `print`.

For example, the loop in Figure 4-10 uses two variable names, `key` and `value` (although they could be `x` and `y` or anything else) to loop through `people.items()`. The `print` statement displays both the `key` and the `value` with each pass through the loop. In that example, the `print()` also has a literal equal sign (enclosed in quotation marks) to separate the key from the value. As you can see in the output, you get a list of all the keys followed by an equal sign and the value assigned to that key.

```
# Make a data dictionary named people
people = {
    'htanaka': 'Haru Tanaka',
    'ppatel': 'Priya Patel',
    'bagarcia': 'Benjamin Alberto Garcia',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    'hajackson': 'Hanna Jackson',
    'papatel': 'Pratyush Aarav Patel',
    'hrjackson': 'Henry Jackson'
    }

# Loop through .items to get the key and the value.
for key, value in people.items():
    # Show the key and value with = in between.
    print(key, "=", value)
```

```
htanaka = Haru Tanaka
ppatel = Priya Patel
bagarcia = Benjamin Alberto Garcia
zmin = Zhang Min
afarooqi = Ayesha Farooqi
hajackson = Hanna Jackson
papatel = Pratyush Aarav Patel
hrjackson = Henry Jackson
```

FIGURE 4-10: Looping through a dictionary with `items()` and two variable names.

Data Dictionary Methods

If you've been diligently following along chapter to chapter, you may have noticed that some of the methods for data dictionaries look similar to those for lists, tuples, and sets. So maybe now would be a good time to list all the methods that dictionaries offer for future reference. (See Table 4-1.) You've already seen some put to use in this chapter. We get to the others a little later.

TABLE 4-1 Data Dictionary Methods

Method	What it Does
`clear()`	Empties the dictionary by remove all keys and values.
`copy()`	Returns a copy of the dictionary.
`fromkeys()`	Returns a new copy of the dictionary but with only specified keys and values.
`get()`	Returns the value of the specified key, or None if it doesn't exist.
`items()`	Returns a list of items as a tuple for each key-value pair.
`keys()`	Returns a list of all the keys in a dictionary.
`pop()`	Removes the item specified by the key from the dictionary, and stores it in a variable.
`popitem()`	Removes the last key-value pair.

(continued)

TABLE 4-1 *(continued)*

Method	What it Does
`setdefault()`	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value.
`update()`	Updates the value of an existing key, or adds a new key-value pair if the specified key isn't already in the dictionary.
`values()`	Returns a list of all the values in the dictionary.

Copying a Dictionary

If you need to make a copy of a data dictionary to work with, use this syntax:

```
newdictionaryname = dictionaryname.copy()
```

Replace *newdictionaryname* with whatever you want to name the new dictionary. Replace *dictionaryname* with the name of the existing dictionary that you want to copy.

Figure 4-11 shows a simple example in which we created a dictionary named `people`. Then we create another dictionary named `peeps2` as a copy of that `people` dictionary. Printing the contents of each dictionary shows that they are indeed identical.

```
# Define a dictionary named people.
people = {
    'htanaka': 'Haru Tanaka',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    }

# Make a copy of the people dictionary and put it in peeps 2.
peeps2 = people.copy()

#Show what's in both dictionaries
print(people)
print(peeps2)
```

```
{'htanaka': 'Haru Tanaka', 'zmin': 'Zhang Min', 'afarooqi': 'Ayesha Farooqi'}
{'htanaka': 'Haru Tanaka', 'zmin': 'Zhang Min', 'afarooqi': 'Ayesha Farooqi'}
```

FIGURE 4-11: Copying a dictionary.

Deleting Dictionary Items

There are several ways to remove data from data dictionaries. The `del` keyword (short for *delete*) can remove any item based on its key. The syntax is as follows:

```
del dictionaryname[key]
```

For example, the following code creates a dictionary named `people`. Then it uses `del people["zmin"]` to remove the item that has `zmin` as its key. Printing the contents of the dictionary afterwards shows that `zmin` is no longer in that dictionary.

```
# Define a dictionary named people.
people = {
    'htanaka': 'Haru Tanaka',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    }

# Show original people dictionary.
print(people)

# Remove zmin from the dictionary.
del people["zmin"]

#Show what's in people now.
print(people)
```

Here is the output of that program:

```
{'htanaka': 'Haru Tanaka', 'zmin': 'Zhang Min', 'afarooqi': 'Ayesha Farooqi'}
{'htanaka': 'Haru Tanaka', 'afarooqi': 'Ayesha Farooqi'}
```

If you forget to include a specific key with the `del` keyword, and specify only the dictionary name, then the entire dictionary is deleted, even its name. For example, if you executed `del people` instead of using `del people["zmin"]` in the preceding code, the output of the second `print(people)` would be an error, as in the following, because after it's deleted the `people` dictionary no longer exists and its content cannot be displayed.

```
{'htanaka': 'Haru Tanaka', 'zmin': 'Zhang Min', 'afarooqi': 'Ayesha Farooqi'}
---------------------------------------------------------------
NameError                 Traceback (most recent call last)
<ipython-input-32-24401f5e8cf0> in <module>()
     13
     14 #Show what's in people now.
---> 15 print(people)

NameError: name 'people' is not defined
```

To remove all key-value pairs from a dictionary without deleting the entire dictionary, use the `clear` method with this syntax:

```
dictionaryname.clear()
```

The following code creates a dictionary named `people`, puts some key-value pairs in it, and then prints that dictionary so you can see its content. Then, `people.clear()` empties out all the data.

```
# Define a dictionary named people.
people = {
    'htanaka': 'Haru Tanaka',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    }

# Show original people dictionary.
print(people)

# Remove all data from the dictionary.
people.clear()

#Show what's in people now.
print(people)
```

The output of running this code shows that initially the people data dictionary contains three `property:value` pairs. After using `people.clear()` to wipe it clear, printing the people dictionary displays `{}`, which is Python's way of telling you that the dictionary is empty.

```
{'htanaka': 'Haru Tanaka', 'zmin': 'Zhang Min', 'afarooqi': 'Ayesha Farooqi'}
{}
```

Using pop() with Data Dictionaries

The `pop()` method offers another way to remove data from a data dictionary. It actually does two things:

- If you store the results of the `pop()` to a variable, that variable gets the value of the popped key.
- Regardless of whether you store the result of the pop in a variable, the specified key is removed from the dictionary.

Figure 4-12 shows an example where in the output, you first see the entire dictionary. Then `adios = people.pop("zmin")` is executed, putting the value of the `zmin` key into a variable named `adios`. Printing the variable (`adios`) shows that the `adios` variable does indeed contain `Zhang Min`, the value of the `zmin` variable. Printing the entire people dictionary again proves that `zmin` has been removed from the dictionary.

```
# Define a dictionary named people.
people = {
    'htanaka': 'Haru Tanaka',
    'zmin': 'Zhang Min',
    'afarooqi': 'Ayesha Farooqi',
    }
# Show original people dictionary.
print(people)

# Pop zmin from the dictionary, store its value in adios variable.
adios = people.pop("zmin")

# Print the contents of adios and people.
print(adios)
print(people)
```

```
{'htanaka': 'Haru Tanaka', 'zmin': 'Zhang Min', 'afarooqi': 'Ayesha Farooqi'}
Zhang Min
{'htanaka': 'Haru Tanaka', 'afarooqi': 'Ayesha Farooqi'}
```

FIGURE 4-12: Popping an item from a dictionary.

Data dictionaries offer a variation on `pop()` that uses this syntax:

```
dictionaryname = popitem()
```

This one is tricky because in some earlier versions of Python it would remove some item at random. That's kind of weird unless you're writing a game or something and you do indeed want to remove things at random. But as of Python version 3.7 (the version used in this book), `popitem()` always removes the very last key-value pair.

If you store the results of `popitem` to a variable, you *don't* get that item's value, which is different from the way `pop()` works. Instead, you get both the key and its value. The dictionary no longer contains that key-value pair. So, in other words, if you replace `adios = people.pop("zmin")` in Figure 4-12 with `adios = people.popitem()`, the output is as follows:

```
{'htanaka': 'Haru Tanaka', 'zmin': 'Zhang Min', 'afarooqi': 'Ayesha Farooqi'}

('afarooqi', 'Ayesha Farooqi')

{'htanaka': 'Haru Tanaka', 'zmin': 'Zhang Min'}
```

Fun with Multi-Key Dictionaries

So far you've worked with a dictionary that has one value (a person's name) for each key (an abbreviation of that person's name). But it's not at all unusual for a dictionary to have multiple key-value pairs for one item of data.

For example, suppose that just knowing the person's full name isn't enough. You want to also know the year they were hired, their date of birth, and whether or not that employee has been issued a company laptop. The dictionary for any one person may look more like this:

```
employee = {
    'name' : 'Haru Tanaka',
    'year_hired' : 2005,
    'dob' : '11/23/1987',
    'has_laptop' : False
}
```

Suppose you need a dictionary of products that you sell. For each product you want to know its name, its unit price, whether or not it's taxable, and how many you currently have in stock. The dictionary may look something like this (for one product):

```
product = {
    'name' : 'Ray-Ban Wayfarer Sunglasses',
    'unit_price' : 112.99,
    'taxable' : True,
    'in_stock' : 10
}
```

Notice that in each example, the key name is always enclosed in quotation marks. We even enclosed the date in `dob` (date of birth) in quotation marks. If you don't, it may be treated as a set of numbers, as in "11 divided by 23 divided by 1987" which isn't useful information. Booleans are either True or False (initial caps) with no quotation marks. Integers (2005, 10) and floats (112.99) are not enclosed in quotation marks either. It's important to make sure you always do these correctly, as shown in the examples above.

The value for a property can be a list, tuple, or set; it doesn't have to be a single value. For example, for the sunglasses product, maybe you offer two models (color), black and tortoise. You could add a `colors` or `model` key and list the items as a comma-separated list in square brackets like this:

```
product = {
    'name' : 'Ray-Ban Wayfarer Sunglasses',
    'unit_price' : 112.99,
    'taxable' : True,
    'in_stock' : 10,
    'models' : ['Black', 'Tortoise']
}
```

Next let's look at how you may display the dictionary data. You can use the simple *dictionaryname*[*key*] syntax to print just the value of each key. For example, using that last product example, the output of this code:

```
print(product['name'])
print(product['unit_price'])
print(product['taxable'])
print(product['in_stock'])
print(product['models'])
```

. . . would be:

```
Ray-Ban Wayfarer Sunglasses
112.99
True
10
['Black', 'Tortoise']
```

You could fancy it up by adding some descriptive text to each `print` statement, followed by a comma and the code. You could also loop through the list to print each model on a separate line. And you can use an f-string to format any data if you like. For example, here is a variation on the `print()` statements shown above:

```
product = {
    'name' : 'Ray-Ban Wayfarer Sunglasses',
    'unit_price' : 112.99,
    'taxable' : True,
    'in_stock' : 10,
    'models' : ['Black', 'Tortoise']
}
print('Name:    ', product['name'])
print('Price:   ',f"${product['unit_price']:.2f}")
print('Taxable: ',product['taxable'])
print('In Stock:',product['in_stock'])
print('Models:')
for model in product['models']:
    print(" " * 10 + model)
```

Here is the output of that code:

```
Name:     Ray-Ban Wayfarer Sunglasses
Price:    $112.99
Taxable:  True
In Stock: 10
Models:
          Black
          Tortoise
```


WARNING

The `" " * 10` on the last line of code says *to print a space (" ") ten times*. In other words, indent ten spaces. If you don't put exactly one space between those quotation marks, you won't get 10 spaces. You'll get 10 of whatever is between the quotation marks, which also means you'll get nothing if you don't put anything between the quotation marks.

Using the mysterious fromkeys and setdefault methods

Data dictionaries in Python offer two methods, named `fromkeys()` and `setdefault()`, which are the cause of much head-scratching among Python learners — and rightly so because it's not easy to find practical applications for their use. But we'll take a shot at it and at least show you exactly what to expect if you ever use these methods in your code.

The `fromkeys()` method uses this syntax:

```
newdictionaryname = dict(iterable[,value])
```

Replace *`newdictionary`* name with whatever you want to name the new dictionary. It doesn't have to be a generic name like *product*. It can be something that uniquely identifies the product, such as a UPC (Universal Product Code) or SKU (Stock Keeping Unit) that's specific to your own business.

Replace the *`iterable`* part with any iterable — meaning, something the code can loop through; a simple list will do. The *`value`* part is optional. If omitted, each key in the dictionary gets a value of `None`, which is simply Python's way of saying *no value has been assigned to this key in this dictionary yet*.

Here is an example in which we created a new dictionary `DWC001` (pretend this is an SKU for some product in our inventory). We gave it a list of key names enclosed in square brackets and separated by commas, which makes it a properly defined list for Python. We provided nothing for *`value`*. The code then prints the new

dictionary. As you can see, the last line of code prints the dictionary, which contains the specified key names with each key having a value of None.

```
DWC001 = dict.fromkeys(['name', 'unit_price', 'taxable', 'in_stock', 'models'])
print(DWC001)
{'name': None, 'unit_price': None, 'taxable': None, 'in_stock': None, 'models':
    None}
```

Now, let's say you don't want to type out all those key names. You just want to use the same keys you're using in other dictionaries. In that case, you can use *dictionary*.keys() for your iterable list of key names, so long as dictionary refers to another dictionary that already exists in the program. For example, in the following code, we created a dictionary named product that has some key names and nothing specific for the values. Then we used DWC001 = dict.fromkeys(product.keys()) to create a new dictionary with the specific name DWC001 that has the same keys as the generic product dictionary. We didn't specify any values in the dict.fromkeys(product.keys()) line, so each of those keys in the new dictionary will have values set to None.

```
# Create a generic dictionary for products name product.
product = {
    'name': '',
    'unit_price': 0,
    'taxable': True,
    'in_stock': 0,
    'models': []
}
# Create a dictionary names DWC001 that has the same keys as product.
DWC001 = dict.fromkeys(product.keys())

#Show what's in the new dictionary.
print(DWC001)
```

The final print() statement shows what's in the new dictionary. You can see it has all the same keys as the product dictionary, with each value set to None.

```
{'name': None, 'unit_price': None, 'taxable': None, 'in_stock': None, 'models':
    None}
```

The .setdefault() value lets you add a new key to a dictionary, with a predefined value. It only adds a new key and value, though. It will not alter the value for an existing key, even if that key's value is None. So it could come in handy after the fact if you defined dictionaries and then later wanted to add a another property:value pair, but only to dictionaries that don't already have that property in them.

Figure 4-13 shows an example in which we created the DWC001 dictionary using the same keys as the product dictionary. After the dictionary is created, setdefault('taxable',True) adds a key named taxable and sets its value to True — but only if that dictionary doesn't already have a key named taxable. It also adds a key named reorder_point and sets its value to 10, but again, only if that key doesn't already exist.

As you can see in the output from the code, after the fromkeys and setdefault operations, the new dictionary has all the same keys as the product dictionary plus a new key-value pair, reorder_point:10, which was added by the second setdefault. The taxable key in that output, though, is still None, because setdefault won't change the value of an existing key. It only adds a new key with the default value to a dictionary that doesn't already have that key.

```
# Create a generic dictionary for products name product.
product = {
    'name': '',
    'unit_price': 0,
    'taxable': True,
    'in_stock': 0,
    'models': []
}
# Create a dictionary for product SKU # DWC001
DWC001 = dict.fromkeys(product.keys())
DWC001.setdefault('taxable',True)
DWC001.setdefault('models',[])
DWC001.setdefault('reorder_point',100)

# Show what's in the new dictionary.
print("Dictionary after fromkeys() and setdefault()")
print(DWC001)

# Change the taxable field from None to True
print("\nDictionary after fromkeys() and setdefault()")
DWC001['taxable']=True

#Print the dictionary after changing taxable to True
print(DWC001)
```

```
Dictionary after fromkeys() and setdefault()
{'name': None, 'unit_price': None, 'taxable': None, 'in_stock': None, 'models': None, 'reorder_point': 100}

Dictionary after fromkeys() and setdefault()
{'name': None, 'unit_price': None, 'taxable': True, 'in_stock': None, 'models': None, 'reorder_point': 100}
```

FIGURE 4-13: Experimenting with fromkeys and setdefault.

So what if you really did want to set the default of taxable to True, rather than none. The simple solution there would be to use the standard syntax, *dictionaryname* [*key*] = *newvalue* to change the value of the extant taxable key from None to True. The second output in Figure 4-13 proves that changing the value of the key in that manner did work.

Nesting Dictionaries

By now it may have occurred to you that any given program you write may require several dictionaries, each with a unique name. But if you just define a bunch of dictionaries with names, how could you loop through the whole kit-and-caboodle

without specifically accessing each dictionary by name? The answer is, make each of those dictionaries a key-value pair in some containing dictionary, where the key is the unique identifier for each dictionary (for example, a UPC or SKU for each product). The value for each key would then be a dictionary of all the key-value pairs for that dictionary. So the syntax would be:

```
containingdictionaryname = {
    key : {dictionary},
    key : {dictionary},
    key : {dictionary},
     ...
}
```

That's just the syntax for the dictionary of dictionaries. You have to replace all the italicized placeholder names as follows:

- *containingdictionaryname*: This is the name assigned to the dictionary as a whole. It can be any name you like, but should describe what the dictionary contains.
- *key*: Each key value must be unique, such as the UPC or SKU for a product, or the username for people, or even just some sequential number, so long as it's never repeated.
- {*dictionary*} Enclose all the key-value pairs for that one dictionary item in curly braces, and follow that whole thing with a comma if another dictionary follows.

Figure 4-14 shows an example in which we have a dictionary named `products` (plural, because it contains many products). This dictionary in turn contains four individual products. Each of those products has a unique key, like `RB0011`, `DWC0317`, and so forth, which are perhaps in-house SKU numbers that the business uses to manage its own inventory. Each of those four products in turn has `name`, `price`, and `models` keys.

```
# Create a generic dictionary to contain multiple product dictionaries.
products = {
    'RB00111':{'name':'Rayban Sunglasses', 'price':112.98, 'models':['black', 'tortoise']},
    'DWC0317':{'name':'Drone with Camera', 'price':72.95, 'models':['white', 'black']},
    'MTS0540':{'name':'T-Shirt', 'price':2.95, 'models':['small', 'medium', 'large']},
    'ECD2989':{'name':'Echo Dot', 'price':29.99, 'models':[]},
}
```

FIGURE 4-14: Multiple product dictionaries contained within a larger products dictionary.

The complex syntax with all the curly braces, commas, and colons makes it hard to see what's really going on there (not to mention hard to type). But outside of

Python, where you don't have to do all the coding, the exact same data could be stored as a simple table named `Products` with the key names as column headings, like the one in Table 4-2.

TABLE 4-2 A Table of Products

ID (key)	Name	Price	Models
RB00111	Rayban Sunglasses	112.98	black, tortoise
DWC0317	Drone with Camera	72.95	white, black
MTS0540	T-Shirt	2.95	small, medium, large
ECD2989	Echo Dot	29.99	

Using a combination of f-strings and some loops, you could get Python to display that data from the data dictionaries in a neat, tabular format. Figure 4-15 shows an example of such code in a Jupyter notebook, with the output from that code right below it.

```
# Create a generic dictionary to contain multiple product dictionaries.
products = {
    'RB00111':{'name':'Rayban Sunglasses', 'price':112.98, 'models':['black','tortoise']},
    'DWC0317':{'name':'Drone with Camera', 'price':72.95, 'models':['white', 'black']},
    'MTS0540':{'name':'T-Shirt', 'price':2.95, 'models':['small', 'medium', 'large']},
    'ECD2989':{'name':'Echo Dot', 'price':29.99, 'models':[]}
    }
print(f"{'ID':<6} {' Name':<17} {'Price':>8}  {' Models'}")
print('-' * 60)
# Loop through each dictionary in the products dictionary
for oneproduct in products.keys():
    # Get the id of one product.
    id = oneproduct
    # Get the name of one product.
    name = products[oneproduct]['name']
    # Get the unit price of one product and format with $
    unit_price = '$' + f"{products[oneproduct]['price']:,.2f}"
    # Create and empty string variable named models
    models = ''
    # Loop through the models list and tack onto models
    # one item from the list followed by a comma and a space.
    for m in products[oneproduct]['models']:
        models += m + ', '
    # If the models variable is more than two characters in lenth,
    # Peel off the last two characters (last comma and space)
    if len(models) > 2:
        models = models[:-2]
    else:
        models="<none>"
    # Print all the variables with a neat f-string.
    print(f"{id:<6} {name:<17} {unit_price:>8} {models}")
```

```
ID     Name               Price   Models
------------------------------------------------------------
RB00111 Rayban Sunglasses  $112.98 black, tortoise
DWC0317 Drone with Camera   $72.95 white, black
MTS0540 T-Shirt              $2.95 small, medium, large
ECD2989 Echo Dot            $29.99 <none>
```

FIGURE 4-15: Printing data dictionaries formatted into rows and columns.

IN THIS CHAPTER

- Creating a function
- Commenting a function
- Passing information to a function
- Returning values from functions
- Unmasking anonymous functions

Chapter 5

Wrangling Bigger Chunks of Code

In this chapter you learn how to better manage larger code projects by creating your own functions. Functions provide a way to compartmentalize your code into small tasks that can be called from multiple places within an app. For example, if something you need to access throughout the app requires a dozen lines of code, chances are you don't want to repeat that code over and over again every time you need it. Doing so just makes the code larger than it needs to be. Also, if you want to change something, or if you have to fix an error in that code, you don't want to have to do it repeatedly in a bunch of different places. If all that code were contained in a function, you would just have to change or fix it in that one location.

To access the task that the function performs, you just *call* the function from your code. You do this in exactly the same way you call a built-in function like `print`: You just type the name into your code. You can make up your own function names too. So, think of functions as a way to personalize the Python language so its commands fit what you need in your application.

Creating a Function

Creating a function is easy. In fact, feel free to follow along here in a Jupyter notebook cell or .py file if you want to get some hands-on experience. Going hands-on really helps with the understanding and remembering things.

To create a function, start a new line with **`def`** (short for *definition*) followed by a space, and then a name of your own choosing followed by a pair of parentheses with no spaces before or inside. Then put a colon at the end of that line. For example, to create a simple function named `hello()`, type:

```
def hello():
```

This is a function. However, it doesn't do anything. To make the function do something, you have to write the Python code that tells it what to do on subsequent lines. To ensure that the new code is "inside" the function, indent each of those lines.


REMEMBER

Indentations count big time in Python. There is no command that marks the end of a function. All indented lines below the `def` line are part of that function. The first un-indented line (indented as far out as the `def` line) is outside the function.

To make this function do something, you need to put an indented line of code under the `def`. We'll start easy by just having the function say `hello`. So, type **`print('Hello')`** indented under the `def` line. Now your code looks like this:

```
def hello():
    print('Hello')
```

If you run the code now, nothing will happen. That's okay. Nothing should happen because the code inside a function isn't executed until the functioned is *called*. You call your own functions the same way you call built-in functions, by writing code that calls the function by name, including the parentheses at the end.

For example, if you're following along, press Enter to add a blank line and then type `hello()` (no spaces in there) and make sure it's *not* indented. (You don't want this code to be indented because it's *calling* the function to execute its code; it's not *part of* the function.) So it looks like this:

```
def hello():
    print('Hello')

hello()
```

Still, nothing happens if you're in a Jupyter cell or a .py file because you've only typed in the code so far. For anything to happen, you have to run the code in the usual way in Jupyter or VS Code (if you're using a .py file in VS Code). When the code executes, you should see the output, which is just the word `Hello`, as shown in Figure 5-1.

FIGURE 5-1: Writing, and calling, a simple function named hello().

Commenting a Function

Comments are always optional in code. But it's somewhat customary to make the first line under the `def` statement a docstring (text enclosed in triple quotation marks) that describes what the function does. It's also fairly common to put a comment, preceded by a # sign, to the right of the parentheses in the first line. Here's an example using the simple `hello()` function:

```
def hello():   # Practice function
    """ A docstring describing the function """
    print('Hello')
```

Because they're just comments, they don't have any effect on what the code does. Running the code again displays exactly the same results. This is because comments are just notes to yourself or to programming team members describing what the code is about.

As an added bonus for VS Code users, when you start typing the function name, VS Code's IntelliSense help shows the `def` statement for your custom function as well as the docstring you typed for it, as in Figure 5-2. So you get to create your own custom help for your own custom functions.

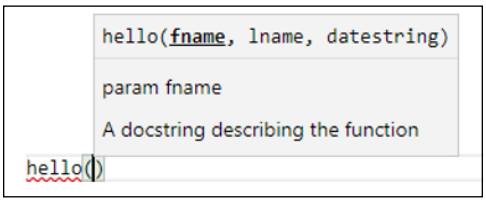


FIGURE 5-2: The docstring comment for your functions appears in VS Code IntelliSense help.

Passing Information to a Function

You can pass information to functions for them to work on. To do so, enter a parameter name for each piece of information you'll be passing into the function. A parameter name is the same as a variable name. You can make it up yourself. Use lowercase letters, no spaces or punctuation. Ideally, the parameter should describe what's being passed in, for code readability, but you can use generic names like `x` and `y`, if you prefer.

Any name you provide as a parameter is local only to that function. For example, if you have a variable named `x` outside the function, and another variable named `x` that's inside the function, and changes you make to `x` inside the function won't affect the variable named `x` that's outside the function.

The technical term for the way variables work inside functions is *local scope*, meaning the scope of the variables' existence and influence stays inside the function and extends out no further. Variables that are created and modified inside a function literally cease to exist the moment the function stops running, and any variables that were defined outside to the function are unaffected by the goings-on inside the function. This is a good thing, because when you're writing a function, you don't have to worry about accidentally changing a variable outside the function that happens to have the same name.

TECHNICAL STUFF

A function can *return* a value, and that returned value *is* visible outside the function. More on how all that works in a moment.

As an example, let's say you want the `hello` function to say `hello` to whoever is using the app (and you have access to that information in some variable). To pass the information into the function and use it there:

- Put a name inside the function's parentheses to act as a placeholder for the incoming information.
- Inside the function, use that exact same name to work with the information that's passed in.

For example, suppose you want to pass a person's name into the `hello` function, and then use it in the `print()` statement. You could use any generic name for both, like this:

```
def hello(x):   # Practice function
    """ A docstring describing the function """
    print('Hello ' + x)
```

Generic names don't exactly help make your code easy to understand. So you'd probably be better off using a more descriptive name, such as `name` or even `user_name`, as in the following:

```
def hello(user_name):   # Practice function
    """ A docstring describing the function """
    print('Hello ' + user_name)
```

TIP

In the `print()` function, we added a space after the *o* in *Hello* so there'd be a space between `Hello` and the name in the output.

When a function has a parameter, you have to pass it a value when you call it or it won't work. For example, if you added the parameter to the `def` statement and still tried to call the function without the parameter, as in the code below, running the code would produce an error:

```
def hello(user_name):   # Practice function
    """ A docstring describing the function """
    print('Hello ' + user_name)

hello()
```

The exact error would read something like `hello() missing 1 required positional argument: 'user_name'`, which is a major nerd-o-rama way of saying the `hello` function expected something to be passed into it.

For this particular function, a string needs to be passed. We know this because we concatenate whatever is passed into the variable to another string (the word `hello` followed by a space). If you try to concatenate a number to a string, you get an error.

The value you pass in can be a literal (the exact date you want to pass in), or it can be the name of a variable that contains that information. For example, when you run this code:

```
def hello(user_name):   # Practice function
    """ A docstring describing the function """
    print('Hello ' + user_name)

hello('Alan')
```

. . . the output is `Hello Alan` because you passed in `Alan` as a string in `hello('Alan')` so the name `Alan` is included in the output.

You can use a variable to pass in data too. For example, in the following code we store the string `"Alan"` in a variable named `this_person`. Then we call the

function using that variable name. So running that code produces `Hello Alan`, as shown in Figure 5-3.

```
def hello(user_name):   # Practice function
    """ A docstring describing the function """
    print('Hello ' + user_name)

# Put a string in a variable named this_person.
this_person = 'Alan'
# Pass that variable name to the function.
hello(this_person)
```

```
Hello Alan
```

FIGURE 5-3: Passing data to a function via a variable.

Defining optional parameters with defaults

In the previous section we mention that when you call a function that "expects" parameters without passing in those parameters, you get an error. That's was a little bit of a lie. You *can* write a function so that passing in a parameter is optional, but you have to tell it what to use if nothing gets passed in. The syntax for that is:

```
def functionname(paremetername = defaultvalue):
```

The only thing that's really different is the `= defaultvalue` part after the parameter name. For example, you could rewrite our sample `hello()` function with a default value, like this:

```
def hello(user_name =  'nobody'):   # Practice function
    """ A docstring describing the function """
    print('Hello ' + user_name)
```

Figure 5-4 shows the function after making that change. That code also called the function, both with a parameter and without. As you can see in the output, there's no error. The first call displays `hello` followed by the passed-in name. The empty `hello()` call to the functions executes the function with the default user name, `nobody`, so the output is `Hello nobody`.

```
def hello(user_name = 'nobody'):   # Practice function
    """ A docstring describing the function """
    print('Hello ' + user_name)

hello('Alan')
hello()
```

```
Hello Alan
Hello nobody
```

FIGURE 5-4: An optional parameter with a default value added to the `hello()` function.

Passing multiple values to a function

So far in all our examples we've passed just one value to the function. But you can pass as many values as you want. Just provide a parameter name for each value, and separate the names with commas.

For example, suppose you want to pass the user's first name, last name, and maybe a date into the function. You could define those three parameters like this:

```
def hello(fname, lname, datestring):   # Practice function
    """ A docstring describing the function """
    print('Hello ' + fname + ' ' + lname)
    print('The date is ' + datestring)
```

Notice that none of the parameters is optional. So when calling the function, you need to pass in three values, like this:

```
hello('Alan', 'Simpson', '12/31/2019')
```

Figure 5-5 shows this code and the output from passing in three values.

```
def hello(fname, lname, datestring):   # Practice function
    """ A docstring describing the function """
    msg = "Hello " + fname + " " + lname
    msg += " you mentioned " + datestring
    print(msg)

hello('Alan', 'Simpson', '12/31/2019')
```

```
Hello Alan Simpson you mentioned 12/31/2019
```

FIGURE 5-5: The `hello` function with three parameters.

If you want to use some (but not all) optional parameters with multiple parameters, make sure the optional ones are the last ones entered. For example, consider the following, which would *not* work right:

```
def hello(fname, lname='unknown', datestring):
```

If you try to run this code with that arrangement, you get an error that reads something along the lines of `SyntaxError: non-default argument follows default argument`. This is trying to tell you that if you want to list both required parameters and optional parameters in a function, you have to put in all the required ones first (in any order). Then the optional parameters can be listed after that with their = signs (in any order). So this would work fine:

```
def hello(fname, lname, datestring=''):
    msg = 'Hello ' + fname + ' ' + lname
```

```
    if len(datestring) > 0:
        msg += ' you mentioned ' + datestring
    print(msg)
```

Logically the code inside the function says:

- Create a variable named `message` and put in `Hello` and the first and last name.
- If the `datestring` passed has a length greater than zero, add " you mentioned " and that `datestring` to that `msg` variable.
- Print whatever is in the `msg` variable at this point.

Figure 5-6 shows two examples of calling that version of the function. The first call passes three values, and the second call passes only two. Both work because that third parameter is optional. The output from the first call is the full output including the date. The second one omits the part about the date in the output.

```
def hello(fname, lname, datestring=''):
    msg = 'Hello ' + fname + ' ' + lname
    if len(datestring) > 0:
        msg += ' you mentioned ' + datestring
    print(msg)

hello('Alan', 'Simpson', '12/31/2019')
hello('Sammy', 'Schmeedledorp')
```

```
Hello Alan Simpson you mentioned 12/31/2019
Hello Sammy Schmeedledorp
```

FIGURE 5-6: Two required parameters and one optional parameter with default value.

Using keyword arguments (kwargs)

If you've ever looked at the official Python documentation at Python.org, you may have noticed they throw the term *kwargs* around a lot. That's short for *keyword arguments* and is yet another way to pass data to a function.

The term *argument* is the technical term for "the value you are passing to a function's parameters." So far we've used strictly positional arguments. For example, consider these three parameters:

```
def hello(fname, lname, datestring=''):
```

When you call the function like this:

```
hello("Alan", "Simpson")
```

Python "assumes" `"Alan"` is the first name, because it's the first argument passed and `fname` is the first parameter in the function. `"Simpson"`, the second argument, is assumed to be `lname` because `lname` is the second parameter in the `def` statement. The `datestring` is assumed to be empty because `datestring` is the third parameter in the `def` statement and nothing is being passed as a third parameter.

It doesn't have to be this way. If you like, you can tell the function what's what by using the syntax *`parameter = value`* in the code that's calling the function. For example, take a look at this call to `hello`:

```
hello(datestring='12/31/2019', lname='Simpson', fname='Alan')
```

When you run this code, it works fine even though the order of the arguments passed in doesn't match the order of the parameter names in the `def` statement. But the order doesn't matter here because the parameter that each argument goes with is included with the call. Clearly the `'Alan'` argument goes with the `fname` parameter because `fname` is the name of the parameter in the `def` statement.

It works if you pass in variables too. Again, the order doesn't matter. Here's another way to do this:

```
attpt_date = '12/30/2019'
last_name = 'Janda'
first_name = 'Kylie'
hello(datestring=apt_date, lname=last_name, fname=first_name)
```

Figure 5-7 shows the result of running the code both ways. As you can see, it all works fine because there's no ambiguity about which argument goes with which parameter, because the parameter name is specified in the calling code.

```
def hello(fname, lname, datestring):   # Practice function
    """ A docstring describing the function """
    msg = "Hello " + fname + " " + lname
    msg += " you mentioned " + datestring
    print(msg)

# Pass in in literal kwargs (identify each by parameter name)
hello(datestring='12/31/2019', lname='Simpson', fname='Alan')

# Pass in in kwargs from variables (identify each by parameter name)
appt_date = '12/30/2019'
last_name = 'Janda'
first_name = 'Kylie'
hello(datestring=appt_date, lname=last_name, fname=first_name)
```

```
Hello Alan Simpson you mentioned 12/31/2019
Hello Kylie Janda you mentioned 12/30/2019
```

FIGURE 5-7: Calling a function with keyword arguments (kwargs).

Passing multiple values in a list

So far we've been passing one piece of data at a time. But you can also pass iterables to a function. Remember an iterable is anything that Python can loop through to get values. A list is a simple and perhaps the most commonly used literal. It's just a bunch of comma-separated values enclosed in square brackets.

The main trick to working with lists is that, if you want to alter the list contents in any way (for example, by sorting the list contents), you need to make a copy of the list within the function and make changes to that copy. This is because the function doesn't receive the original list in a mutable (changeable) format, it just receives a "pointer" to the list, which is basically a way of telling it where the list is so it can get its contents. It can certainly get those contents, but it can't change them in that original list.

It's not a huge problem because you can sort any list using the simple `sort()` method (or `sort(reverse=True)`, if you want to sort in descending order). For example, here is a new function named `alphabetize()` that takes one argument called `names`. Inside this function is assumed to be a list of values. So the function can alphabetize a list of any number of words or names:

```
def alphabetize(original_list=[]):
    """ Pass any list in square brackets, displays a string with items
   sorted """
    # Inside the function make a working copy of the list passed in.
    sorted_list = original_list.copy()
    # Sort the working copy.
    sorted_list.sort()
    # Make a new empty string for output
    final_list = ''
    # Loop through sorted list and append name and comma and space.
    for name in sorted_list:
        final_list += name + ', '
    # Knock of last comma space if final list is long enough
    final_list = final_list[:-2]
    # Print the alphabetized list.
    print(final_list)
```

We'll walk you through this function to explain what's going on. The first line defines the function. You'll notice that for the parameter we put `original_list=[]`. The default value (`=[]`) is optional, but we put it there for two reasons. For one, so that if you accidentally call it without passing in a list, it doesn't "crash." It just returns an empty list. The docstring provides additional information. For example, when you start to type the function name in VS Code, you get both the `def` statement and the docstring as IntelliSense help to remind you how to use the function, as in Figure 5-8.

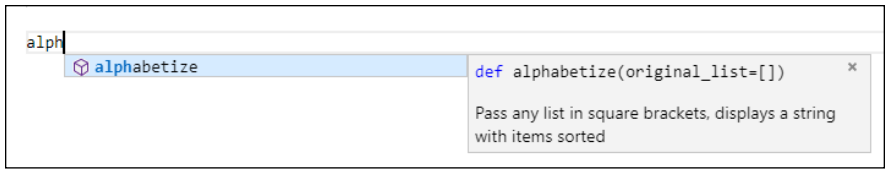


FIGURE 5-8: Using the alphabetize function in VS Code.

Because the function can't alter the list directly, it first makes a copy of the original list (the one that was passed in) in a new list called `sorted_list`, with this line of code:

```
    sorted_list = original_list.copy()
```

At this point, the `sorted_list` isn't really sorted, it's still just a copy of the original. The next line of code does the actual sorting:

```
sorted_list.sort()
```

This function actually creates a string with the sorted items separated by commas. So this line creates a new variable name, `final_list`, and starts it off as an empty string after the = sign (that's two single quotation marks with no space in between).

```
final_list = ''
```

This loop loops through the sorted list and adds each item from the list, separated by a comma and a space, to that `final_list` string:

```
    for name in sorted_list:
        final_list += name + ', '
```

When that's done, if anything was added to `final_list` at all, it will have an extra comma and a space at the end. So this statement removes those last two characters, assuming the list is at least two characters in length:

```
final_list = final_list[:-2]
```

The next statement just prints the `final_list` so you can see it.

To call the function, you can pass a list right inside the parentheses of the function, like this:

```
alphabetize(['Schrepfer', 'Maier', 'Santiago', 'Adams'])
```

As always, you can also pass in the name of a variable that already contains the list, as in this example.

```
names = ['Schrepfer', 'Maier', 'Santiago', 'Adams']
alphabetize(names)
```

Either way, the function shows those names in alphabetical order, as follows.

```
Adams, Maier, Santiago, Schrepfer
```

Passing in an arbitrary number of arguments

A list provides one way of passing a lot of values into a function. You can also design the function so that it accepts any number of arguments. It's not particularly faster or better, so there's no "right time" or "wrong time" to use this method. Just use whichever seems easiest to you, or whichever seems to make the most sense at the moment. To pass in any number of arguments, use `*args` as the parameter name, like this:

```
def sorter(*args)
```

Whatever you pass in becomes a tuple named `args` inside the function. A tuple is an immutable list (a list you can't change). So again, if you want to change things, you need to copy the tuple to a list and then work on that copy. Here is an example where the code uses the simple statement `newlist = list(args)` (you can read that as *the variable named newlist is a list of all the things that are in the args tuple*). The next line, `newlist.sort()` sorts the list, and the `print` displays the contents of the list:

```
def sorter(*args):
    """ Pass in any number of arguments separated by commas
    Inside the function, they treated as a tuple named args """
    # The passed-in
    # Create a list from the passed-in tuple
    newlist = list(args)
    # Sort and show the list.
    newlist.sort()
    print(newlist)
```

Figure 5-9 shows an example of running this code with a series of numbers as arguments in a Jupyter cell. As you can see, the resulting list is in sorted order, as expected.

```
def sorter(*args):
    """ Pass in any number of arguments separated by commas
    Inside the function, they treated as a tuple named args """
    # The passed-in
    # Create a list from the passed-in tuple
    newlist = list(args)
    # Sort and show the list.
    newlist.sort()
    print(newlist)

sorter(1, 0.001, 100000,-900,  2)
```

```
[-900, 0.001, 1, 2, 100000]
```

FIGURE 5-9: A function accepting any number of arguments with `*args`.

Returning Values from Functions

So far, all our functions have just displayed some output on the screen so you can make sure the function works. In real life, it's more common for a function to *return* some value and put it in a variable that was specified in the calling code. This is typically the last line of the function followed by a space and the name of the variable (or some expression) that contains the value to be returned.

Here is a variation on the `alphabetize` function. It contains no `print` statement. Instead, at the end, it simply returns the `final_list` that the function created:

```
def alphabetize(original_list=[]):
    """ Pass any list in square brackets, displays a string with items
    sorted """
    # Inside the function make a working copy of the list passed in.
    sorted_list = original_list.copy()
    # Sort the working copy.
    sorted_list.sort()
    # Make a new empty string for output
    final_list = ''
    # Loop through sorted list and append name and comma and space.
    for name in sorted_list:
        final_list += name + ', '
    # Knock of last comma space if final list is long enough
    final_list = final_list[:-2]
    # Return the alphabetized list.
    return final_list
```

The most common way to use functions is to store whatever they return into some variable. For example, in the following code, the first line defines a variable called `random_list`, which is just a list containing names in no particular order, enclosed in square brackets (which is what tells Python it's a list). The second line creates a new variable named `alpha_list` by passing `random_list` to the

alphabetize() function and storing whatever that function returns. The final print statement displays whatever is in the alpha_list variable:

```
random_list = ['McMullen', 'Keaser', 'Maier', 'Wilson', 'Yudt', 'Gallagher',
   'Jacobs']
alpha_list = alphabetize(random_list)
print(alpha_list)
```

Figure 5-10 shows the result of running the whole kit-and-caboodle in a Jupyter cell.

```
def alphabetize(original_list=[]):
    """ Pass any list in square brackets, displays a string with items sorted """
    # Inside the function make a working copy of the list passed in.
    sorted_list = original_list.copy()
    # Sort the working copy.
    sorted_list.sort()
    # Make a new empty string for output
    final_list = ''
    # Loop through sorted list and append name and comma and space.
    for name in sorted_list:
        final_list += name + ', '
    # Knock of last comma space if final list is long enough
    final_list = final_list[:-2]
    # Print the alphabetized list.
    print(final_list)

names = ['McMullen', 'Keaser', 'Maier', 'Wilson', 'Yudt', 'Gallagher', 'Jacobs']
alphabetize(names)
```

```
Gallagher, Jacobs, Keaser, Maier, McMullen, Wilson, Yudt
```

FIGURE 5-10: Printing a string returned by the alphabetize() function.

Unmasking Anonymous Functions

Python supports the concept of *anonymous functions*, also called *lambda functions*. The *anonymous* part of the name is based on the fact that the function doesn't need to have a name (but *can* have one if you want it to). The *lambda* part is based on the use of the keyword *lambda* to define them in Python. *Lambda* is also the 11th letter of the Greek alphabet. But the main reason that the name is used in Python is because the term *lambda* is used to describe anonymous functions in calculus. Now that we've cleared that up, you can use this info to spark enthralling conversation at office parties.

The minimal syntax for defining a lambda expression (with no name) is:

```
Lambda arguments : expression
```

When using it:

- Replace *arguments* with data being passed into the expression.
- Replace *expression* with an expression (formula) that defines what you want the lambda to return.

A fairly common example of using that syntax is when you're trying to sort strings of text where some of the names start with uppercase letters and some start with lowercase letters, as in these names:

```
Adams, Ma, diMeola, Zandusky
```

Suppose you write the following code to put the names into a list, sort it, and then print the list, like this:

```
names = ['Adams', 'Ma', 'diMeola', 'Zandusky']
names.sort()
print(names)
```

That output from this is:

```
['Adams', 'Ma', 'Zandusky', 'diMeola']
```

Having diMeola come after Zandusky seems wrong to me, and probably to you. But computers don't always see things the way we do. (Actually, they don't "see" anything because they don't have eyes or brains . . . but that's beside the point.) The reason diMeola comes after Zandusky is because the sort is based on ASCII, which is a system in which each character is represented by a number. All the lowercase letters have numbers that are higher than uppercase numbers. So, when sorting, all the words starting with lowercase letters come after the words that start with an uppercase letter. If nothing else, it at least warrants a minor *hmm*.

To help with these matters, the Python sort() method lets you include a `key=` expression inside the parentheses, where you can tell it how to sort. The syntax is:

```
.sort(key = transform)
```

The *transform* part is some variation on the data being sorted. If you're lucky and one of the built-in functions like `len` (for *length*) will work for you, then you can just use that in place of *transform*, like this:

```
names.sort(key=len)
```

Unfortunately for us, the length of the string doesn't help with alphabetizing. So when you run that, the order turns out to be:

```
['Ma', 'Adams', 'diMeola', 'Zandusky']
```

The sort is going from the shortest string (the one with the fewest characters) to the longest string. Not helpful at the moment.

You can't write `key=lower` or `key=upper` to base the sort on all lowercase or all uppercase letters either, because `lower` and `upper` aren't built-in functions (which you can verify pretty quickly by googling *python 3.7 built-in functions*).

In lieu of a built-in function, you can use a custom function that you define yourself using `def`. For example, we can create a function named `lower()` that accepts a string and returns that string with all of its letters converted to lowercase. Here is the function:

```
def lower(anystring):
    """ Converts string to all lowercase """
    return anystring.lower()
```

The name `lower` is a name I made up, and *`anystring`* is a placeholder for whatever string you pass to it in the future. The `return anystring.lower()` returns that string converted to all lowercase using the `.lower()` method of the `str` (string) object.


TECHNICAL STUFF

You can't use `key=lower` in the `sort()` parentheses because `lower()` isn't a built-in function. It's a method . . . not the same. Kind of annoying with all these buzzwords, I know.

Suppose you write this function in a Jupyter cell or .py file. Then you call the function with something like `print(lowercaseof('Zandusky'))`. What you get as output is that string converted to all lowercase, as in Figure 5-11.

```
def lowercaseof(anystring):
    """ Converts string to all lowercase """
    return anystring.lower()

print(lowercaseof('Zandusky'))
```

```
zandusky
```

FIGURE 5-11: Putting a custom function named `lower()` to the test.

Okay, so now we have a custom function to convert any string to all lowercase letters. How do we use that as a sort key? Easy, use `key=`*`transform`* the same as before, but replace *`transform`* with your custom function name. Our function is

named `lowercaseof`, so we'd use `.sort(key= lowercaseof)`, as shown in the following:

```
def lowercaseof(anystring):
    """ Converts string to all lowercase """
    return anystring.lower()

names = ['Adams', 'Ma', 'diMeola', 'Zandusky']
names.sort(key=lowercaseof)
```

Running this code to display the list of names puts them in the correct order, because it based the sort on strings that are all lowercase. The output is the same as before because only the sorting, which took place behind the scenes, used lowercase letters. The original data is still in its original uppercase and lowercase letters.

```
'Adams', 'diMeola', 'Ma', 'Zandusky'
```

If you're still awake and conscious after reading all of this you may be thinking, "Okay, you solved the sorting problem. But I thought we were talking about lambda functions here. Where's the lambda function?" There is no lambda function yet. But this is a perfect example of where you *could* use a lambda function, because the function you're calling, `lowercaseof()`, does all of its work with just one line of code: `return anystring.lower()`.

When your function can do its thing with a simple one-line expression like that, you can skip the `def` and the function name and just use this syntax:

```
lambda parameters : expression
```

Replace *`parameters`* with one or more parameter names that you make up yourself (the names inside the parentheses after `def` and the function name in a regular function). Replace *`expression`* with what you want the function to return without the word `return`. So in this example the key, using a lambda expression, would be:

```
lambda anystring : anystring.lower()
```

Now you can see why it's an anonymous function. The whole first line with function name `lowercaseof()` has been removed. So the advantage of using the lambda expression is that you don't even need the external custom function at all. You just need the parameter followed by a colon and an expression that tells it what to return.

Figure 5-12 shows the complete code and the result of running it. You get the proper sort order without the need for a customer external function like `lowercaseof()`. You just use `anystring : anystring.lower()` (after the word `lambda`) as the sort key.

```
names = ['Adams', 'Ma', 'diMeola', 'Zandusky']
names.sort(key = lambda anystring : anystring.lower())
print(names)
```

```
['Adams', 'diMeola', 'Ma', 'Zandusky']
```

FIGURE 5-12: Using a lambda expression as a sort key.

I may also add that *anystring* is a longer parameter name than most Pythonistas would use. Python folks are fond of short names, even single-letter names. For example, you could replace *anystring* with *s* (or any other letter), as in the following, and the code will work exactly the same:

```
names = ['Adams', 'Ma', 'diMeola', 'Zandusky']
names.sort(key=lambda s: s.lower())
print(names)
```

Way back at the beginning of this tirade I mentioned that a `lambda` function doesn't have to be anonymous. You can give them names and call them as you would other functions.

For example, here is a `lambda` function named `currency` that takes any number and returns a string in currency format (that is, with a leading dollar sign, commas between thousands, and two digits for pennies):

```
currency = lambda n : f"${n:,.2f}"
```

Here is one named `percent` that multiplies any number you send to it by 100 and shows it with two a percent sign at the end:

```
percent = lambda n : f"{n:.2%}"
```

Figure 5-13 shows examples of both functions defined at the top of a Jupyter cell. Then a few `print` statements call the functions by name and pass some sample data to them. Each `print()` statements displays the number in the desired format.

The reason you can define those as single-line lambdas is because you can do all the work in one line, `f"${n:,.2f}"` for the first one and `f"{n:.2%}"` for the second one. But just because you *can* do it that way, doesn't mean you *must*. You could use regular functions too, as follows:

```
# Show number in currency format.
def currency(n):
    return f"${n:,.2f}"
```

```
def percent(n):
# Show number in percent format.
    return f"{n:.2%}"
```

```
# Show number in currency format.
currency = lambda n : f"${n:,.2f}"
# Show number in percent format.
percent = lambda n : f"{n:.2%}"

# Test currency function
print(currency(99))
print(currency(123456789.09876543))

# Test percent function
print(percent(0.065))
print(percent(.5))
```

```
$99.00
$123,456,789.10
6.50%
50.00%
```

FIGURE 5-13: Two functions for formatting numbers.

With this longer syntax, you could pass in more information too. For example, you may default to a right-aligned format within a certain width (say 15 characters) so all numbers came out right-aligned to the same width. Figure 5-14 shows this variation on the two functions.

```
# Show number in currency format, specify width.
def currency(n, w=15):
    """ Show in currency format, width = 15 or width of your choosing """
    s = f"${n:,.2f}"
    # Pad left of output with spaces to width of w.
    return s.rjust(w)

# Show number in percent format, specify width.
def percent(n, w=15):
    """ Show in percent format, width = 15 or width of your choosing """
    # Show number in percent format.
    s = f"{n:.1%}"
    # Pad left of output with spaces to width of w.
    return s.rjust(w)
```

FIGURE 5-14: Two functions for formatting numbers with fixed width.

In Figure 5-14, the second parameter is optional and defaults to 15 if omitted. So if you call it like this:

```
print(currency(9999))
```

. . . you get $9,999.00 padding with enough spaces on the left to make it 15 characters wide. If you call it like this instead:

```
print(currency(9999,20)
```

. . . you still get $9,999.00 but padded with enough spaces on the left to make it 20 characters wide.

TIP

The `.ljust()` used in Figure 5-14 is a Python built-in string method that pads the left side of a string with sufficient spaces to make it the specified width. There's also an `rjust()` method to pad the right side. You can also specify a character other than a space. Google *python 3 ljust rjust* if you need more info.

So there you have it, the ability to create your own custom functions in Python. In real life, what you want to do is, any time you find that you need access to the same chunk of code — the same bit of login — over and over again in your app, don't simply copy/paste that chunk of code over and over again. Instead, put all that code in a function that you can call by name. That way, if you decide to change the code, you don't have to go digging through your app to find all the places that need changing. Just change it in the function where it's all defined in one place.

IN THIS CHAPTER

- Mastering classes and objects
- Creating a class
- Initializing an object in a class
- Populating an object's attributes
- Giving a class methods
- Understanding class inheritance

Chapter 6
Doing Python with Class

In the previous chapter, we talk about functions, which allow you to compartmentalize chunks of code that do specific tasks. In this chapter, you learn about classes, which allow you to compartmentalize code *and* data. You discover all the wonder, majesty, and beauty of classes and objects (okay, maybe we oversold things a little there). But classes have become a defining characteristic of modern object-oriented programming languages like Python.

We're aware we threw a whole lot of techno jargon your way in the previous chapters. Don't worry. For the rest of this chapter we start off assuming that — like 99.9 percent of people in this world — you don't know a class from an object from a pastrami sandwich.

Mastering Classes and Objects

Object-oriented programming (OOP) has been a major buzzword in the computer world for at least a couple decades now. The term *object* stems from the fact that the model resembles objects in the real word in that each object is a thing that has certain attributes and characteristics that make it unique. For example, a chair is an object. In the world, there are lots of different chairs that differ in size, shape, color, and material. But they're all still chairs.

How about cars? We all recognize a car when we see one. (Well, usually.) Even though cars aren't all exactly the same, they all have certain *attributes* (year, make, model, color) that make each unique. They have certain *methods* in common, where a method is an action or a thing the car can do. For example, cars all have go, stop, and turn actions you can control pretty much the same in each one.

Figure 6-1 shows the concept where all cars (although not identical) have certain attributes and methods in common. In this case, you can think of the class "cars" as being sort of a factory that creates all cars. After it's created, each car is an independent object. Changing one car has no effect on the other cars or the `Car` class.

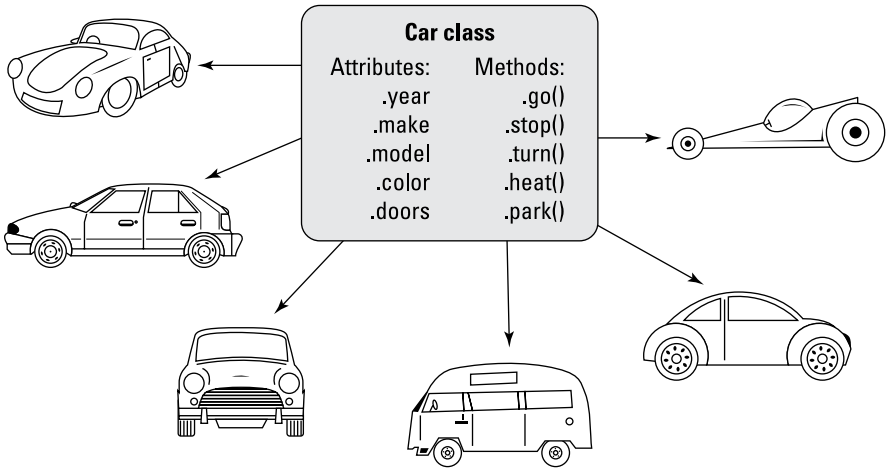


FIGURE 6-1: Different car objects.

If the factory idea doesn't work for you, think of a class as type of blueprint. For instance, take dogs. No, there is no physical blueprint for creating dogs. But there is some dog DNA that pretty much does the same thing. The dog DNA can be considered a type of blueprint (like a Python class) from which all dogs are created. Dogs vary in the attributes like breed, color, and size, but they share certain behaviors (methods) like "eat" and "sleep." Figure 6-2 shows an example where there is a class of animal called `Dog` from which all dogs originate.

Even people can easily be viewed as objects in this manner. In your code, you could create a `Member` class with which you manage your site members. Each member would have certain attributes, such as a username, full name, and so forth. You could also have methods such as `.archive()` to deactivate an account and `.restore()` to reactivate an account. The `.archive()` and `.restore()`

methods are behaviors that let you control membership, in much the same way the accelerator, brake, and steering wheel allow you to control a car. Figure 6-3 shows the concept.

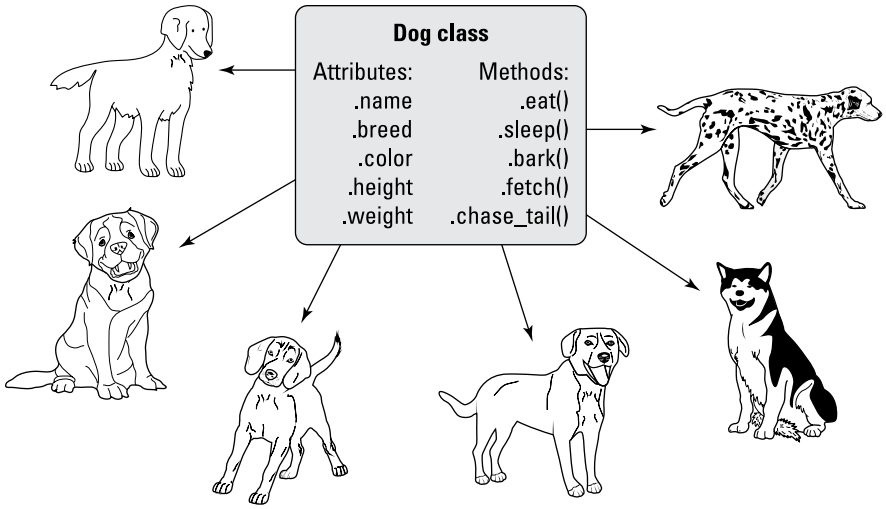


FIGURE 6-2: The Dog class creates many unique dogs.

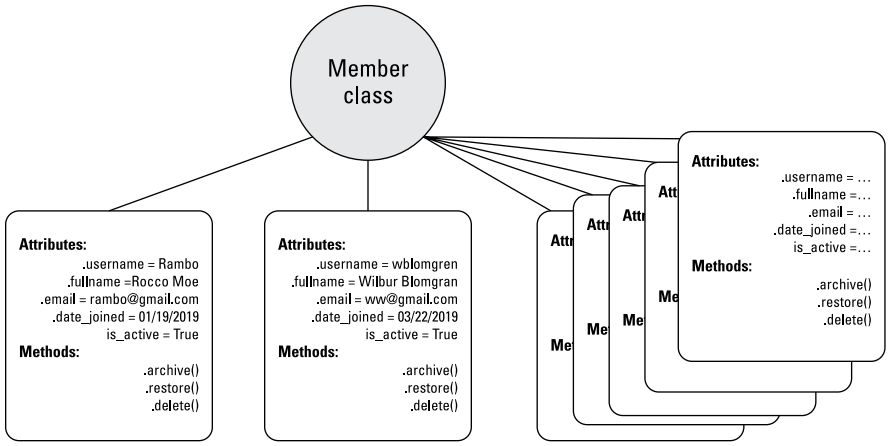


FIGURE 6-3: The `Member` class and member instances.

The main point is that each instance of a class is its own independent object with which you can work. Changing one instance of a class has no effect on the class or on other instances, just as painting one car a different color has no effect on the car factory or on any other cars produced by that factory.

All this business of classes and instances stems from a type of programming called object-oriented programming (OOP for short). It's been a major concept in the biz for a few decades now. Python, like any significant, serious, modern programming language is object-oriented. The main buzzwords you need to get comfortable with are the ones I've harped on in the last few paragraphs:

- **Class:** A piece of code from which you can generate a unique object, where each object is a single instance of the class. Think of it as a blueprint or factory from which you can create individual objects.
- **Instance:** One unit of data plus code generated from a class as an instance of that class. Each instance of a class is also called an *object* just like all the different cars are objects, all created by some car factory (class).
- **Attribute:** A characteristic of an object that contains information about the object. Also called a *property* of the object. An attribute name is preceded by dot, as in `member.username` which may contain the username for one site member.
- **Method:** A Python function that's associated with the class. It defines an action that object can perform. In an object, you call a method by preceding the method name with a dot, and following it with a pair of parentheses. For example `member.archive()` may be a method that archives (deactivates) the member's account.

Creating a Class

You create your own classes like you create your own functions. You are free to name the class whatever you want, so long as it's a legitimate name that starts with a letter and contains no spaces or punctuation. It's customary to start a class name with an uppercase letter to help distinguish classes from variables. To get started, all you need is the word `class` followed by a space, a class name of your choosing, and a colon. For example, to create a new class named Member, use `class Member:`

To make your code more descriptive, feel free to put a comment above the class definition. You can also put a docstring below the class line, which will show up whenever you type the class name in VS Code. For example, to add comments for your new `Member` class, you might type up the code like this:

```
# Define a new class name Member.
class Member:
    """ Create a new member. """
```

EMPTY CLASSES

If you start a class with `class` *name*: and then run your code before finishing the class, you'll actually get an error. To get around that, you can tell Python that you're just not quite ready to finish writing the class by putting the keyword `pass` below the definition, as in the following code:

```
# Define a new class name Member.
class Member:
    pass
```

In essence, what you're doing there is telling Python "Hey I know this class doesn't really work yet, but just let it pass and don't throw an error message telling me about it."

That's it for defining a new class. However, it isn't useful until you specify what attributes you want each object that you create from this class to inherit from the class.

How a Class Creates an Instance

To grant to your class the ability to create instances (objects) for you, you give the class an *init method*. The word *init* is short for *initialize*. As a method, it's really just a function that's defined inside of a class. But it must have the specific name `__init__` — that's two underscores followed by `init` followed by two more underscores.

TIP

That `__init__` is sometimes spoken as "*dunder init.*" The *dunder* part is short for *d*ouble *under*line.

The syntax for creating an `init` method is:

```
def __init__(self[, suppliedprop1, suppliedprop2,...])
```

The `def` is short for *define*, and `__init__` is the name of a built-in Python method that's capable of creating objects from within a class. The *`self`* part is just a variable name, and it is used to refer to the object being created at the moment. You can use the name of your own choosing instead of *`self`*. But *`self`* would be considered by most a good "best practice" since it's explanatory and customary.

This whole businesses of classes is easier to learn and understand if you start off simply. So, for a working example, let's create a class named `Member`, into which you will pass a username (`uname`) and full name (`fname`) whenever you want to create a member. As always, you can precede the code with a comment. You can also put a docstring (in triple quotation marks) under the first line both as a comment but also as an IntelliSense reminder when typing code in VS Code:

```
# Define a class named Member for making member objects.
class Member:
    """ Create a member from uname and fname """
    def __init__(self, uname, fname):
```

When that `def __init__` line executes, you have an empty object, named `self`, inside of the class. The `uname` and `fname` parameters just hold whatever data you pass in, and you'll see how that works in a moment.

An empty object with no data doesn't do you much good. What makes an object useful is the information it contains that's unique to that object (its attributes). So, in your class, the next step is to assign a value to each of the object's attributes.

Giving an Object Its Attributes

Now that you have a new, empty `Member` object, you can start giving it attributes and *populate* (store values in) those attributes. For example, let's say you want each member to have a `.username` attribute that contains the user's username (perhaps for logging in). You have a second attribute named `fullname`, which is the member's full name. To define and populate those attributes use

```
self.username = uname
self.fullname = fname
```

The first line creates an attribute named `username` for the new instance (`self`) and puts into it whatever was passed into the `uname` attribute when the class was called. The second line creates an attribute named `fullname` for the new `self` object, and puts into it whatever was passed in as the `fname` variable. Add in a comment and the whole class may look like this:

```
# Define a new class name Member.
class Member:
    """ Create a new member. """
    def __init__(self, uname, fname):
        # Define attributes and give them values.
```

```
        self.username = uname
        self.fullname = fname
```

So do you see what's happening there? The `__init__` line creates a new empty object named `self`. Then the `self.username = uname` line adds an attribute named `username` to that empty object, and puts into that attribute whatever was passed in as `uname`. Then the `self.fullname = fname` line does the same thing for the `fullname` attribute and the `fname` value that was passed in.

The convention for naming things in classes suggests using an initial cap for the class name, but attributes should follow the standard for variables, being all lowercase with an underscore to separate words within the name.

Creating an instance from a class

When you have created the class, you can create instances (objects) from it using this simple syntax:

```
this_instance_name = Member('uname', 'fname')
```

Replace *this_instance_name* with a name of your own choosing (in much the same way you may name a dog, who is an instance of the `dog` class). Replace *uname* and *fname* with the username and full name you want to put into the object that will be created. Make sure you don't indent that code; otherwise, Python will think that new code still belongs to the class's code. It doesn't. It's new code to test the class.

So, for the sake of example, let's say you want to create a member named `new_guy` with the username `Rambo` and the full name `Rocco Moe`. Here's the code for that:

```
new_guy = Member('Rambo','Rocco Moe')
```

If you run this code and don't get any error messages, then you know it at least ran. But to make sure, you can print the object or its attributes. So to see what's really in the `new_guy` instance of `Members`, you can print it as a whole. You can also print just its attributes, `new_guy.username` and `new_guy.fullname`. You can also print `type(new_guy)` to ask Python what "type" `new_guy` is. Here is that code:

```
print(new_guy)
print(new_guy.username)
print(new_guy.fullname)
print(type(new_guy))
```

Figure 6-4 shows all the code and the result of running it in a Jupyter cell.

```
: # Define a new class name Member.
  class Member:
      """ Create a new member. """
      def __init__(self, uname, fname):
          # Define attributes and give them values.
          self.username = uname
          self.fullname = fname

  # The class ends at the first un-indented line.

  # Create an instance of the Member class named new_guy
  new_guy = Member('Rambo','Rocco Moe')

  #See what's in the instance, as well as its individual properties.
  print(new_guy)
  print(new_guy.username)
  print(new_guy.fullname)
  print(type(new_guy))

  <__main__.Member object at 0x000002175EA2E160>
  Rambo
  Rocco Moe
  <class '__main__.Member'>
```

FIGURE 6-4 Creating a member from the `Member` class in a Jupyter cell.

In the figure you can see that the first line of output is this:

```
<__main__.Member object at 0x000002175EA2E160>
```

This is telling you that `new_guy` is an object created from the `Member` class. The number at the end is its location in memory, but don't worry about that; you won't need to know about those right now.

The next three lines of output are

```
Rambo
Rocco Moe
<class '__main__.Member'>
```

The `Rambo` line is `new_guy`'s `username` (`new_guy.username`), the `Rocco Moe` is `new_guy`'s `full name` (`new_guy.fullname`). The `type is <class '__main__.Member'>`, which again is just telling you that `new_guy` is an instance of the `Member` class.

WARNING

Much as we hate to put any more burden on your brain cells right now, the words *object* and *property* are synonymous with *instance* and *attribute*. The `new_guy` instance of the `Member` class can also be called an object, and the `fullname` and `username` attributes of `new_guy` are also properties of that object.

Admittedly it can be a little difficult to wrap your head around this at first, but it's really quite simple: An object is just a handy way to compartmentalize information about an item that's similar to other items (like all dogs are dogs and all cars are cars). What makes the item unique is its attributes, which won't necessarily

all be the same as the attributes of other objects of the same type, in much the same way that not all dogs are the same breed and not all cars are the same color.

We intentionally used `uname` and `fname` as parameter names to distinguish them from the attribute names `username` and `fullname`. However, this isn't a requirement. In fact, if anything, people tend to use the same names for the parameters as they do for the attributes.

Instead of `uname` for the placeholder, you can use `username` (even though it's the same as the attribute name). Likewise, you can use `fullname` in place of `fname`. Doing so won't alter how the class behaves. You just have to remember that the same name is being used in two different ways, first as a placeholder for data being passed into the class, and then later as an actual attribute name that gets its value from that passed-in value.

Figure 6-5 shows the same code as Figure 6-4 with `uname` replaced with `username` and `fname` replaced with `fullname`. Running the code produces exactly the same output as before; using the same name for two different things didn't bother Python one bit.


```
# Define a new class name Member.
class Member:
    """ Create a new member. """
    def __init__(self, username, fullname):
        # Define attributes and give them values.
        self.username = username
        self.fullname = fullname

# The class ends at the first un-indented line.

# Create an instance of the Member class named new_guy
new_guy = Member('Rambo','Rocco Moe')

#See what's in the instance, as well as its individual properties.
print(new_guy)
print(new_guy.username)
print(new_guy.fullname)
print(type(new_guy))
```

```
<__main__.Member object at 0x000002175EA2E240>
Rambo
Rocco Moe
<class '__main__.Member'>
```

FIGURE 6-5: The `Member` class with `username` and `fullname` for parameters and attributes.

After you type a class name and the opening parenthesis in VS Code, its IntelliSense shows you the syntax for parameters and the first docstring in the code, as shown in Figure 6-6. Naming things in a way that's meaningful to you and including a descriptive docstring in the class makes it easier for you to remember how to use the class in the future.

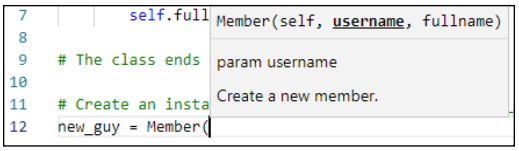


FIGURE 6-6: VS Code displays help when accessing your own custom classes.

Changing the value of an attribute

When working with tuples, you can define *key*:*value* pairs, much like the *attribute*:*value* pairs you see here with instances of a class. There is one major difference, though: Tuples are immutable, meaning that after they're defined, you code can't change anything about them. This is not true with objects. After you create an object, you can change the value of any attribute at any time using the following simple syntax:

```
objectname.attributename = value
```

Replace *objectname* with the name of the object (which you've already created via the class). Replace *attributename* with the name of the attribute whose value you want to change. Replace *value* with the new value.

Figure 6-7 shows an example in which, after initially creating the `new_guy` object, the following line of code executes:

```
new_guy.username = "Princess"
```

The lines of output under that show that `new_guy`'s `username` has indeed been changed to `Princess`. His full name hasn't changed because you didn't do anything to that in your code.

Defining attributes with default values

You don't have to pass in the value of every attribute for a new object. If you're always going to give those some default value at the moment the object is created, you can just use `self.`*attributename* = *value*, the same as before, in which *attributename* is some name of your own choosing. The *value* can be some value you just set, such as `True` or `False` for a Boolean, or today's date, or anything that can be calculated or determined by Python without your telling it the value.

For example, let's say that whenever you create a new membe, you want to track the date that you created that member in an attribute named `date_joined`. And you want the ability to activate and deactivate accounts to control user logins. So you create an attribute named `is_active`. Let's suppose that you want to start off a new member with that set to `True`.

```
# Define a new class name Member.
class Member:
    """ Create a new member. """
    def __init__(self, username, fullname):
        # Define attributes and give them values.
        self.username = username
        self.fullname = fullname

# The class ends at the first un-indented line.

# Create an instance of the Member class named new_guy
new_guy = Member('Rambo','Rocco Moe')

# See what's in the instance, as well as its individual properties.
print(new_guy.username)
print(new_guy.fullname)
print() #This just prints a blanks line.

# Change new_guy's user name then print both attributes again.
new_guy.username = "Princess"
print(new_guy.username)
print(new_guy.fullname)
```

```
Rambo
Rocco Moe

Princess
Rocco Moe
```

FIGURE 6-7: Changing the value of an object's attribute.

If you're going to be doing anything with dates and times, you'll want to import the `datetime` module, so put that at the top of your file, even before the `class Member:` line. Then you can add these lines before or after the other lines that assign values to attributes within the class:

```
        self.date_joined  = dt.date.today()
        self.is_active = True
```

Here is how you could add the `import` and those two new attributes to the class:

```
import datetime as dt

# Define a new class name Member.
class Member:
    """ Create a new member. """
    def __init__(self, username, fullname):
        # Define attributes and give them values.
        self.username = username
        self.fullname = fullname

        # Default date_joined to today's date.
        self.date_joined =  dt.date.today()
        # Set is active to True initially.
        self.is_active = True
```

WARNING

If you forget to import `datetime` at the top of the code, you'll get an error message when you run the code telling you it doesn't know what `dt.date.today()` means. Just add the `import` line to the top of the code and try again.

PERSISTING CHANGES TO DATA

Maybe you're wondering what the point is of creating all these different classes and objects if everything just ceases to exist the moment the program ends. What does it mean to create a "member" if you can't store that information "forever" and use it to control members logging into a website or whatever?

Truthfully, it doesn't do you any good . . . by itself. However, all the data you create and manage with classes and objects can be *persisted* (retained indefinitely) and be at your disposal at any time by storing that data in some kind of external file, usually a database.

We get to that business of persisting data in Book 3 of this book. But first you really need to learn the core Python basics because it's pretty much impossible to understand how any of *that* works if you don't already understand how all of this (the stuff we're talking about in this book) works.

There is no need to pass any new data into the class for the `date_joined` and `is_active` attributes, because those can be determined by the code.

Note that a default value is just that: It's a value that gets assigned automatically when you first create the object. But that doesn't mean it can't be changed. You can change those values the same as you would change any other attribute's value using the syntax

```
objectname.attributename = value
```

For example, suppose you use the `is_active` attribute to determine whether a user is active and can log into your site. If a member turns out to be an obnoxious troll and you don't want him logging in anymore, you could just change the `is_active` attribute to False like this:

```
newmember.is_active = False
```

Giving a Class Methods

Any object you define can have any number of attributes, each given any name you like, in order to store information *about* the object, like a dog's breed and color or a car's make and model. You can also define you own methods for any object, which are more like behaviors than facts about the object. For example, a dog can eat, sleep, and bark. A car can go, stop, and turn. A method is really just a function,

like you learned about in the previous chapter. What makes it a method is the fact that it's associated with a particular class and with each specific object you create from that class.

Method names are distinguished from attribute names for an object by the pair of parentheses that follow the name. To define what the methods will be in your class, use this syntax for each method:

```
def methodname(self[, param1, param2, ...])
```

Replace *methodname* with a name of your choosing (all lowercase, no spaces). Keep the word `self` in there as a reference to the object being defined by the class. You can also pass in parameters after `self` using commas, as with any other function, but this is entirely optional. Never type the square brackets (`[]`). They're shown here in the syntax only to indicate that parameter names after `self` are allowed but not required.

Let's create a method named `.show_date_joined()` that returns the user's name and date joined in a formatted string. Here is how you could write that code to define this method:

```
# Define methods as functions, use self for "this" object.
def show_datejoined(self):
    return f"{self.fullname} joined on {self.date_joined:%m/%d/%y}"
```

The name of the method is `show_datejoined`. The task of this method, when called, is to simply put together some nicely formatted text containing the display the member's full name and date joined. Make sure you indent the `def` at the same level as the first `def`, as these cannot be indented under attributes.

To call the method from your code, use this syntax:

```
objectname.methodname()
```

Replace *objectname* with the name of the object to which you're referring. Replace *methodname* with the name of the method you want to call. Include the parentheses (no spaces) and leave them empty if inside the class the only parameter between the parentheses is the `self` parameter. Figure 6-8 shows a complete example.

Notice in Figure 6-8 how the `show_datejoined()` method is defined within the class. Its `def` is indented to the same level of the first `def`. The code that the method executes is indented under that. Outside the class, `new_guy = Member('romo', 'Rocco Moe')` creates a new member named `new_guy`. Then `new_guy.show_datejoined()` executes the `show_datejoined()` method, which in turn displays `Rocco Moe joined 12/06/18`, the day I ran the code.

```
import datetime as dt

# Define a new class name Member.
class Member:
    """ Create a new member. """
    def __init__(self, username, fullname):
        # Define attributes and give them values.
        self.username = username
        self.fullname = fullname
        # Default date_joined to today's date.
        self.date_joined =  dt.date.today()
        # Set is active to True initially.
        self.is_active = True

    # Methods for each instance created (instance methods)

    # A method to return a formatted string with showing date joined.
    def show_datejoined(self):
        return f"{self.fullname} joined on {self.date_joined:%m/%d/%y}"

# The class ends at the first un-indented line.

# Create an instance of the Member class named new_guy.
new_guy = Member('Rambo','Rocco Moe')

# Try out the date_joined method.
print(new_guy.show_datejoined())
```

```
Rocco Moe joined on 12/06/18
```

FIGURE 6-8: Changing the value of an object's attributes.

Passing parameters to methods

You can pass data into methods in the same way you do functions, by using parameter names inside the parentheses. However, keep in mind that `self` always appears there first, and it never receives data from the outside. For example, let's say you want to create a method called `.activate()` and set it to `True` if the user is allowed to log in, `False` when the user isn't. Whatever you pass in is assigned to the `.is_active` attribute. Here is how to define that method in your code:

```
# Method to activate (True) or deactivate (False) account.
def activate(self, yesno):
    """ True for active, False to make inactive """
    self.is_active = yesno
```

The docstring we put in there is optional. But it does appear on the screen when you're typing relevant code in VS Code, so it serves as a good reminder about what you can pass in. When executed, this method doesn't show anything on the screen, it just changes the `is_active` attribute for that member to whatever you passed in as the `yesno` parameter.

REMEMBER

It helps to understand that a *method* is really just a *function*. What makes a method different from a generic function is the fact that a method is always associated with some class. So it's not as generic as a function.

Figure 6-9 shows the whole class followed by some code to test it. The line `new_guy = Member('romo', 'Rocco Moe')` creates a new member object named `new_guy`. Then, `print(new_guy.is_active)` displays the value of the `is_active` attribute, which is `True` because that's the default for all new members.

```
import datetime as dt

# Define a new class name Member.
class Member:
    """ Create a new member. """
    def __init__(self, username, fullname):
        # Define attributes and give them values.
        self.username = username
        self.fullname = fullname
        # Default date_joined to today's date.
        self.date_joined =  dt.date.today()
        # Set is active to True initially.
        self.is_active = True

    # Methods for each instance created (instance methods)

    # A method to return a formatted string with showing date joined.
    def show_datejoined(self):
        return f"{self.fullname} joined on {self.date_joined:%m/%d/%y}"

    # Method to activate (True) or deactive (False) account.
    def activate(self, yesno):
        """ True for active, False to make inactive """
        self.is_active = yesno

# The class ends at the first un-indented line.

# Create an instance of the Member class named new_guy.
new_guy = Member('Rambo','Rocco Moe')

# Is new-guy active?
print(new_guy.is_active)

# Try out the activate method.
new_guy.activate(False)

# Is new-guy still active?
print(new_guy.is_active)
```

```
True
False
```

FIGURE 6-9 Adding and testing an `.activate()` method.

The line `new_guy.activate(False)` calls the `activate()` method for that object and passes to it a Boolean `False`. Then `print(new_guy.is_active)` proves that the call to activate did indeed change the `is_active` attribute for `new_guy` from `True` to `False`.

Calling a class method by class name

As you've seen, you can call a class's method using the syntax

```
specificobject.method()
```

An alternative is to use the specific class name, which can help make the code a little easier to understand for a human. There's no right or wrong way, no best

practice or worst practice, to do this. There's just two different ways to achieve a goal, and you can use whichever you prefer. Anyway, that alternative syntax is:

```
Classname.method(specificobject)
```

Replace *Classname* with the name of the class (which we typically define starting with a capital letter), followed by the method name, and then put the specific object (which you've presumably already created) inside the parentheses.

For example, suppose we create a new member named Wilbur using the `Member` class and this code:

```
wilbur = Member('wblomgren', 'Wilbur Blomgren')
```

Here, `wilbur` is the specific object we created from the `Member` class. We can call the `show_datejoined()` method on that object using the syntax you've already seen, like this:

```
print(wilbur.show_datejoined())
```

The alternative is to call the `show_datejoined()` method of the `Member` class and pass to it that specific object, `wilbur`, like this:

```
print(Member.show_datejoined(wilbur))
```

The output from both methods is exactly the same, as in the following (but with the date on which you ran the code):

```
Wilbur Blomgren joined on 12/06/18
```

Again, the latter method isn't faster, slower, better, worse, or anything like that. It's just an alternative syntax you can use, and some people prefer it because starting the line with `Member` makes it clear to which class the `show_datejoined()` method belongs. This in turn can make the code more readable by other programmers, or by yourself a year from now when you don't remember any of the things you originally wrote in this app.

Using class variables

So far you've seen examples of attributes, which are sometimes called *instance variables*, because they're placeholders that contain information that varies from one instance of the class to another. For example, in a dog class, `dog.breed` may be `Poodle` for one dog, `Schnauzer` for another dog.

There is another type of variable you can use with classes, called a *class variable*, which is applied to all new instances of the class that haven't been created yet. Class variables inside a class don't have any tie-in to `self`, because the keyword `self` always refers to the specific object being created at the moment. To define a class variable, place the mouse pointer above the `def __init__` line and define the variable using the standard syntax:

```
variablename = value
```

Replace *variablename* with a name of your own choosing, and replace *value* with the specific value to want to assign that variable. For example, let's say there is code in your member's application that grants people three months (90 days) of free access on sign-up. You're not sure if you want to commit to this forever, so rather than hardcode it into your app (so it's difficult to change), you can just make it a class variable that's automatically applied to all new objects, like this:

```
# Define a class named Member for making member objects.
class Member:
    """ Create a member object """
    free_days = 90

    def __init__(self, username, fullname):
```

That `free_days` variable is defined before the `__init__` is defined, so it's not tied to any specific object in the code. Then, let's say, later in the code you want to store the date that the free trial expires. You could have attributes named `date_joined` and `free_expires` which represent today's date plus the number of days defined by `free_days`. Intuitively it may seem as though you could add `free_days` to the date using a simple syntax like this:

```
self.free_expires = dt.date.today() + dt.timedelta(days = free_days)
```

This wouldn't work. If you tried to run the code that way, you'd get an error saying Python doesn't recognize the `free_days` variable name (even though it's defined right at the top of the class). For this to work, you have to precede the variable name with the class name or `self`. For example, this would work:

```
self.free_expires = dt.date.today() + dt.timedelta(days = Member.free_days)
```

Figure 6-10 shows the bigger picture. We removed some of the code from the original class to trim it down and make it easier to focus on the new stuff. The `free_days = 365` line near the top sets the value of the `free_days` variable to 365. Then, later in the code a method used `Member.free_freedays` to add that number of days to the current date. Running this code by creating a new member named

wilbur and viewing his date_joined and free_expires attributes shows the current date (whatever that is where you're sitting when you run the code), and the date 365 days after that.

```
import datetime as dt
# Define a new class name Member.
class Member:
    # Default number of free days.
    free_days = 365

    """ Create a new member. """
    def __init__(self, username, fullname):
        self.date_joined =  dt.date.today()
        # Set an expiration date
        self.free_expires = dt.date.today() + dt.timedelta(days = Member.free_days)

# The class ends at the first un-indented line.

# Create an instance of the Member class named new_guy.
wilbur = Member('wblomgren', 'Wilbur Blomegren')

print(wilbur.date_joined)
print(wilbur.free_expires)
```

```
2018-12-06
2019-12-06
```

FIGURE 6-10: The variable free_days is a class variable in this class.

So, what if later down the road you decide giving people 90 free days is plenty. You could just change that in the class directly, but since it's a variable, you can do it on-the-fly, like this, outside the class:

```
#Set a default for free days.
Member.free_days = 90
```

When you run this code, you still create a user named wilbur with date_joined and free_days variables. But this time, wilbur.free_expires will be 90 days after the datejoined, not 365 days

Using class methods

Recall that a method is a function that's tied to a particular class. So far, the methods you've used, like .show_datejoined() and .activate() have been *instance methods*, because you always use them with a specific object . . . a specific instance of the class. With Python, you can also create *class methods*.

As the name implies, a class method is a method that is associated with the class as a whole, not specific instances of the class. In other words, class methods are similar in scope to class variables in that they apply to the whole class and not just individual instances of the class.

As with class variables, you don't need the `self` keyword with class methods, because that keyword always refers to the specific object being created at the moment, not to all objects created by the class. So for starters, if you want a method to do something to the class as a whole, don't use `def` *`name`*`(self)` because the `self` immediately ties the method to one object.

It would be nice if all you had to do to create class methods is exclude the word `self`, but unfortunately it doesn't work that way. To define a class method, you first need to type this into your code:

```
@classmethod
```

The `@` at the start of this defines it as a *decorator* — yep, yet another buzzword to add to your ever-growing list of nerd-o-rama buzzwords. A decorator is generally something that alters or extends the functionality of that to which it is applied.

Underneath that line, define your class method using this syntax:

```
    def methodname(cls,x):
```

Replace *`methodname`* with whatever name you want to give you method. Leave the `cls` as-is because that's a reference to the class as a whole (because the `@classmethod` decorator defined it as such behind-the-scenes). After the `cls`, you can have commas and the names of parameters that you want to pass to the method, just as you can with regular instance methods.

For example, suppose you want to define a method that sets the number of free days just before you start creating objects, so all objects get that same `free_days` amount. The code below accomplishes that by first defining a class variable named `free_days` that has a given default value of zero (the default value can be anything).

Further down in the class is this class method:

```
# Class methods follow @classmethods and use cls rather then self.
@classmethod
def setfreedays(cls,days):
    cls.free_days = days
```

This code tells Python that when someone calls the `setfreedays()` method on this class, it should set the value of `cls.free_days` (the `free_days` class variable for this class) to whatever number of days they passed in. Figure 6-11 shows a complete example in a Jupyter cell (which you can certainly type in and try for yourself), and the results of running that code.

```
import datetime as dt
# Define a new class name Member.
class Member:
    # Default number of free days.
    free_days = 365

    """ Create a new member. """
    def __init__(self, username, fullname):
        self.date_joined =  dt.date.today()
        # Set an expiration date
        self.free_expires = dt.date.today() + dt.timedelta(days = Member.free_days)

    # Class methods follow @classmethod decorator and refer to cls rather then to self.
    @classmethod
    def setfreedays(cls,days):
        cls.free_days = days
```

FIGURE 6-11: The set freedays( ) method is a class method in this class.

So let's see what happens when you run this code. This line:

```
Member.setfreedays(30)
```

. . . tells Python to call the `setfreedays()` method of the `Python` class and to pass to it the number 30. So, inside the class, the `free_days = 0` variable receives a new value of `30`, overriding the original `0`.

REMEMBER

It's easy to forget that upper- and lowercase letters matter a lot in Python, especially since it seems you're using lowercase 99.9 percent of the time. But as a rule, class names start with an initial cap, so any call to the class name must also start with an initial cap.

Next, the code creates a member named `wilbur`, and then prints the values of his `date_joined` and `free_expires` attributes:

```
wilbur = Member('wblomgren', 'Wilbur Blomgren')
print(wilbur.date_joined )
print(wilbur.free_expires)
```

The exact output of this depends on the date of the day when you run this code. But the first date should be today's date, whereas the `free_expires` date should be 30 days later (or whatever number of days you specified in the `Member.setfreedays(30)` line).

Using static methods

Just when you thought you may finally be done learning about classes, it turns out there is another kind of method you can create in a Python class. It's called a *static method* and it starts with this decorator: `@staticmethod`.

So at least that part is easy. What makes a static method different from instance and class methods is that a static method doesn't relate specifically to an instance

of an object, or even to the class as a whole. It really is generic function, and really the only reason to define it as part of a class would be that you wanted to use that same name elsewhere for something else in your code. In other words, it's strictly an organizational thing to keep together code that goes together so it's easy to find when you're looking through your code to change or embellish things.

Anyway, underneath that `@staticmethod` line you define your static method the same as any other method, but you don't use `self` and you don't use `cls`. Because a static method isn't strictly tied to a class or object, except to the extent you want to keep it there for organizing your code. Here's an example:

```
    @staticmethod
    def currenttime():
        now = dt.datetime.now()
        return f"{now:%I:%M %p}"
```

So we have a method called `currenttime()` that isn't expecting any data to be passed in, and doesn't even care about that object you're working with, or even the class; it just gets the current datetime using `now = dt.datetime.now()` and then returns that information in a nice 12:00 PM type format.

Figure 6-12 shows a complete example in which you can see the static method properly indented and typed near the end of the class. When code outside the class calls `Member.currenttime()`, it dutifully returns whatever the time is at the moment, even without your saying anything about a specific object from that class.

```
import datetime as dt

# Define a class named Member for making member objects.
class Member:
    # This is a class variable that's the same for all instances.
    free_days = 0

    """ Create a member object from username and fullname """
    def __init__(self, username, fullname):
        # Define properties and assign default values.
        self.datejoined = dt.date.today()
        self.free_expires = dt.date.today() + dt.timedelta(Member.free_days)

    # Class methods follow @classmethods and use cls rather then self.
    @classmethod
    def setfreedays(cls,days):
        cls.free_days = days

    @staticmethod
    def currenttime():
        now = dt.datetime.now()
        return f"{now:%I:%M %p}"

# Class definition ends at last indented line

# Try out the new static method (no object required)
print(Member.currenttime())
```

```
03:24 PM
```

FIGURE 6-12: The `Member` class now has a static method named `currenttime()`.

Understanding Class Inheritance

People who are really into object-oriented programming live to talk about class inheritance and subclasses and so on, stuff that means little or nothing to the average Joe or Josephine on the street. Still, what they're talking about as a Python concept is actually something you see in real life all the time.

As mentioned earlier, if we consider dog DNA to be a kind of "factory" or Python class, we can lump all dogs together as members of class of animals we call dogs. Even though each dog is unique, all dogs are still dogs because they are members of the class we call dogs, and we can illustrate that, as in Figure 6-13.

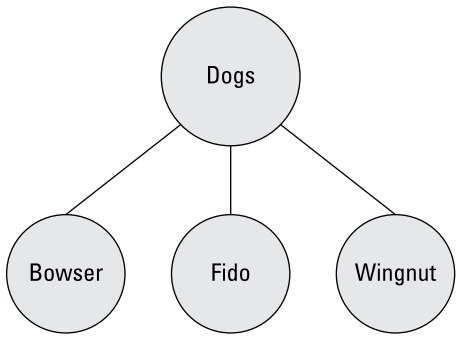


FIGURE 6-13: Dogs as "objects" of the class dogs.

So each dog is unique (although no other dog is as good as yours), but what makes dogs similar to each another are the characteristics that they *inherit* from the class of dogs.

The notions of class and class inheritance that Python and other object-oriented languages offer didn't materialize out of the clear blue sky just to make it harder and more annoying to learn this stuff. Much of the world's information can best be stored, categorized, and understood by using classes and subclasses and sub-subclasses, on down to individuals.

For example, you may have noticed that there are other dog-like creatures roaming the planet (although they're probably not the kind you'd like to keep around the house as pets). Wolves, coyotes, and jackals come to mind. They are similar to dogs in that they all *inherit* their dogginess from a higher level class we could call canines, as shown in Figure 6-14.

Using our dog analogy, we certainly don't need to stop at canines on the way up. We can put mammals above that, because all canines are mammals. We can put animals above that, because all mammals are animals. And we can put living things above that, because all animals are living things. So basically all the things

that make a dog a dog stem from the fact that each one *inherits* certain characteristics from numerous "classes" or critters that preceded it.

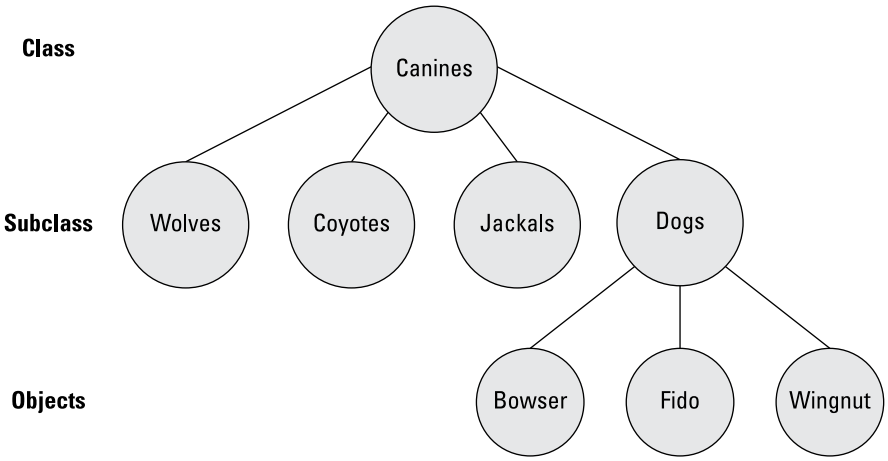


FIGURE 6-14: Several different kinds of animals are similar to dogs.


TECHNICAL STUFF

To the biology brainiacs out there, yes I know that Mammalia is a class, Canis is a genus, and below that are species. So you don't need to email or message me on that. I'm using *class* and *subclass* terms here just to relate the *concept* to classes, subclasses, and objects in Python.

Obviously the concept doesn't just apply to dogs. There are lots of different cats in the world too. There's cute little Bootsy, with whom you'd be happy to share your bed, and plenty of other felines, such as lions, tigers, and jaguars, with whom you probably wouldn't.


TIP

If you google *living things hierarchy* and click Images, you'll see just how many ways there are to classify all living things, and how inheritance works its way down from the general to the specific living thing.

Even our car analogy can follow along with this. At the top, we have transportation vehicles. Under that, perhaps boats, planes, and automobiles. Under automobiles we have cars, trucks, vans, and so forth and so on, down to any one specific car. So classes and subclasses are nothing new. What's new is simply thinking about representing those things to mindless machines that we call computers. So let's see how you would do that.

From a coding perspective, the easiest way to do inheritance is by creating subclasses within a class. The class defines things that apply to all instances of that class. Each subclass defines things that are relevant only to the subclass without replacing anything that's coming from the generic "parent" class.

Creating the base (main) class

Subclasses inherit all the attributes and methods of some higher-level main class, or parent class, which is usually referred to as the *base class*. This class is just any class, no different from what you've seen in this chapter so far. We'll use a `Members` class again, but we'll whittle it down to some bare essentials that have nothing to do with subclasses, so you don't have all the extra irrelevant code to dig through. Here is the basic class:

```
import datetime as dt
# Class is used for all kinds of people.
import datetime as dt

# Base class is used for all kinds of Members.
class Member:
    """ The Member class attributes and methods are for everyone """
    # By default, a new account expires in one year (365 days)
    expiry_days = 365

    # Initialize a member object.
    def __init__(self, firstname, lastname):
        # Attributes (instance variables) for everybody.
        self.firstname = firstname
        self.lastname = lastname
        # Calculate expiry date from today's date.
        self.expiry_date = dt.date.today() + dt.timedelta(days=self.expiry_days)
```

By default, new accounts expire in one year. So this class first sets a class variable name `expiry_days` to 365 to be used in later code to calculate the expiration date from today's date. As you'll see later, we used a class variable to define that, because we can give it a new value from a subclass.

To keep the code example simple and uncluttered, this version of the `Member` class accepts only two parameters, `firstname` and `lastname`.

Figure 6-15 shows an example of testing the code with a hypothetical member named Joe. Printing Joe's `firstname`, `lastname`, and `expiry_date` shows what you would expect the class to do when passing the `firstname` Joe and the `lastname` Anybody. When you run the code, the `expiry_date` should be one year from whatever date it us when you run the code.

Now suppose our real intent is to make two different kinds of users, `Admins` and `Users`. Both types of users will have the attributes that the `Member` class offers. So by defining those types of users as subclasses of `Member`, they will automatically get the same attributes (and methods, if any).

```
import datetime as dt
# Class is used for all kinds of people.
import datetime as dt

# Base class is used for all kinds of Members.
class Member:
    """ The Member class attributes and methods are for everyone """
    # By default, a new account expires in one year (365 days)
    expiry_days = 365

    # Initialize a member object.
    def __init__(self, firstname, lastname):
        # Attributes (instance variables) for everybody.
        self.firstname = firstname
        self.lastname = lastname
        # Calculate expiry date from today's date.
        self.expiry_date = dt.date.today() + dt.timedelta(days=self.expiry_days)

# Outside the class now.
Joe = Member('Joe', 'Anybody')
print(Joe.firstname)
print(Joe.lastname)
print(Joe.expiry_date)
```

```
Joe
Anybody
2019-12-08
```

FIGURE 6-15: A simplified `Member` class.

Defining a subclass

To define a subclass, make sure you get the cursor below the base class, and back to no indentation, because the subclass isn't a part of, or contained within, the base class. To define a subclass, use this syntax:

```
class subclassname(mainclassname):
```

Replace *`subclassname`* with whatever you want to name this subclass. Replace *`mainclassname`* with the name of the base class, as defined at the top of the base class. For example, to make a subclass of `Member` named `Admin`, use:

```
class Admin(Person):
```

To create another subclass named `User`, add this code:

```
class User(Person):
```

If you leave the classes empty, you won't be able to test because you'll get an error message telling you the class is empty. But you can put the word `pass` as the first command in each one. This is your way of telling Python "Yes I know these classes are empty, but let it pass, don't throw an error message"). You can put a comment above each one to remind you of what each one is for, as in the following:

```
# Subclass for Admins.
class Admin(Member):
    pass
```

```
# Subclass for Users.
class User(Member):
    pass
```

When you use the subclasses, you don't have to make any direct reference to the `Member` class. The `Admin`s and `User`s will both inherit all the `Member` stuff automatically. So, for example, to create an `Admin` named `Annie`, you'd use this syntax:

```
Ann = Admin('Annie', 'Angst')
```

To create a `User`, do the same thing with the `User` class and a name for the user. For example:

```
Uli = User('Uli', 'Ungula')
```

To see if this code works, you can do the same thing you did for Member Joe. After you create the two accounts, use `print()` statements to see what's in them. Figure 6-16 shows the results of creating the two users. Ann is an Admin, and Uli us a User, but both of them automatically get all the attributes (attributes) assigned to members. (The `Member` class is directly above the code shown in the image. I left that out because it hasn't changed).

```
# Subclass for Admins.
class Admin(Member):
    pass

# Subclass for Users.
class User(Member):
    pass

Ann = Admin('Annie', 'Angst')
print(Ann.firstname)
print(Ann.lastname)
print(Ann.expiry_date)
print()
Uli = User('Uli', 'Ungula')
print(Uli.firstname)
print(Uli.lastname)
print(Uli.expiry_date)
```

```
Annie
Angst
2019-12-03

Uli
Ungula
2019-12-03
```

FIGURE 6-16: Creating and testing a `Person` subclass.

So what you've learned here is that the subclass accepts all the different parameters that the base class accepts and assigns them to attributes, same as the `Person` class. But so far `Admin` and `User` are just members with no unique characteristics. In real life, there will probably be some differences between these two types of users. In the next sections you learn different ways to make these differences happen.

Overriding a default value from a subclass

One of the simplest things you can do with a subclass is give an attribute that has a default value in the base class some other value. For example, in the `Member` class we created a variable named `expiry_days` to be used later in the class to calculate an expiration date. But suppose you want `Admin` accounts to never expire (or to expire after some ridiculous duration so there's still some date there). All you have to do is set the new `expiry_date` in the `Admin` class (and you can remove the `pass` line since the class won't be empty anymore). Here's how this may look in your `Admin` subclass:

```
# Subclass for Admins.
class Admin(Member):
    # Admin accounts don't expire for 100 years.
    expiry_days = 365.2422 * 100
```

Whatever value you pass will override the default set near the top of the `Member` class, and will be used to calculate the `Admin`'s expiration date.

Adding extra parameters from a subclass

Sometimes, members of a subclass have some parameter value that other members don't. In that case, you may want to pass a parameter from the subclass that doesn't even exist in the base class. Doing so is a little more complicated than just changing a default value, but it's a fairly common technique so you should be aware of it. Let's work through an example.

For starters, your subclass will need its own `def __init__` line that contains everything that's in the base class's `__init__`, plus any extra stuff you want to pass. For example, let's say admins have some secret code and you want to pass that from the `Admin` subclass. You still have to pass the first and last name, so your `def __init__` line in the `Admin` subclass will look like this:

```
def __init__(self, firstname, lastname, secret_code):
```

The indentation level will be the same as the lines above it.

Next, any parameters that belong to the base class, `Member`, need to be passed up there using this rather odd-looking syntax:

```
        super().__init__(param1, param2,...)
```

Replace *`param1`*, *`param2`*, and so forth with the names of parameters you want to send to the base class. This should be everything that's already in the `Member`

parameters excluding `self`. In this example, `Member` expects only `firstname` and `lastname`. So the code for this example will be:

```
        super().__init__(firstname, lastname)
```

Whatever is left over you can assign to the subclass object using the standard syntax:

```
self.parametername = parametername
```

Replace *parametername* with the name of the parameter that you didn't send up to `Member`. In this case, that would be the `secret_code` parameter. So the code would be:

```
      self.secret_code = secret_code
```

Figure 6-17 shows an example in which we created an `Admin` user named `Ann` and passed `PRESTO` as her secret code. Printing all her attributes shows that she does indeed have the right expiration date still, and a secret code. As you can see, we also created a regular `User` named `Uli`. `Uli`'s data isn't impacted at all by the changes to `Admin`.

```
# Subclass for Admins.
class Admin(Member):
    # Admin accounts don't expire for 100 years.
    expiry_days = 365.2422 * 100
    # Subclass parameters
    def __init__(self, firstname, lastname, secret_code):
        # Pass Member parameters on up to Member class.
        super().__init__(firstname, lastname)
        # Assign the remaining parameter to this object.
        self.secret_code = secret_code

# Subclass for Users.
class User(Member):
    pass

Ann = Admin('Annie', 'Angst', 'PRESTO')
print(Ann.firstname, Ann.lastname, Ann.expiry_date, Ann.secret_code)

print()
Uli = User('Uli', 'Ungula')
print(Uli.firstname, Uli.lastname, Uli.expiry_date)
```

```
Annie Angst 2118-12-03 PRESTO

Uli Ungula 2019-12-03
```

FIGURE 6-17: The `Admin` subclass has a new `secret_code` parameter.

There is one little loose end remaining, which has to do with the fact that a `User` doesn't have a secret code. So if you tried to print the `.secret_code` for a person who has a `User` account rather than an `Admin` account, you'd get an error message. One way to deal with this is to just remember that Users don't have secret codes and never try to access one.

As an alternative, you can give `Users` a secret code that's just an empty string. So when you try to print or display it, you get nothing, but you don't get an error message either. To use this method, just add this to the main `Member` class:

```
# Default secret code is nothing
self.secret_code = ""
```

So even though you don't do anything with `secret_code` in the `User` subclass, you don't have to worry about throwing an error when you try to access the secret code for a `User`. The `User` will have a secret code, but it will just be an empty string. Figure 6-18 shows all the code with both subclasses, and also an attempt to print `Uli.secret_code`, which just displays nothing without throwing an error message.

```
import datetime as dt

# Base class is used for all kinds of Members.
class Member:
    """ The Member class properties and methods are for everyone """
    # By default, a new account expires in one year (365 days)
    expiry_days = 365

    # Initialize a member object.
    def __init__(self, firstname, lastname):
        # Properties (instance variables) for everybody.
        self.firstname = firstname
        self.lastname = lastname
        # Calculate expiry date from today's date.
        self.expiry_date = dt.date.today() + dt.timedelta(days=self.expiry_days)
        # Default secret code is nothing
        self.secret_code = ''

    # Method in the base class
    def showexpiry(self):
        return  f"{self.firstname} {self.lastname} expires on {self.expiry_date}"

# Subclass for Admins.
class Admin(Member):
    # Admin accounts don't expire for 100 years.
    expiry_days = 365.2422 * 100

    # Subclass parameters
    def __init__(self, firstname, lastname, secret_code):
        # Pass Member parameters on up to Member class.
        super().__init__(firstname, lastname)
        # Assign the remaining parameter to this object.
        self.secret_code = secret_code

# Subclass for Users.
class User(Member):
    pass

Ann = Admin('Annie', 'Angst', 'PRESTO')
print(Ann.firstname, Ann.lastname, Ann.expiry_date, Ann.secret_code)
print() # Add a blank line to output.

Uli = User('Uli', 'Ungula')
print(Uli.firstname, Uli.lastname, Uli.expiry_date, Uli.secret_code)
```

```
Annie Angst 2118-12-08 PRESTO

Uli Ungula 2019-12-08
```

FIGURE 6-18: The complete `Admin` and `User` subclasses.

We left the `User` subclass with `pass` as its only statement. In real life, you would probably come up with more default values or parameters for your other subclasses. But the syntax and code is exactly the same for all subclasses, so we won't dwell on that one. The skills you've learned in this section will work for all your classes and subclasses.

Calling a base class method

Methods in the base class work the same for subclasses as they do for the base class. To try it out, add a new method called `showexpire(self)` to the bottom of the base class, as follows:

```
class Member:
    """ The Member class attributes and methods are for everyone """
    # By default, a new account expires in one year (365 days)
    expiry_days = 365

    # Initialize a member object.
    def __init__(self, firstname, lastname):
        # Attributes (instance variables) for everybody.
        self.firstname = firstname
        self.lastname = lastname
        # Calculate expiry date from today's date.
        self.expiry_date = dt.date.today() + dt.timedelta(days=self.expiry_days)
        # Default secret code is nothing
        self.secret_code = ''

    # Method in the base class
    def showexpiry(self):
        return  f"{self.firstname} {self.lastname} expires on {self.
   expiry_date}"
```

The `showexpiry()` method, when called, returns a formatted string containing the user's first and last name and expiration date. Leaving the subclasses untouched and executing the code displays the names and expiry dates of Ann and Uli:

```
Ann = Admin('Annie', 'Angst', 'PRESTO')
print(Ann.showexpiry())


Uli = User('Uli', 'Ungula')
print(Uli.showexpiry())
```

Here is that output, although your dates will differ based on the date that you ran the code:

```
Annie Angst expires on 2118-12-04
Uli Ungula expires on 2019-12-04
```

Using the same name twice

The one loose end you may be wondering about is what happens when you use the same name more than once? Python will always opt for the most specific one, the one that's tied to the subclass. It will only use the more generic method from the base class if there is nothing in the subclass that has that method name.

To illustrate, here's some code that whittles the `Member` class down to just a couple of attributes and methods, to get any irrelevant code out of the way. Comments in the code describe what's going on in the code:

```
class Member:
    """ The Member class attributes and methods """
    # Initialize a member object.
    def __init__(self, firstname, lastname):
        # Attributes (instance variables) for everybody.
        self.firstname = firstname
        self.lastname = lastname

    # Method in the base class
    def get_status(self):
        return  f"{self.firstname} is a Member."

# Subclass for Administrators
class Admin(Member):
    def get_status(self):
        return  f"{self.firstname} is an Admin."

# Subclass for regular Users
class User(Member):
    def get_status(self):
        return  f"{self.firstname} is a regular User."
```

The `Member` class, and both the `Admin` and `User` classes have a method named `get_status()`, which shows the member's first name and status. Figure 6-19 shows the result of running that code with an `Admin`, a `User`, and a `Member` who is neither an `Admin` nor a `User`. As you can see, the `get_status` called in each case is the `get_status()` that's associated with the user's subclass (or base class in the case of the person who is a `Member`, neither an `Admin` or `User`).

Python has a built-in `help()` method that you can use with any class to get more information about that class. For example, at the bottom of the code in Figure 6-19, add this line:

```
help(Admin)
```

When you run the code again, you'll see some information about that Admin class, as you can see in Figure 6-20.

```
class Member:
    """ The Member class attributes and methods are for everyone """
    # Initialize a member object.
    def __init__(self, firstname, lastname):
        # Attributes (instance variables) for everybody.
        self.firstname = firstname
        self.lastname = lastname

    # Method in the main class
    def get_status(self):
        return  f"{self.firstname} is a Member."

# Subclass for Administrators
class Admin(Member):
    def get_status(self):
        return  f"{self.firstname} is an Admin."

# Subclass for regular Users
class User(Member):
    def get_status(self):
        return  f"{self.firstname} is a regular User."

# Create an admin
Ann = Admin('Annie', 'Angst')
print(Ann.get_status())

#Create a user
Uli = User('Uli', 'Ungula')
print(Uli.get_status())

# Create a member (neither Admin or User)
Manny = Member("Mindy", "Membo")
print(Manny.get_status())
```

```
Annie is an Admin.
Uli is a regular User.
Mindy is a Member.
```

FIGURE 6-19: Three methods with the same name, `get_status()`.

```
help(Admin)
```

```
Help on class Admin in module __main__:

class Admin(Member)
 |  Admin(firstname, lastname)
 |
 |  The Member class attributes and methods are for everyone
 |
 |  Method resolution order:
 |      Admin
 |      Member
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  get_status(self)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from Member:
 |
 |  __init__(self, firstname, lastname)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Member:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

FIGURE 6-20: Output from `help(Admin)`.

You don't really need to worry about all the details of that figure right now, so don't worry if it's a little intimidating. For now, the most important thing is the section titled Method Resolution Order, which looks like this:

```
Method resolution order:
     Admin
     Member
     builtins.object
```

What the method resolution order tells you is that if a class (and its subclasses) all have methods with the same name (like `get_status`), then a call to `get_status()` from an `Admin` user will cause Python to look in `Admin` for that method and to use that one, if it exists. If no `get_status()` method was defined in the `Admin` subclass, then it looks in the `Member` class and uses that one, if found. If neither of those had a `get_status` method, it looks in `builtins.object`, which is a reference to certain `built_in` methods that all classes and subclasses share.

So the bottom line is, if you do store your data in hierarchies of classes and subclasses, and you call a method on a subclass, it will use that subclass method if it exists. If not, it will use the base class method, if it exists. If that also doesn't exist, it will try the built-in methods. And if all else fails, it will throw an error because it can't find the method your code is trying to call. Usually the main reason for this type of error is that you simply misspelled the method name in your code, so Python can't find it.

An example of a built-in method is `__dict__`. The `dict` is short for *dictionary*, and those are double-underscores surrounding the abbreviation. Referring back to Figure 6-20, executing the command

```
print(Admin.__dict__)
```

. . . doesn't cause an error, even though we've never defined a method named `__dict__`. That's because there is a built-in method with that name, and when called with `print()`, it shows a dictionary of methods (both yours and built-in ones) for that object. It's not really something you have to get too involved with this early in the learning curve. Just be aware that if you try to call a method that doesn't exist at any of those three levels, such as this:

```
print(Admin.snookums())
```

. . . you get an error that looks something like this:

```
--->  print(Admin.snookums())

AttributeError: type object 'Admin' has no attribute 'snookums'
```

This is telling you that Python has no idea what `snookums()` is about, so it can only throw an error. In real life, this kind of error is usually caused simply by misspelling the method name in your code.

Classes (and to some extent, subclasses) are pretty heavily used in the Python world, and what you've learned here should make it relatively easy to write your own classes, as well as to understand classes written by others. There is on more "core" Python concept you'll want to learn about before we finish this book, and that's how Python handles errors, and things you can do in your own code to better handle errors.

IN THIS CHAPTER

- Understanding exceptions
- Handling errors gracefully
- Keeping your app from crashing
- Using `try...except...else...finally`
- Raising your own exceptions

Chapter 7
Sidestepping Errors

We all want our programs to run perfectly all the time. But sometimes there are situations out there in the real world that won't let that happen. This is no fault of yours or your program's. It's usually something the person using the program did wrong. Error handling is all about trying to anticipate what those problems may be, and then "catching" the error and informing the user of the problem so they can fix it.

It's important to keep in mind the techniques here aren't for fixing bugs in your code. Those kinds of errors you have to fix yourself. We're talking strictly about errors in the environment in which the program is running, over which you have no control. *Handling* the error is simply a way of replacing the tech-speak error message that Python normally displays, which is meaningless to most people, with a message that tells them in plain English what's wrong and, ideally, how to fix it.

Again, the user will be fixing *the environment in which the program* is running . . . they won't be fixing your code.

Understanding Exceptions

In Python (and all other programing languages) the term *exception* refers to an error that isn't due to a programming error. Rather it's an error out in the real world that prevents the program from running properly. As simple

example, let's have your Python app open a file. The syntax for that is easy. The code is just

```
name = open(filename)
```

Replace *name* with a name of your own choosing, same as any variable name. Replace *filename* with the name of the file. If the file is in the same folder as the code, you don't need to specify a path to the folder because the current folder is assumed.

Figure 7-1 shows an example. We used VS Code for this example so that you can see the contents of the folder in which we worked. The folder contains a file named `showfilecontents.py`, which is the file that contains the Python code we wrote. The other file is named `people.csv`.

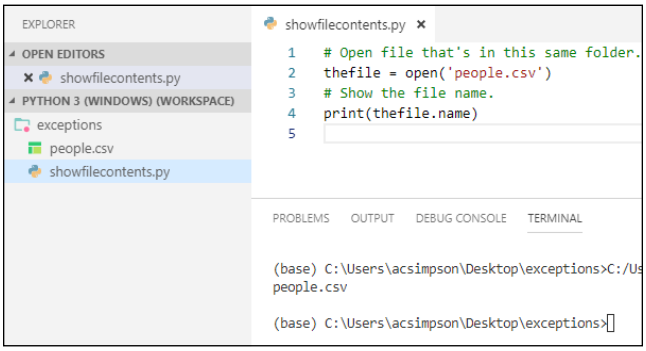
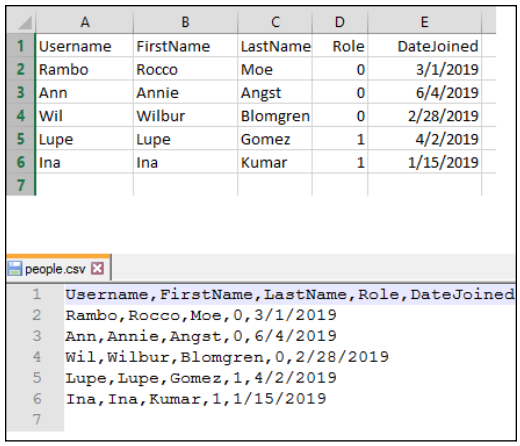


FIGURE 7-1: The `showfile contents.py` and `people.csv` files in a folder in VS Code.

The `showcontents.py` file is the only one that contains code. The `people.csv` file contains data, information about people. Its content doesn't really matter much right now; what you're learning here will work in any external file. But just so you know, Figure 7-2 shows the contents of that file in Excel (top) so it's easy for you to read, and in a text editor (bottom), which is how it actually looks to Python and other languages.

The Python code is just two lines (excluding the comments), as follows:

```
# Open file that's in this same folder.
thefile = open('people.csv')
# Show the file name.
print(thefile.name)
```



	A	B	C	D	E
1	Username	FirstName	LastName	Role	DateJoined
2	Rambo	Rocco	Moe	0	3/1/2019
3	Ann	Annie	Angst	0	6/4/2019
4	Wil	Wilbur	Blomgren	0	2/28/2019
5	Lupe	Lupe	Gomez	1	4/2/2019
6	Ina	Ina	Kumar	1	1/15/2019
7					

FIGURE 7-2: The contents of the `people.csv` file in Excel (top) and a text editor (bottom).

So it's simple. The first line of code opens the file named `people.csv`. The second line of code shows the filename (`people.csv`) on the screen. Running that simple `showfilecontents.py` app (by right-clicking its name in VS Code and choosing Run Python File in Terminal) shows `people.csv` on the screen — assuming there is a file named `people.csv` in the folder to open. This assumption is where exception handling comes in.

Suppose that for reasons beyond your control, the file named `people.csv` isn't there because some person or some automated procedure failed to put it there, or because someone accidentally misspelled the filename. It's easy to accidentally type, say, `.cvs` rather than `.csv` for the filename, as in Figure 7-3. If that's the case, running the app *raises an exception* (which in English means "displays an error message"), as you can see in the terminal in that same image. The exception reads

```
Traceback (most recent call last):
  File "c:/ Users/ acsimpson/ Desktop/ exceptions/ showfilecontents.py", line 2,
    in <module>
    thefile = open('people.csv')
FileNotFoundError: [Errno 2] No such file or directory: 'people.csv'
```

The `Traceback` is a reference to the fact that if there were multiple exceptions, they'd all be listed with the most recent being listed first. In this case, there is just one exception. The `File` part tells you where the exception occurred, in line `2` of the file named `showfilecontents.py`. The part that reads

```
thefile = open('people.csv')
```

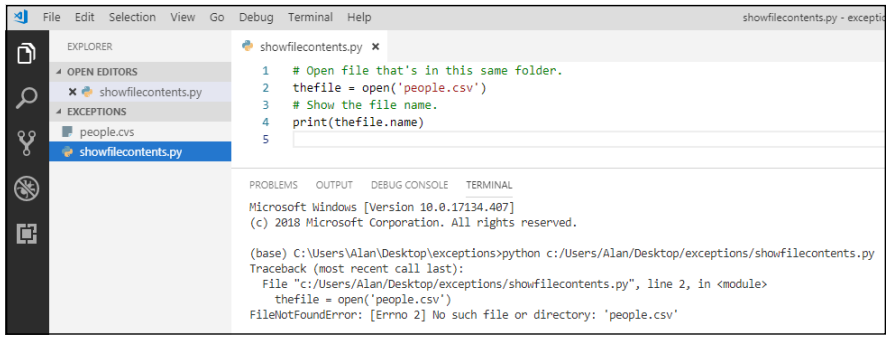


FIGURE 7-3: The `showfilecontents.py` raises an exception.

. . . shows you the exact line of code that caused the error. And finally the exception itself is described like this:

```
FileNotFoundError: [Errno 2] No such file or directory: 'people.csv'
```

The generic name for this type of error is `FileNotFoundError`. Many exceptions also have a number associated with them, but that tends to vary depending on the operating system environment, so it's not typically used for handling errors. In this case, the main error is `FileNotFoundError`, and the fact that's its `ERRNO 2` where I'm sitting right now doesn't really matter much.

TECHNICAL STUFF

Some people use the phrase *throw an exception* rather than *raise an exception*. But they're just two different ways to describe the same thing. There's no difference between *raising* and *throwing* an exception.

The last part tells you *exactly* what went wrong `No such file or directory: 'people.csv.'` In other words, Python can't do the `open('people.csv')` business, because there is no file named `people.csv` in the current folder with that name.

You could correct this problem by changing the code, but `.csv` is a common file extension for files that contain comma-separated values. It would make more sense to change the name of `people.cvs` to `people.csv` so it matches what the program is looking for and the `.csv` extension is well known.

You also can't have the Python app rename the file, because you don't know whether other files in that folder have the kind of data the Python app is looking for. It's up to a human to create the `people.csv` file, to make sure it's named correctly, and to make sure it contains the type of information the Python program it looking for.

Handling Errors Gracefully

The best way to handle this kind of error is to *not* show what Python normally shows for this error. Instead, it would be best to replace that with something the person that's using the app is more likely to understand. To do that, you can code a *try . . . except block* using this basic syntax:

```
try:
     The things you want the code to do
except Exception:
     What to do if it can't do what you want it to do
```

Here's how we can rewrite the `showfilecontents.py` code to handle the missing (or misspelled) file error:

```
try:
    # Open file and shows its name.
    thefile = open('people.csv')
    print(thefile.name)
except Exception:
    print("Sorry, I don't see a file named people.csv here")
```

Because we know that if the file the app is supposed to open may be missing, we start with `try:` and then attempt to open the file under that. If the file opens, great, the `print()` statement runs and shows the filename. But if trying to open the file raises an exception, the program doesn't "bomb" and display a generic error message. Instead it shows a message that's better for the average computer user, as shown in Figure 7-4.

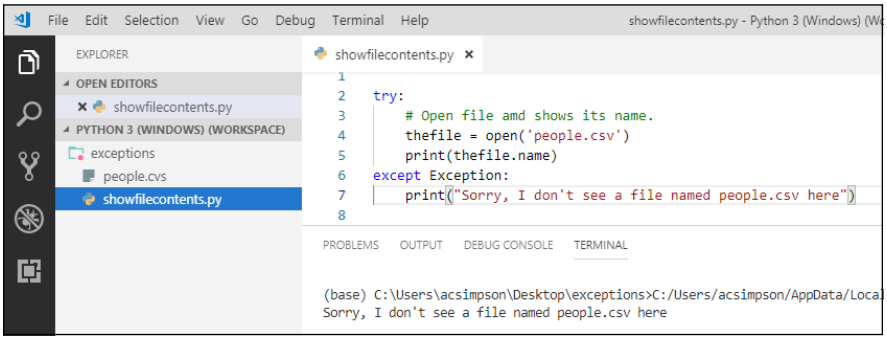


FIGURE 7-4: `showfile contents.py` catches the error and displays something nice.

Being Specific about Exceptions

The preceding syntax handled the "file not found" error gracefully. But it could be more graceful. For example, if you rename `people.cvs` to `people.csv` and run the app again, you see the filename on the screen. No error.

Now suppose you add another line of code under the `print` statement, something along these lines:

```
try:
    # Open file and shows its name.
    thefile = open('people.csv')
    print(thefile.name)
    print(thefile.wookems())
except Exception:
    print("Sorry, I don't see a file named people.csv here")
```

Running this code displays the following:

```
people.csv
Sorry, I don't see a file named people.csv here
```

It must have found the file in order to print the filename (which it does). But then it says it can't find the file. So what gives?

The problem is in the line `except Exception:` What that says is "if *any* exception is raised in this `try` block, do the code under the `except` line." Hmmm, this is not good because the error is actually caused by the line that reads

```
    print(thefile.wookems())
```

This line raises an exception because there is no property named `wookems()` in Python.

To clean this up, you want to replace `Exception:` with the specific exception you want it to catch. But how do you know what that specific exception is? Easy. The exception that gets raised with no exceptions handing is:

```
FileNotFoundError: [Errno 2] No such file or directory: 'people.csv'
```

That very first word is the name of the exception that you can use in place of the generic `Exception` name, like this:

```
try:
    # Open file and shows its name.
    thefile = open('people.csv')
    print(thefile.name)
    print(thefile.wookems())
except FileNotFoundError:
    print("Sorry, I don't see a file named people.csv here")
```

Granted, that doesn't do anything to help with the bad method name. But the bad method name isn't really an exception, it's a programming error that needs to be corrected in the code by replacing `.wookems()` with whatever method name you really want to use. But at least the error message you see isn't the misleading `Sorry, I don't see a file named people.csv here` error. This code just works, and so the regular error — `object has no attribute 'wookems'` — shows instead, as in Figure 7-5.

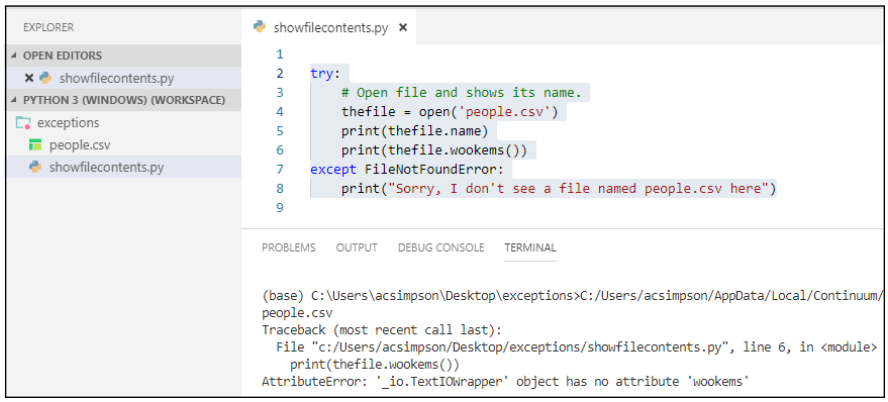


FIGURE 7-5: The correct error message shown.

Again, if you're thinking about handling the `.wookems` error, that's not an exception for which you'd write an exception handler. Exceptions occur when something *outside* the program upon which the program depends isn't available. Programming errors, like nonexistent method names, are errors inside the program and have to be corrected inside the program by the programmer who writes the code.

Keeping Your App from Crashing

You can stack up `except:` statements in a `try` block to handle different errors. Just be aware that when the exception occurs, it looks at each one starting at the top. If it finds a handler that matches the exception, it raises that one. If some exception occurred that you didn't handle, then you get the standard Python error message. But there's a way around that too.

If you want to avoid all Python error messages, you can just make the last one `except Exception:` so that it catches any exception that hasn't already be caught by any preceding `except:` in the code. For example, here we just have two handlers, one for `FileNotFound`, and one for everything else:


REMEMBER

I know you haven't learned about `open` and `readline` and `close`, but don't worry about that. All we care about here is the exception handling, which is the `try:` and `except:` portions of the code — for now.

```
try:
    # Open file and shows its name.
    thefile = open('people.csv')
    # Print a couple blank lines then the first line from the file.
    print('\n\n', thefile.readline())
    # Close the file.
    thefile.closed()

except FileNotFoundError:
    print("Sorry, I don't see a file named people.csv here")
except Exception:
    print("Sorry, something else went wrong")
```

Running this code produces the following output:

```
Username,FirstName,LastName,Role,DateJoined
Sorry, something else went wrong
```

The first line shows the first line of text from the `people.csv` file. The second line is the output from the second `except:` statement, which reads `Sorry, something else went wrong`. This message is vague and doesn't really help you find the problem.

Rather than just print some generic message for an unknown exception, you can capture the error message in a variable and then display the contents of that variable to see the message. As usual, you can name that variable anything you like, though a lot of people use *e* or *err* as an abbreviation for *error*.

For example, consider the following rewrite of the preceding code. The generic handler, `except Exception` now has an `as e` at the end, which means "whatever exception gets caught here, put the error message in a variable named `e`." Then the next line uses `print(e)` to display whatever is in that `e` variable:

```
try:
    # Open file and shows its name.
    thefile = open('people.csv')
```

```
    # Print a couple blank lines then the first line from the file.
    print('\n\n', thefile.readline())
    thefile.wigwam()

except FileNotFoundError:
   print("Sorry, I don't see a file named people.csv here")
   except Exception as e:
   print(e)
```

Running this code displays the following:

```
Username,FirstName,LastName,Role,DateJoined
'_io.TextIOWrapper' object has no attribute 'wigwam'
```

The first line with `Username, Firstname` and so forth is just the first line of text from the `people.csv` file. There's no error in the code, and that file is there, so that all went well. The second line is this:

```
'_io.TextIOWrapper' object has no attribute 'wigwam'
```

This is certainly not plain English. But it's better than "Something else went wrong." At least the part that reads `object has no attribute 'wigwam'` lets you know that the problem has something to do with the word *wigwam*. So you still handled the error gracefully, and the app didn't "crash." And you at least got some information about the error that should be helpful to you, even though it may not be so helpful to people who are using the app with no knowledge of its inner workings.

Adding an else to the Mix

If you look at code written by professional Python programmers, they usually don't have a whole lot of code under the `try:`. This is because you want your `catch:` blocks to catch certain possible errors and top processing with an error message should the error occur. Otherwise, you want it to just continue on with the rest of the code (which may also contain situations that generate other types of exceptions).

A more elegant way to deal with the problem uses this syntax:

```
try:
     The thing that might cause an exception
```

```
catch (a common exception):
     Explain the problem
catch Exception as e:
     Show the generic error message
else:
     Continue on here only if no exceptions raised
So the logic with this flow is
Try to open the file...
   If the file isn't there, tell them and stop.
   If the file isn't there, show error and stop
Otherwise...
   Go on with the rest of the code.
```

By limiting the `try:` to the one thing that's most likely to raise an exception, we can stop the code dead in its tracks before it tries to go any further. But if no exception is raised, then it continues on normally, below the `else`, where the previous exception handlers don't matter anymore. Here is all the code with comments explaining what's going on:

```
try:
    # Open the file name people.csv
    thefile = open('people.csv')
# Watch for common error and stop program if it happens.
except FileNotFoundError:
    print("Sorry, I don't see a file named people.csv here")
# Catch any unexpected error and stop program if one happens.
except Exception as err:
    print(err)
# Otherwise, if nothing bad has happened by now, just keep going.
else:
    # File must be open by now if we got here.
    print('\n') # Print a blank line.
    # Print each line from the file.
    for one_line in thefile:
        print(one_line)
    thefile.close()
    print("Success!")
```

WARNING

As always with Python, indentations matter a lot. Make sure you indent your own code as shown in this chapter or your code may not work right.

Figure 7-6 also shows all the code and the results of running that code in VS Code.

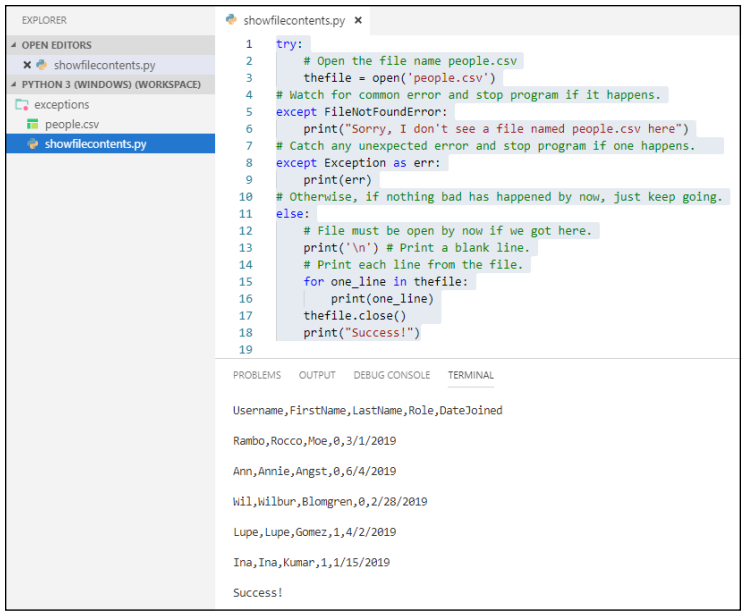


FIGURE 7-6: Code with try, exception handlers, and an else for when there are no exceptions.

Using try . . . except . . . else . . . finally

If you look at the complete syntax for Python exception handling, you'll see there is one more option at the end, like this:

```
try:
    try to do this
except:
    if x happens, stop here
except Exception as e:
    if something else bad happens, stop here
else:
    if no exceptions, continue on normally here
finally:
    do this code no matter what happened above
```

The `finally:` block, if included, is the code that runs whether an exception occurs or not. This pattern tends to be used in more complex apps, such as those in which a new chunk of code that's dependent on the availability of some external resource is called after some series of events is carried out. If the resource is available, the code plays out, but if the resource isn't available, some other code executes.

To illustrate, here is some code that expects to find an external resource named `people.csv` to be available to the code.

```
print('Do this first')
try:
    open('people.csv')
except FileNotFoundError:
    print('Cannot find file named people.csv')
except Exception as e:
    print('e')
else:
    print('Show this if no exception.')
finally:
    print('This is in the finally block')
print("This is outside the try...except...else...finally")
```

When you run this code with a file named `people.csv` in the folder, you get this output:

```
Do this first
Show this if no exception.
This is in the finally block
This is outside the try...except...else...finally
```

None of the exception-reporting code executed because the `open()` statement was able to open the file named people.csv.

Run this same code without a file named `people.csv` in the same folder, you get the following result. This time the code reports that it cannot fine a file named people.csv. But the app doesn't "crash". Rather, it just keeps on executing the rest of the code.

```
Do this first
Cannot find file named people.csv
This is in the finally block
This is outside the try...except...else...finally
```

What these examples show you is that you can control exactly what happens in some small part of a program that's vulnerable to user errors or other "outside" exceptions, while still allowing other code to run normally.

Raising Your Own Errors

Python has lots of built-in exceptions for recognizing and identifying errors, as you'll see while writing and testing code, especially when you're first learning. However, you aren't limited to those. If your app has some vulnerability that isn't covered by the built-in exceptions, you can invent your own.

For a detailed list of all the different exceptions that Python can catch, take a look at `https://docs.python.org/3/library/exceptions.html` in the Python.org documentation.

The general syntax for raising your own error is:

```
raise error
```

Replace *error* with the wording of the known error that you want to raise (such as `FileNotFoundError`). Or, if the error isn't covered by one of those built-in errors, you can just raise `Exception` and that will execute whatever is under `catch Exception:` in your code.

As a working example, let's say you want two conditions to be met for the program to run successfully:

- The `people.csv` file must exist so you can open it.
- The `people.csv` file must contain more than one row of data (the first row is just column names, not data, but if it has only column headings we want to consider it empty.

Here is an example of how you may handle that situation, just looking at the exception handling part:

```
try:
    # Open the file (no error check for this example).
    thefile = open('people.csv')
    # Count the number of lines in file.
    line_count = len(thefile.readlines())
    # If there is fewer than 2 lines, raise exception.
    if line_count < 2:
        raise Exception
# Handles missing file error.
except FileNotFoundError:
        print('\nThere is no people.csv file here')
```

```
# Handles all other exceptions
except Exception as e:
        # Show the error.
        print('\n\nFailed: The error was ' + str(e))
        # Close the file.
        thefile.close()
```

So let's step through it. The first lines try to open the `people.csv` file:

```
try:
    # Open the file (no error check for this example).
    thefile = open('people.csv')
```

We know that if the `people.csv` file doesn't exist, execution will jump to this exception handler, which tells the user the file isn't there:

```
except FileNotFoundError:
        print('\nThere is no people.csv file here')
```

Assuming that didn't happen and the file is now open, this next line counts how many lines are in the file:

```
    line_count = len(thefile.readlines())
```

If the file is empty, the line count will be 0. If the file contains only column headings, like this:

```
Username,FirstName,LastName,DateJoined
```

. . . then the length will be 1. We want the rest of the code to run only if the length of the file is 2 or more. So if the line count is less than 2 the code can raise an exception. You may not know what that exception is, so you tell it to raise a generic exception, like this:

```
    if line_count < 2:
        raise Exception
```

The exception handler in the code for general exceptions looks like this:

```
# Handles all other exceptions
except Exception as e:
        # Show the error.
        print('\n\nFailed: The error was ' + str(e))
```

```
        # Close the file.
        thefile.close()
```

The `e` grabs the actual exception, and then the next `print` statement shows what that was. So, let's say you run that code and `people.csv` is empty or incomplete. The output will be:

```
Failed: The error was
```

Notice there is no explanation of the error. That's because error that Python can recognize on its own was found. You could raise a known exception instead. For example, rather than raising a general `Exception`, you can raise a `FileNotFoundError`, like this:

```
if line_count < 2:
    raise FileNotFoundError
```

But if you do that, the `FileNotFoundError` handler is called and displays `There is no people.csv file`, which isn't really true in this case, and it's not the cause of the problem. There is a `people.csv` file; it just doesn't have any data to loop through. What you need is your own custom exception and handler for that exception.

All exceptions in Python are actually objects, instances of the built-in class named `Errors` in Python. To create your own exception, you first have to import the `Errors` class to use as a base class (much like the `Member` class was a base class for different types of users). Then, under that, you define your own error as a subclass of that. This code goes up at the top of the file so it's executed before any other code tries to use the custom exception:

```
# define Python user-defined exceptions
class Error(Exception):
   """Base class for other exceptions"""
   pass

# Your custom error (inherits from Error)
class EmptyFileError(Error):
    pass
```

As before, the word `pass` in each class just tells Python "I know this class has no code in it, and that's okay here. You don't need to raise an exception to tell me that."

Now that there exists an exception called `EmptyFileError`, you can raise *that* exception when the file has insufficient content. Then write a handler to handle that exception. Here's that code:

```
    # If there is fewer than 2 lines, raise exception.
    if line_count < 2:
        raise EmptyFileError
# Handles my custom error for too few rows.
except EmptyFileError:
        print("\nYour people.csv file doesn't have enough stuff.")
```

Figure 7-7 shows all the code.

```
# Base class for defining your own user-defined exceptions.
class Error(Exception):
   """Base class for other exceptions"""
   pass

# Now define your exception as a subclass of Error.
class EmptyFileError(Error):
    pass

try:
    # Open the file (no error check for this example).
    thefile = open('people.csv')
    # Count the number of lines in file.
    line_count = len(thefile.readlines())
    # If there is fewer than 2 lines, raise exception.
    if line_count < 2:
        raise EmptyFileError

# Handles missing file error.
except FileNotFoundError:
        print('\nThere is no people.csv file here')

# Handles my custom error for too few rows.
except EmptyFileError:
        print("\nYour people.csv file doesn't have enough stuff.")

# Handles all other exceptions
except Exception as e:
        # Show the error.
        print('\n\nFailed: The error was ' + str(e))
        # Close the file.
        thefile.close()
else:
    # This code runs only if no exceptions above.
    print() # Print a blank line.

    # File must be open by now if we got here, show content.
    for one_line in thefile:
        print(one_line)
    thefile.close()
    print("Success!")
```

FIGURE 7-7: Custom `EmptyFileError` added for exception handling.

So here is how things will play out when the code runs. If there is no `people.csv` file at all, this error shows:

```
There is no people.csv file here.
```

If there is a `people.csv` file but it's empty or contains only column headings, this is all the program shows:

```
Your people.csv file doesn't have enough stuff.
```

Assuming neither error happened, then the code under the `else:` runs and shows whatever is in the file on the screen.

So as you can see, exception handling lets you plan ahead for errors caused by vulnerabilities in your code. We're not taking about bugs in your code or coding errors here. We're generally talking about outside resources that the program needs to run correctly.

When those outside resources are missing or insufficient, you don't have to let the program just "crash" and display any nerd-o-rama error message on the screen to baffle your users. Instead, you can catch the exception and show them some text that tells them exactly what's wrong, which will, in turn, help them fix that problem and run the program again, successfully this time. That's what exception handling is all about.

3 Working with Python Libraries

Contents at a Glance

IN THIS CHAPTER

- Understanding text and binary files
- Opening and closing files
- Reading a file's contents
- Looping through a file
- Reading and copying binary files
- Conquering with CSV files
- From CSV to dictionaries and objects

Chapter 1

Working with External Files

Pretty much everything that's stored in your computer, be it a document, program, movie, photograph . . . whatever, is stored in a file. Most files are organized info *folders* (also called *directories*). On a Mac you can use Finder to browse around through folders and files. In Windows you use File Explorer or Windows Explorer (but not Internet Explorer) to browse around through folders and files.

Python offers many tools for creating, reading from, and writing to many different kinds of files. In this chapter, you learn all the most important skills for working with files using Python code.

Understanding Text and Binary Files

There are basically two types of files:

- **Text files:** Text files contain plain text characters. When you open these in a text editor, they show human-readable content. The text may not be in a

language you know or understand, but you will see mostly normal characters that you can type at any keyboard.

» **Binary files:** A binary file stores information in bytes that aren't quite so humanly readable. We don't recommend you do this, but if you open the binary file in a text editor, what you see may resemble Figure 1-1.

FIGURE 1-1: How a binary files looks in a program for editing text files.

WARNING

If you do ever open a binary file in a text editor and see this gobbledygook, don't panic. Just close the file or program and choose **No** if asked to save it. The file will be fine, so long as you don't save it.

Figure 1-2 lists examples of different kinds of text and binary files, some of which you may have worked with in the past. There are other files types, of course, but these are among the most widely used.

Text File

- **Plain Text**: .txt, .csv
- **Source Code**: .py, .html, .css. .js
- **Data**: .json, .xml

Binary File

- **Executable**: .exe, .dmg, .bin
- **Images**: .jpg, .png, .gif, .tiff, .ico
- **Video**: .mp4, .m4v, .mp4, .mov
- **Audio**: .aif, .mp3, .mpa, wav
- **Compressed**: .zip, .deb, .tar.gz
- **Font**: .woff, .otf, .ttf
- **Document**: .pdf, .docx, .xlsx

FIGURE 1-2: Common text and binary files.

As with any Python code, you can use a Jupyter notebook, VS Code, or virtually any coding editor to write your Python code. In this chapter, we use VS Code simply because its Explorer bar (on the left, when it's open) displays the contents of the folder in which you're currently working.

Opening and Closing Files

To open a file from within a Python app, use the syntax:

```
open(filename.ext[,mode])
```

Replace *filename.ext* with the filename of the file you want to open. If the file is not in the same directory as the Python code, you need to specify a path to the file. Use forward slashes, even if you're working in Windows. For example, if the file you want to open is `foo.txt` on your desktop, and your user account name is Alan, you'd use the path `C:/Users/Alan/Desktop/foo.txt` rather than the more common Windows syntax with backslashes like `C:\Users\Alan\Desktop\foo.txt`.

The `[,mode]` is optional (as indicated by the square brackets). Use it to specify what kind of access you want your app to have, using the following single-character abbreviations:

- **r: (Read):** Allows Python to open the file but not make any changes. This is the default if you don't specify a mode. If the file doesn't exist, Python raises a `FileNotFoundError` exception.
- **r+: (Read/Write):** Allows Python to read and write to the file.
- **a: (Append):** Opens the file and allows Python to add new content to the end of the file but not to change existing content. If the file doesn't exist, this mode creates the file.
- **w: (Write):** Opens the file and allows Python to make changes to the file. Creates the file if it doesn't exist.
- **x: (Create):** Creates the file if it doesn't already exist. If the file does exist, it raises a `FileExistsError` exception.

REMEMBER

For more information on exceptions, see Book 2, Chapter 7.

You can also specify what type of file you're opening (or creating). If you already specified one of the above modes, just add this as another letter. If you use just one of the letters below on its own, the file opens in Read mode.

- **t: (Text):** Open as a text file, read and write text.
- **b: (Binary):** Open as a binary file, read and write bytes.

There are basically two ways to use the `open` method. With one syntax you assign a variable name to the file, and use this variable name in later code to refer to the file. That syntax is:

```
var = open(filename.ext[,mode])
```

Replace *`var`* with a name of your choosing (though it's very common in Python to use just the letter `f` as the name). Although this method does work, it's not ideal because, after it's opened, the file remains open until you specifically close it using the `close()` method. Forgetting to close files can cause the problem of having too many files open at the same time, which can corrupt the contents of a file or cause your app to raise and exception and crash.

After the file is open, there are a few ways to access its content, as we discuss a little later in this chapter. For now, we simply copy everything that's in the file to a variable named *`filecontents`* in Python, and then we display this content using a simple `print()` function. So to open `quotes.txt`, read in all its content, and display that content on the screen, use this code:

```
f = open('quotes.txt')
filecontents = f.read()
print(filecontents)
```

With this method, the file remains open until you specifically close it using the file variable name and the `.close()` method, like this:

```
f.close()
```

It's important for your apps to close any files it no longer needs open. Failure to do so allows open file handlers to accumulate, which can eventually cause the app to throw an exception and crash, perhaps even corrupting some of the open files along the way.

Getting back to the act of actually opening the file, though: another way to do this is by using a *context manager* or by using *contextual coding.* This method starts with the word *`with`*. You still assign a variable name. But you do so near the end of the

line. The very last thing on the line is a colon which marks the beginning of the *with* block. All indented code below that is assumed to be relevant to the context of the open file (like code indented inside a loop). At the end of this you don't need to specifically close the file; Python does it automatically:

```
# ---------------- Contextual syntax
with open('quotes.txt') as f:
    filecontents = f.read()
    print(filecontents)

# The unindented line below is outside the with... block;
print('File is closed: ', f.closed)
```

The following code shows a single app that opens `quotes.txt`, reads and displays its content, and then closes the file. With the first method you have to specifically use `.close()` to close the file. With the second, the file closes automatically, so no `.close()` is required:

```
# - Basic syntax to open, read, and display file contents.
f = open('quotes.txt')
filecontents = f.read()
print(filecontents)
# Returns True if the file is closed, otherwise alse.
print('File is closed: ', f.closed)

# Closes the file.
f.close() #Close the file.
print() # Print a blank line.

# ---------------- Contextual syntax
with open('quotes.txt') as f:
    filecontents = f.read()
    print(filecontents)

# The unindented line below is outside the with... block;
print('File is closed: ', f.closed)
```

The output of this app is as follows. At the end of the first output, `.closed` is `False` because it's tested before the `close()` actually closes the file. At the end of the second output, `.closed` is `True`, without executing a `.close()`, because leaving the code that's indented under the `with:` line closes the file automatically.

```
I've had a perfectly wonderful evening, but this wasn't it.
Groucho Marx
The difference between stupidity and genius is that genius has its limits.
Albert Einstein
```

```
We are all here on earth to help others; what on earth the others are here for,
   I have no idea.
W. H. Auden
Ending a sentence with a preposition is something up with I will not put.
Winston Churchill

File is closed:  False
```

```
I've had a perfectly wonderful evening, but this wasn't it.
Groucho Marx
The difference between stupidity and genius is that genius has its limits.
Albert Einstein
We are all here on earth to help others; what on earth the others are here for,
    I have no idea.
W. H. Auden
Ending a sentence with a preposition is something up with I will not put.
Winston Churchill

File is closed:  True
```

For the rest of this chapter we stick with the contextual syntax because it's generally the preferred and recommended syntax, and a good habit to acquire right from the start.

The previous example works fine because `quotes.txt` is a really simple text file that contains only ASCII characters — the kinds of letters, numbers, and punctuation marks that you can type from a standard keyboard for the English language.

```
with open('happy_pickle.jpg') as f:
    filecontents = f.read()
    print(filecontents)
```

Attempting to run this code results in the following error:

```
UnicodeDecodeError: 'charmap' codec can't decode byte 0x90 in position 40:
   character maps to <undefined>
```

This isn't the most helpful message in the world. Suppose you try to open `names.txt`, which (one would assume) is a text file like `quotes.txt`, using this code:

```
with open('names.txt') as f:
    filecontents = f.read()
    print(filecontents)
```

You run this code, and again you get a strange error message, like this:

```
UnicodeDecodeError: 'charmap' codec can't decode byte 0x81 in position 45:
    character maps to <undefined>
```

So what the heck is going on here?

The first problem is caused by the fact that the file is a .jpg, a graphic image, which means it's a binary file, not a text file. So to open this one, you need a `b` in the mode. Or use `rb`, which means *read binary*, like this:

```
with open('happy_pickle.jpg', 'rb') as f:
    filecontents = f.read()
    print(filecontents)
```

Running this code doesn't generate an error. But what it does show doesn't look anything like the actual picture. The output from this code is a lot of stuff that looks something like this:

```
\x07~}\xba\xe7\xd2\x8c\x00\x0e|\xbd\xa8\x12l+\xca\xf7\xae\xa5\x9e^\x8d\x89
\x7f\xde\xb4f>\x98\xc7\xfc\xcf46d\xcf\x1c\xd0\xa6\x98m$\xb6(U\x8c\xa6\x83
\x19\x17\xa6>\xe6\x94\x96|g\'4\xab\xdd\xb8\xc8=\xa9[\x8b\xcc`\x0e8\xa3
\xb0;\xc6\xe6\xbb(I.\xa3\xda\x91\xb8\xbd\xf2\x97\xdf\xc1\xf4\xefI\xcdy
\x97d\x1e`;\xf64\x94\xd7\x03
```

If we open `happy_pickle.jpg` in a graphics app or in VS Code, it looks nothing like that gibberish. Instead, it looks like Figure 1-3.

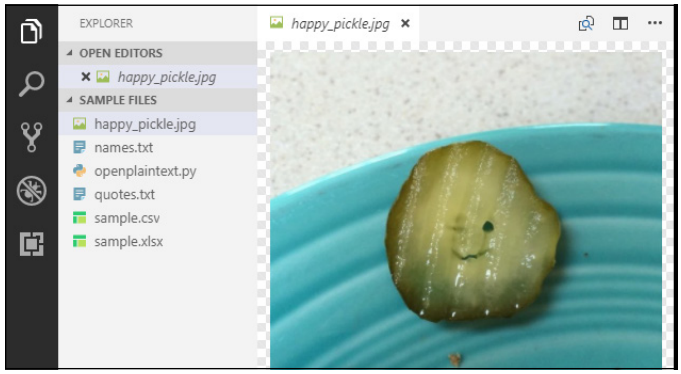


FIGURE 1-3: How `happy_pickle.jpg` is supposed to look.

So why does it look so messed up in Python? That's because `print()` just shows the raw bytes that make up the file. It has no choice because it's not a graphics app. This is not a problem or issue, just not a good way to work with a .jpg file right now.

The problem with `names.txt` is something different. That file is a text file (.txt) just like `quotes.txt`. But if you open it and look at its content, as in Figure 1-4, you'll notice it has a lot of unusual characters in it characters that you don't normally see in ASCII, the day-to-day numbers, letters, and punctuation marks you see on your keyboard.

FIGURE 1-4: Names.txt is text, but with lots of non-English characters.

This `names.txt` file is indeed a text file, but all those fancy-looking characters in there tell you it's not a simple ASCII text file. More likely it's a UTF-8 file, which is basically a text file that uses more than just the standard ASCII text characters. To get this file to open, you have to tell Python to "expect" UTF-8 characters by using `encoding='utf-8'` in the `open()` statement, as in Figure 1-5.

Figure 1-5 shows the results of opening `names.txt` as a text file for reading with the addition of the `encoding =`. The output from Python accurately matches what's in the `names.txt` file.

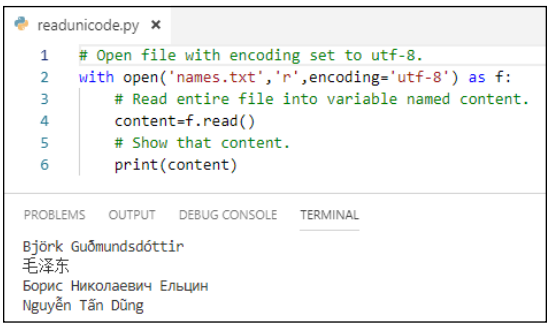


FIGURE 1-5: Contents of names.txt displayed.

Not all terminal windows in VS Code show Unicode characters correctly, and these may be replaced with generic question mark symbols on your screen. But don't worry about it, in real life you won't care about output in the Terminal window. Here, all that matters is that you're able to open the file without raising an exception.

When it comes to opening files, there are three things you need to be aware of:

- If it's a plain text file (ASCII) it's sufficient to use `r` or nothing as the mode.
- If it's a binary file, you have to specify `b` in the mode.
- If it's a text file with fancy characters, you most likely need to open it as a text file but with `encoding` set to `utf-8` in the `open()` statement.

WHAT IS UTF-8?

UTF-8 is short for Unicode Transformation Format, 8-bit, and is a standardized way for representing letters and numbers on computers. The original ASCII set of characters, which contains mostly uppercase and lowercase letters, numbers, and punctuation marks, worked okay in the early days of computing. But when you start bringing other languages into the mix, these characters are just not enough. Many different standards for dealing with other languages have been proposed and accepted over the years since. Of those, UTF-8 has steadily grown in use whereas most others declined. Today, UTF-8 is pretty much the standard for all things Internet, and so it's a good choice if you're ever faced with having to choose a character set for some project.

If you're looking for more history or technical info on UTF-8, take a look at these web pages:

- `https://www.w3.org/International/questions/qa-what-is-encoding`
- `https://pythonconquerstheuniverse.wordpress.com/2010/05/30/unicode-beginners-introduction-for-dummies-made-simple/`
- `https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/`

If you really get stuck trying to open a file that's supposed to be UTF-8 but isn't cooperating, Google *convert file to utf-8 encoding*. Then look for a web page or app that will work with your operating system to make the conversion.

Reading a File's Contents

Earlier in this chapter, you saw how you can read all of an open file's contents using `.read()`. But that's not the only way to do it. You actually have three choices:

- **`read([size])`:** Reads in the entire file if you leave the parentheses empty. If you specify a size inside the parentheses, it reads that many characters (for a text file) or that many bytes (for a binary file).
- **`readline()`:** Reads one line of content from a text file (the line ends wherever there's a newline character).
- **`readlines()`:** Reads all the lines of a text file into a list.

People don't type binary files, so any newline characters that happen to be in there would be arbitrary. Therefore, `readline()` and `readlines()` are useful only for text files.

Both the `read()` and `readline()` methods read in the entire file at once. The only real difference is that `read` reads it in as one big chunk of data, whereas `readlines()` reads it in one line at a time and stores each line as an item in a list. For example, the following code opens `quotes.txt`, reads in all the content, and then displays it

```
with open('quotes.txt') as f:
# Read in entire file
    content = f.read()
    print(content)
```

The `content` variable ends up storing a copy of everything that's in the CSV file. Printing this variable shows the contents. It's broken into multiple lines exactly as the original file is because the newline character at the end of each line in the file also starts a new line on the screen when printing the content.

Here is the same code using `readlines()` rather than `read`:

```
with open('quotes.txt') as f:
    content = f.readlines()
    print(content)
```

The output from this code is

```
["I've had a perfectly wonderful evening, but this wasn't it.\n", 'Groucho
   Marx\n', 'The difference between stupidity and genius is that genius has its
   limits.\n', 'Albert Einstein\n', 'We are all here on earth to help others;
```

```
    what on earth the others are here for, I have no idea.\n', 'W. H. Auden\n',
    'Ending a sentence with a preposition is something up with I will not put.\n',
    'Winston Churchill\n']
```

The square brackets surrounding the output tell you that it's a list. Each item in the list is surrounded by quotation marks and separated by commas. The `\n` at the end of each item is the newline character that ends the line in the file.

Unlike `readlines()` (plural), `readline()` reads just one line from the file. The line extends from the beginning of the file to just after the first newline character. Executing another `readline()` reads the next line in the file, and so forth. For example, suppose you run this code:

```
with open('quotes.txt') as f:
    content = f.readline()
    print(content)
```

The output is

```
I've had a perfectly wonderful evening, but this wasn't it.
```

Executing another `readline()` after this would read the next line. As you may guess, when it comes to `readline()` and `readlines()`, you're likely to want to use some loops to access all the data in a way where you have some more control.

Looping through a File

You can loop through a file using either readlines() or readline(). The readlines() method always reads in the file as a whole. Which means if the file is very large, you may run out of memory (RAM) before the file has been read in. But if you know the size of the file and it's relative small (maybe a few hundred rows of data or less), readlines() is a speedy way to get all the data. Those data will be in a list. So you will then loop through the list rather than a file. You can also loop through binary files, but they don't have lines of text like text files do. So those you read in "chunks" as you'll see at the end of this ection.

Looping with readlines()

When you read a file with `readlines()`, you read the entire file in one fell swoop as a list. So you don't really loop through the file one row at a time. Rather, you

loop through the list of items that `readlines()` stores in memory. The code to do so looks like this:

```
with open('quotes.txt') as f:
    # Reads in all lines first, then loops through.
    for one_line in f.readlines():
        print(one_line)
```

If you run this code, the output will be double-spaced because each list item ends with a newline, and then `print` always adds its own newline with each pass through the loop. If you want to retain the single spacing, add `end=''` to the `print` statement (make sure you use two single or double quotation marks with nothing in between after the =). Here's an example:

```
with open('quotes.txt') as f:
    # Reads in all lines first, then loops through.
    for one_line in f.readlines():
        print(one_line, end='')
```

The output from this code is:

```
I've had a perfectly wonderful evening, but this wasn't it.
Groucho Marx
The difference between stupidity and genius is that genius has its limits.
Albert Einstein
We are all here on earth to help others; what on earth the others are here for,
    I have no idea.
W. H. Auden
Ending a sentence with a preposition is something up with I will not put.
Winston Churchill
```

Let's say you're pretty good with this, except that if the line to be printed is a name, you want to indent the name by a space and put an extra blank line beneath it. How could you do that? Well, Python has a built-in `enumerate()` function that, when used with a list, counts the number of passes through the loop, starting at zero. So instead of the `for:` loop shown in the previous example, you write it as `for one_line in enumerate(f.readlines()):`. With each pass through the loop `one_line[0]` contains the number of that line, whereas `one_line[1]` contains its content . . . the text of the line. With each pass through the loop, you can see whether the counter is an even number (that number % 2 will be zero for even numbers, because % returns the remainder after division). So you could write the code this way:

```
with open('quotes.txt') as f:
    # Reads in all lines first, then loops through.
    # Count each line starting at zero.
```

```
    for one_line in enumerate(f.readlines()):
        # If counter is even number, print with no extra newline
        if one_line[0] % 2 == 0:
            print(one_line[1], end='')
        # Otherwise print a couple spaces and an extra newline.
        else:
            print('  ' + one_line[1])
```

The output from this will be as follows:

```
I've had a perfectly wonderful evening, but this wasn't it.
  Groucho Marx

The difference between stupidity and genius is that genius has its limits.
  Albert Einstein

We are all here on earth to help others; what on earth the others are here for,
   I have no idea.
  W. H. Auden

Ending a sentence with a preposition is something up with I will not put.
  Winston Churchill
```

Looping with readline()

If you aren't too sure about the size of the file you're reading, or the amount of RAM in the computer running your app, using `readlines()` to read in an entire file can be risky. Because if there isn't enough memory to hold the entire file, the app will crash when it runs out of memory. To play it safe, you can loop through the file one line at a time so only one line of content from the file is in memory at any given time. To use this method you can open the file, read one line and put it in a variable. Then loop through the file *as long as* (`while`) the variable isn't empty. Because each line in the file contains some text, the variable won't be empty until after the very last line is read. Here is the code for this approach to looping:

```
with open('quotes.txt') as f:
    one_line = f.readline()
    while one_line:
        print(one_line, end='')
        one_line = f.readline()
```

For larger files this would be the way to go because at no point are you reading in the entire file. The only danger there is forgetting to do the `.readline()` inside the loop to advance to the next pointer. Failure to do this creates in infinite loop that prints the first line over and over again. If you ever find yourself in this

situation, pressing Ctrl+C in the terminal window where the code us running will stop the loop.

So what about if you want to do a thing like in `readlines()` where you indent and print an extra blank line after peoples' names? In this example, you really can't use `enumerate` with the `while` loop. But you could set up a simple counter yourself, starting at 1 if you like, and increment it by 1 with each pass through the loop. Indent and do the extra space on even-numbered lines like this:

```
# Store a number to use as a loop counter.
counter = 1
# Open the file.
with open('quotes.txt') as f:
    # Read one line from the file.
    one_line = f.readline()
    # As long as there are lines to read...
    while one_line:
        # If the counter is an even number, print a couple spaces.
        if counter % 2 == 0:
            print('  ' + one_line)
        # Otherwise print with no newline at the end.
        else:
            print(one_line,end='')
        # Increment the counter
        counter += 1
        # Read the next line.
        one_line = f.readline()
```

The output from this loop is the same as for the second `readlines()` loop in which each author's name is indented and followed by an extra blank line caused by using `print()` without the `end=''`.

Appending versus overwriting files

Any time you work with files it's important to understand the difference between *write* and *append*. If a file contains information already, and you open it in write mode, then write more to it, your new content will actually overwrite (replace) whatever is already in the file. There is no undo for this. So if the content of the file is important, you want to make sure you don't make that mistake. To add content to a file, open the file in append (`a`) mode, then use `.write` to write to a file.

As a working example, suppose you want to add the name `Peña Calderón` to the `names.txt` file used in the previous section. This name, as well as the names that are already in this file, use special characters beyond the English alphabet, which tells you to make sure you set the encoding to UTF-8. Also, if you want each name

in the file on a separate line, you should add a `\n` (newline) to the end of whatever name you're adding. So your code should look like this:

```
# New name to add with \n to mark end of line.
new_name = 'Peña Calderón\n'
# Open names.txt in append mode with encoding.
with open('names.txt', 'a', encoding='utf-8') as f:
    f.write(new_name)
```

To verify that it worked, start a new block of code, with no indents, so `names.txt` file closes automatically. Then open this same file in read (`r`) mode and view its contents. Figure 1-6 shows all the code to add the new name and the code to display the `names.txt` file after adding this name.

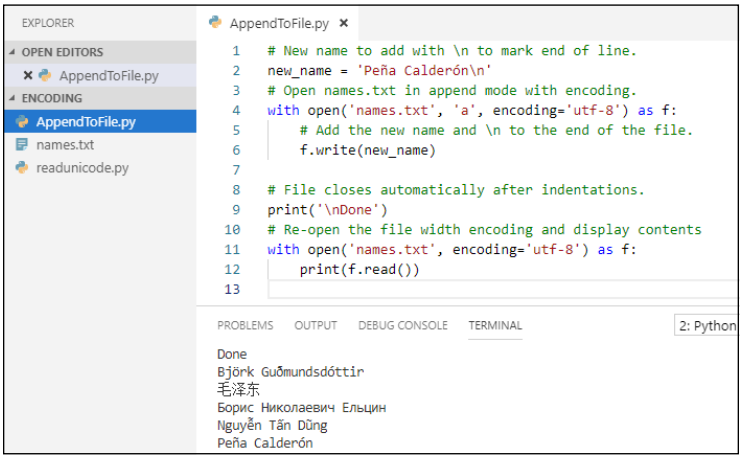


FIGURE 1-6: A new name appended to the end of the `names.txt` file.

TIP

Typing special characters like *ñ* and *ó* usually involves holding down the Alt key and typing 3 or 4 numeric digit; for example, Alt+164 for *ñ* or Alt+0243 for *ó*. Exactly how you do this depends on the operating system and editor you're using. But as a rule you can google a phrase like *type tilde n on Windows* or *type accented o on Mac* and so on to find out exactly what you need to do to type a special character.

Using tell() to determine the pointer location

Whenever you loop through a file, its contents are read top-to-bottom, left-to-right. Python maintains an invisible pointer to keep track of where it is in the file. When you're reading a text file with `readline()`, this is always the character position of the next line in the file.

If all you've done so far is open the file, the character position will be zero, the very start of the file. Each time you execute a `readline()`, the pointer advances to the start of the next row. Here is some code to illustrate; its output is below the code:

```
with open('names.txt', encoding='utf-8') as f:
    # Read first line to get started.
    print(f.tell())
    one_line = f.readline()
    # Keep reading one line at a time until there are no more.
    while one_line:
        print(one_line[:-1], f.tell())
        one_line = f.readline()
```

```
0
Björk Guðmundsdóttir 25
毛泽东 36
Борис Николаевич Ельцин 82
Nguyễn Tấn Dũng 104
Peña Calderón 121
```

The first zero is the position of the pointer right after the file is opened. The 25 at the end of the next line is the position of the pointer after reading this first line. The 36 at the end of the next line is the pointer position at the end of the second line, and so forth, until the 121 at the end, when the pointer is at the very end of the file.

If you try to do this with `readlines()` you get a very different result. Here is the code:

```
with open('names.txt', encoding='utf-8') as f:
    print(f.tell())
    # Reads in all lines first, then loops through.
    for one_line in f.readlines():
        print(one_line[:-1],f.tell())
```

Here is the output:

```
0
Björk Guðmundsdóttir 121
毛泽东 121
Борис Николаевич Ельцин 121
Nguyễn Tấn Dũng 121
Peña Calderón 121
```

The pointer starts out at position zero, as expected. But each line shows a 121 at the end. This is because `readlines()` reads in the entire file when executed,

leaving the pointer at the end, position 121. The loop is actually looping through the copy of the file that's in memory; it's no longer reading through the file.

If you try to use `.tell()` with the super-simple `read()` loop shown here:

```
with open('names.txt', encoding='utf-8') as f:
    for one_line in f:
        print(one_line, f.tell())
```

. . . it won't work in Windows. So if for whatever reason you need to keep track of where the pointer is in some external text file you're reading, make sure you use a loop with `readline()`.

Moving the pointer with seek()

Although the `tell()` method tells you where the pointer is in an external file, the `seek()` method allows you to reposition the pointer. The syntax is:

```
file.seek(position[,whence])
```

Replace *`file`* with the variable name of the open file. Replace *`position`* to indicate where you want to put the pointer. For example, 0 to move it back to the top of the file. The *`whence`* is optional and you can use it to indicate to which place in the file the position should be calculated. Your choices are:

- **0**: Set position relative to the start of the file.
- **1**: Set position relative to the current pointer position.
- **2**: Set position relative to the end of the file. Use a negative number for *`position`*.

If you omit the *`whence`* value, it defaults to zero.

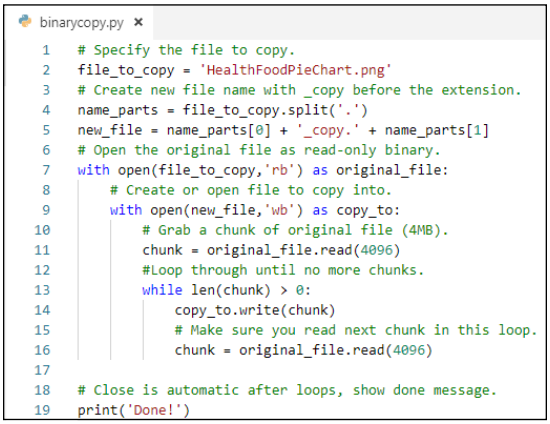
By far the most common use of `seek` is to just reset the pointer back to the top of the file for another pass through the file. The syntax for this is simply `.seek(0)`.

Reading and Copying a Binary File

Suppose you have an app that somehow changes a binary file,and you want to always work with a copy of the original file to play it safe. Binary files can be huge, so rather than opening it all at once and risking running out of memory, you can

read it in chunks and write it out in chunks. Binary files have no human-readable content in them. Nor do they have lines of text. So `readline()` and `readlines()` aren't a good choice for looping through binary files. But you can use `.read()` with a specified size to achieve a similar result with binary files.

Figure 1-7 shows a file named `binarycopy.py` that will make a copy of any binary file. We'll take you through it step-by-step so you can understand how it works.


binarycopy.py

```
# Specify the file to copy.
file_to_copy = 'HealthFoodPieChart.png'
# Create new file name with _copy before the extension.
name_parts = file_to_copy.split('.')
new_file = name_parts[0] + '_copy.' + name_parts[1]
# Open the original file as read-only binary.
with open(file_to_copy,'rb') as original_file:
    # Create or open file to copy into.
    with open(new_file,'wb') as copy_to:
        # Grab a chunk of original file (4MB).
        chunk = original_file.read(4096)
        #Loop through until no more chunks.
        while len(chunk) > 0:
            copy_to.write(chunk)
            # Make sure you read next chunk in this loop.
            chunk = original_file.read(4096)

# Close is automatic after loops, show done message.
print('Done!')
```

FIGURE 1-7: The `binarycopy.py` file copies any binary file.

The first step is to specify the file you want to copy. We chose `happy_pickle.jpg`, which, as you can see in the figure, is in the same folder as the `binarycopy.py` folder:

```
# Specify the file to copy.
file_to_copy = 'happy_pickle.jpg'
```

To make an empty file to copy into, you first need a filename for the file. The following code takes care of that:

```
# Create new file name with _copy before the extension.
name_parts = file_to_copy.split('.')
new_file = name_parts[0] + '_copy.' + name_parts[1]
```

The first line after the copy splits the existing filename in two at the dot, so `name_parts[0]` contains *`happy_pickle`* and `name_parts[1]` contains *`png`*. Then the `new_file` variable gets a value consisting of the first part of the name with `_copy` and a dot attached, and then the last part of the name. So after this line executes, the `new_file` variable contains *`happy_pickle_copy.png`*.

In order to make the copy, you can open the original file in `rb` mode (read, binary file). The open the file into which you want to copy the original file in `wb` mode (write, binary). With *write*, Python creates a file of this name if the file doesn't already exist. If the file does exist, then Python opens it with the pointer set at 0, so anything that you write into the file will *replace* (not *add to*) the existing file. In the code you can see that we used `original_file` as the variable name from which to copy, and `copy_to` as the variable name of the file into which you copy data. Indentations, as always, are critical:

```
# Open the original file as read-only binary.
with open(file_to_copy,'rb') as original_file:
    # Create or open file to copy into.
    with open(new_file,'wb') as copy_to:
```

If you use `.read()` to read in the entire binary file, you run the risk of it being so large that it overwhelms the computer's RAM and crashes the program. To avoid this, we've written this program to read in a modest 4MB (4,096 kilobytes) of data at a time. This 4MB chunk is stored in a variable named `chunk`:

```
        # Grab a chunk of original file (4MB).
        chunk = original_file.read(4096)
```

The next line sets up a loop that keeps reading one chunk at a time. The pointer is automatically positioned to the next chunk with each pass through the loop. Eventually, it will hit the end of the file where it can't read anymore. When this happens, `chunk` will be empty, meaning it has a length of 0. So this loop keeps going through the file until it gets to the end:

```
        #Loop through until no more chunks.
        while len(chunk) > 0:
```

Within the loop, the first line copies the last-read chunk into the `copy_to` file. The second line reads the next 4MB chunk from the original file. And so it goes until everything from `original_file` has been copied to the new file:

```
            copy_to.write(chunk)
            # Make sure you read in the next chunk in this loop.
            chunk = original_file.read(4096)
```

All the indentations stop after this line. So when the loop is done, the files close automatically, and the last line just shows the word `Done!`

```
print('Done!')
```

Figure 1-8 shows the results of running the code. The terminal pane simply shows `Done!`. But as you can see, there's now a file named `happy_pickle_copy.jpg` in the folder. Opening this file will prove that it is an exact copy of the original file.

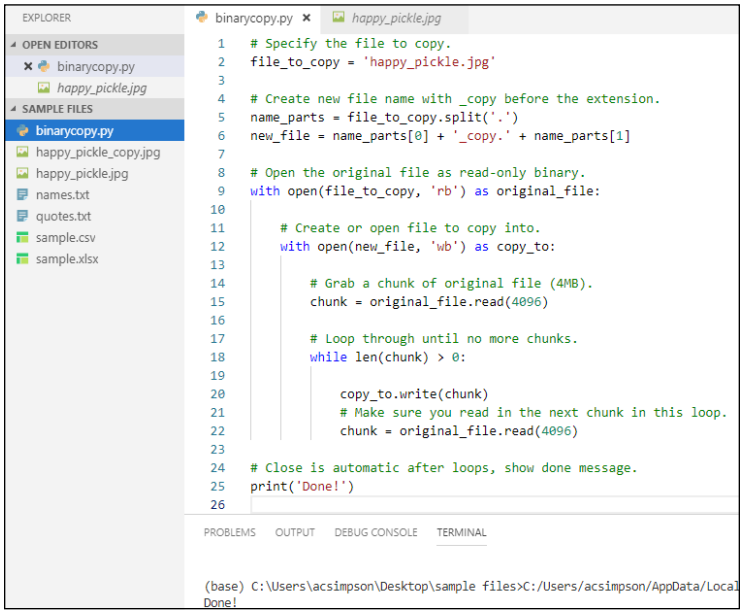


FIGURE 1-8: Running `binarycopy.py` added `happy_pickle_copy.jpg` to the folder.

Conquering CSV Files

CSV (short for comma separated values) is a widely used format for storing and transporting tabular data. *Tabular* means that it can generally be displayed in a table format consisting of rows and columns. In a spreadsheet app like Microsoft Excel, Apple Numbers, or Google Sheets, the tabular format is pretty obvious, as shown in Figure 1-9.

	A	B	C	D	E
1	**Full Name**	**Birth Year**	**Date Joined**	**Is Active**	**Balance**
2	Angst, Annie	1982	1/11/2011	TRUE	$300.00
3	Bónañas, Barry	1973	2/11/2012	FALSE	-$123.45
4	Schadenfreude, Sandy	2004	3/3/2003	TRUE	$0.00
5	Weltschmerz, Wanda	1995	4/24/1994	FALSE	$999,999.99
6	Malaise, Mindy	2006	5/5/2005	TRUE	$454.01
7	O'Possum, Ollie	1987	7/27/1997	FALSE	-$1,000.00
8					
9	Pusillanimity, Pamela	1979	8/8/2008	TRUE	$12,345.67

FIGURE 1-9: A CSV file in Microsoft Excel.

Without the aid of some special program to make the data in the file display in a neat tabular format, each row is just a line in the file. And each unique value is separated by a comma. For instance, opening the file shown in Figure 1-10 in a simple text editor like Notepad or TextEdit shows what's really stored in the file.

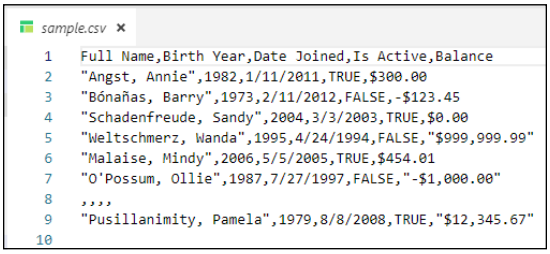


FIGURE 1-10: A CSV file in a text editor.

In the text editor, the first row, often called the *header*, contains the column headings, or *field names*, that appear across the first row of the spreadsheet. If you look at the names in the second example, the raw CSV file, you'll notice that they're all enclosed in quotation marks, like this:

```
"Angst, Annie"
```

In real life, they may be single quotation marks, or double, as shown. But either way, they indicate that the stuff *between* the quotation marks is all one thing. In other words, the comma between the last and first name is all part of the name. This comma isn't the start of a new column. So the first two columns on this row one are

```
"Angst, Annie", 1982
```

. . . and not

```
Angst, Annie
```

The same is true in all other rows: The name enclosed in quotation marks (including commas) is just one name, not two separate columns of data.

If any of the strings contains an apostrophe, which is the same character as a single quotation mark, then you have to use double quotation marks around the string. Because if you do it like this:

```
'O'Henry, Harry'
```

The first part of the string looks like `'O'` and then Python won't know what to do with the text after the second single quotation mark. Using double-quotation marks alleviates any confusion because there are no other double quotation marks contained within the name:

"O'Henry, Harry"

Figure 1-10 also contains a few other problems that you may encounter when working with CSV files on your own. For example, the `Bónañas, Barry` name contains some non-ASCII characters. The second-to-last row just contains a bunch of commas. If in a CSV file a cell is missing its data, you just put the comma that ends this cell with nothing to its left. The Balance column has dollar signs and commas in the numbers, which don't work with the Python `float` data type. We talk about how to deal with all of this in the sections to follow.

Although it would certainly be possible to work with CSV files using just what you've learned so far, it's a lot quicker and easier if you use the `csv` module, which you already have. To use it, just put this near the top of your program:

```
import csv
```

Remember, this doesn't bring in a CSV *file*. It just brings in the pre-written code that makes it easier for you to work with CSV files in your own Python code.

Opening a CSV file

Opening a CSV file is really no different from opening any other file. Just remember that if the file contains special characters, you need to include the `encoding='utf-8'` to avoid an error message. Optionally, when importing data, you probably don't want to read in the newline character at the end of each row, so you can add `newline=''` to the `open()` statement. Here is how you might comment and code this, except you'd replace `sample.csv` with the path to the CSV file you want to open:

```
# Open CSV file with UTF-8 encoding, don't read in newline characters.
with open('sample.csv', encoding='utf-8', newline='') as f:
```

To loop through a CSV file, you can use the built-in reader function, which reads one row with execution. Again, the syntax is pretty simple, as shown in the following code. Replace `f` with whatever name you used at the end of your `open` statement (without the colon at the very end).

```
reader = csv.reader(f)
```

Although it's entirely optional, you can also count rows as you go. Just put everything to the right of the = in an `enumerate()`, as shown in the following (where we've also added a comment above the code):

```
# Create a CVS row counter and row reader.
reader = enumerate(csv.reader(f))
```

Next, you can set up your loop to read one row at a time. Because you put an enumerator on it, you can use two variable names in your `for`: the first one (which we'll call `i`) will keep track of the counter (which starts at zero and increases by 1 with each pass through the loop). The second variable, `row`, will contain the entire row of data from the CSV file:

```
    # Loop through one row at a time, i is counter, row is entire row.
    for i, row in reader:
```

You could start with this followed by a `print()` function to print the value of `i` and `row` with each pass through the loop, like this:

```
import csv
# Open CSV file with UTF-8 encoding, don't read in newline characters.
with open('sample.csv', encoding='utf-8', newline='') as f:
    # # Create a CVS row counter and row reader.
    reader = enumerate(csv.reader(f))
    # Loop through one row at a time, i is counter, row is entire row.
    for i, row in reader:
        print(i, row)
print('Done')
```

The output from this, using the `sample.csv` file described earlier as input is as follows:

```
0 ['\ufeffFull Name', 'Birth Year', 'Date Joined', 'Is Active', 'Balance']
1 ['Angst, Annie', '1982', '1/11/2011', 'TRUE', '$300.00']
2 ['Bónañas, Barry', '1973', '2/11/2012', 'FALSE', '-$123.45']
3 ['Schadenfreude, Sandy', '2004', '3/3/2003', 'TRUE', '$0.00']
4 ['Weltschmerz, Wanda', '1995', '4/24/1994', 'FALSE', '$999,999.99']
5 ['Malaise, Mindy', '2006', '5/5/2005', 'TRUE', '$454.01']
6 ["O'Possum, Ollie", '1987', '7/27/1997', 'FALSE', '-$1,000.00']
7 ['', '', '', '', '']
8 ['Pusillanimity, Pamela', '1979', '8/8/2008', 'TRUE', '$12,345.67']
```

Notice how the row of column names is row zero. The weird `\ufeff` before Full Name in that row is called the Byte Order Mark (BOM) and it's just something Excel sticks in there. Typically you don't care what's in that first row because the real data doesn't start until the next row down. So don't give the BOM a second thought, it's of no value to you, nor is it doing any harm.

Notice how each row is actually a list of five items separated by commas. In your code you can refer to each column by its position. For example, `row[0]` is the first column in the row (the person's name). Then, `row[1]` is the birth year, `row[2]` is date joined, `row[3]` is whether the person is active, and `row[4]` is the balance. All the data in the CSV file are strings — even if they don't look like strings. But anything and everything coming from a CSV file is a string because a CSV file is a type of text file, and a text file contains only strings (text), and no integers, dates, Booleans, or floats.

In your app, it's likely that you will want to convert the incoming data to Python data types, so you can work with them more effectively, or perhaps even transfer them to a database. In the next sections, we look at how to do the conversion for each data type.

Converting strings

Technically, you don't have to convert anything from the CSV file to a string. But you may want to chop it up a bit, or deal with empty strings in some way, so there are some things you can do. First, as we mentioned earlier, we care only about the data here, not that first row. So inside the loop you can start with an `if` that doesn't do anything if the current row is row zero. Replace the `print(i,row)` like this:

```
# Row 0 is just column headings, ignore it.
if i > 0:
    full_name = row[0].split(',')
    last_name=full_name[0].strip()
    first_name=full_name[1].strip()
```

This code says "So long as we're not looking at the first row, create a variable named `fullname` and store in it whatever is in the first column split into two separate values at the comma." After that line executes, `full_name[0]` contains the person's last name, which we then put into a variable named `first_name`, and `full_name[1]` contains the person's first name, which we put into a variable named `first_name`. But if you run the code that way, it will bomb, because row 7 doesn't have a name, and Python can't split an empty string at a comma (because the empty string contains no comma).

To get around this, you can tell Python to *try* to split the name at the comma, if it can. But if it bombs out when trying, just store and empty string in the `full_name`, `last_name`, and `first_name` variables. Here's that code with some extra comments thrown in to explain all that's going on. Instead of printing `i` and the whole row, the code just prints the first name and last name (and nothing for the row whose information is missing). You can see the output below the code below.

```
import csv
# Open CSV file with UTF-8 encoding, don't read in newline characters.
with open('sample.csv', encoding='utf-8', newline='') as f:
    # # Create a CVS row counter and row reader.
    reader = enumerate(csv.reader(f))
    # Loop through one row at a time, i is counter, row is entire row.
    for i, row in reader:
        # Row 0 is just column headings, ignore it.
        if i > 0:
            # Whole name split into two at comma.
            try:
                full_name = row[0].split(',')
                # Last name, strip extra spaces.
                last_name=full_name[0].strip()
                # First name, strip extra spaces.
                first_name=full_name[1].strip()
            except IndexError:
                full_name = last_name = first_name = ""
            print(first_name, last_name)
print('Done!')
```

```
Annie Angst
Barry Bónañas
Sandy Schadenfreude
Wanda Weltschmerz
Mindy Malaise
Ollie O'Possum

Pamela Pusillanimity
Done!
```

Converting to integers

The second column in each row, `row[1]`, is the birth year. So long as the string contains something that can be converted to a number, you can use the simple built-in `int()` function to convert it to an integer. We do have a problem in row 7 though, which is empty. Python won't automatically convert this to a zero, you have to help it along a bit. Here is the code for that:

```
            # Birth year integer, zero for empty string.
             birth_year= int(row[1] or 0)
```

The code looks surprisingly simple, but this is the beauty of Python: It *is* surprisingly simple. This line of code says "create a variable named `birth_year` and put in it the second column value, if you can, or if there is nothing to convert to an integer, then just put in a zero."

Converting to date

The third column in our CSV file, `row[2]`, is the date joined, and it appears to have a reasonable date in each row (except the row whose data is missing). To convert this to a date, you first need to import the `datetime` module by adding `import datetime as dt` up near the top of the program. Then the simple conversion is just:

```
date_joined = dt.datetime.strptime(row[2],"%m/%d/%Y").date()
```

There's a lot going on there, so let us unpack it a bit. First, you create a variable named `date_joined`. The `strptime` means "string parse for date time." The `[row,2]` means the third column (because the first column is always column 0). The `"%m/%d/%Y"` tells `strptime` that the string date contains the month, followed by a slash, the day of the month, followed by a slash, and then the four-digit year (uppercase `%Y`). The `.date()` at the very end means "just the date; there is no time here to parse."

One small problem. When it gets to the row whose date is missing, this will bomb. So once again we'll use a `try ...` to do the date, and if it can't come up with a date, then put in the value `None`, which is Python's word for an empty object.


REMEMBER

In Python, `datetime` is a class, so any date and time you create is actually an object (of the `datetime` type). You don't use `''` for an empty object, `''` is for an empty string. Python uses the word `None` for an empty object.

Here is the code as it stands now with the `import` up top for the `datetime`, and `try ... except` for converting the string date to a Python date:

```
import csv
import datetime as dt
# Open CSV file with UTF-8 encoding, don't read in newline characters.
with open('sample.csv', encoding='utf-8', newline='') as f:
    # # Create a CVS row counter and row reader.
    reader = enumerate(csv.reader(f))
    # Loop through one row at a time, i is counter, row is entire row.
    for i, row in reader:
        # Row 0 is just column headings, ignore it.
        if i > 0:
            # Whole name split into two at comma.
            try:
                full_name = row[0].split(',')
                # Last name, strip extra spaces.
                last_name = full_name[0].strip()
                # First name, strip extra spaces.
                first_name = full_name[1].strip()
```

```
            except IndexError:
                full_name = last_name = first_name = ""
            # Birth year integer, zero for empty string.
            birth_year = int(row[1] or 0)
            # Date_joined is a date.
            try:
                date_joined = dt.datetime.strptime(row[2], "%m/%d/%Y").date()
            except ValueError:
                date_joined = None
            print(first_name, last_name, birth_year, date_joined)
print('Done!')
```

Here is the output from this code, which now prints `first_name`, `last_name`, `birth_year`, and `date_joined` with each pass through the data rows in the table:

```
Annie Angst 1982 2011-01-11
Barry Bónañas 1973 2012-02-11
Sandy Schadenfreude 2004 2003-03-03
Wanda Weltschmerz 1995 1994-04-24
Mindy Malaise 2006 2005-05-05
Ollie O'Possum 1987 1997-07-27
  0 None
Pamela Pusillanimity 1979 2008-08-08
Done!
```

Converting to Boolean

The fourth column, `row[3]` in each row contains `TRUE` or `FALSE`. Excel uses all uppercase letters like this, and that is automatically carried over to the CSV file when saving as CSV in Excel. Python uses initial caps, `True` and `False`. Python has a simple `bool()` function for making this conversion. And it won't bomb out when it hits an empty cell . . . it just considers that cell `False`. So this conversion can be as simple as this:

```
            # is_active is a Boolean, automatically False for empty string.
            is_active=bool(row[3])
```

Converting to floats

The fifth column in each row contains the balance, which is a dollar amount. In Python, you want this to be a float. But there's a problem right off the bat. Python floats can't contain a dollar sign ($) or a comma (,). So the first thing you need to do is remove those from the string. Also, you can't have any accidental leading or

trailing spaces. These you can easily remove with the `strip()` method. This line creates a variable named `str_balance` (which is still a string), but with the dollar sign, comma, and any trailing leading spaces removed:

```
            # Remove $, commas, leading trailing spaces.
            str_balance = (row[4].replace('$','').replace(',','')).strip()
```

You can read this second line as "the new string named balance consists of whatever is in the fifth column after replacing any dollar signs with nothing, and replacing any commas with nothing, and stripping off all leading and trailing spaces." Below that line, you can add a comma and then another line to create a float named `balance` that uses the built-in `float()` method to convert the `str_balance` string into a float. Like `int()`, `float()` has its own built-in exception handler, which, if it can't make sense of the thing it's trying to convert to a float, stores a zero as the value of the float. The code in Figure 1-11 shows everything in place, including a `print()` line that displays the values of all five columns after conversion.

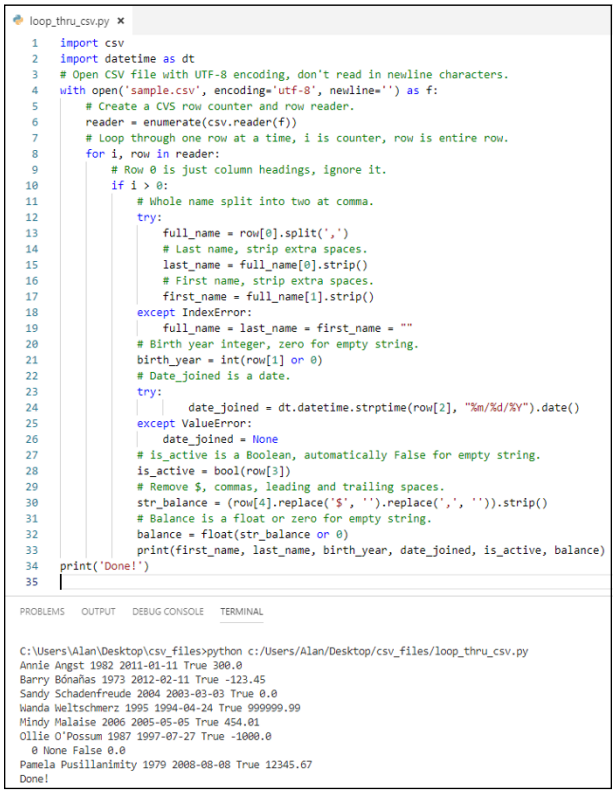


FIGURE 1-11: Reading a CSV file and converting to Python data types.

USING REGULAR EXPRESSIONS IN PYTHON

Even though this book assumes you're not already familiar with other programming languages, some readers inevitably will be, and some of those are likely to ask why we didn't use a *regular expression* to remove the dollar sign and comma from the `balance` instead of the `replace()` method. The answer to this would be, "Because you're not required to do it that way, and not everyone reading this book is aware that a thing called *regular expressions* is available in most programming languages."

But if you happen to be a person who was thinking of asking this question, the first thing to know is that regular expressions aren't built-in to Python. So if you want to use them, you need to put an `import re` at the top of your code. In this particular example, which just uses the substitution capabilities of regular expressions, you'd need this near the top of your code:

```
from re import sub.
```

Later in the code, you can remove the

```
str_balance = (row[4].replace('$','').replace(',','')).strip()
```

line completely and replace it with

```
str_balance = (sub(r'[\s\$,]','',row[4])).strip()
```

This line does exactly the same thing as the original line. It removes the dollar sign, commas, and any leading and trailing spaces from the fifth column value.

From CSV to Objects and Dictionaries

You've seen how you can read in data from any CSV file, and how to convert that data from the default string data type to an appropriate Python data type. Chances are, in addition to all of this, you may want to organize the data into a group of objects, all generated from the same class, or perhaps into a set of dictionaries inside a larger dictionary. All the code you've learned far will be useful, because it's all necessary to get the job done. To reduce the code clutter in these examples, we've taken the various bits of code for converting the data and put them into their own functions. This allows you to convert a data item just using the function name with the value to convert in parentheses, like this: `balance(row[4])`.

Importing CSV to Python objects

If you want the data from your CSV file to be organized into a list of objects, write your code as shown here:

```
import datetime as dt
import csv
# Use these functions to convert any string to appropriate Python data type.
# Get just the first name from full name.
def fname(any):
    try:
        nm = any.split(',')
        return nm[1]
    except IndexError:
        return ''
# Get just the last name from full name.
def lname(any):
    try:
        nm = any.split(',')
        return nm[0]
    except IndexError:
        return ''
# Convert string to integer or zero if no value.
def integer(any):
    return int(any or 0)
# Conver mm/dd/yyyy date to date or None if no valid date.
def date(any):
    try:
        return dt.datetime.strptime(any,"%m/%d/%Y").date()
    except ValueError:
        return None
# Convert any string to Boolean, False if no value.
def boolean(any):
    return bool(any)
# Convert string to float, or to zero if no value.
def floatnum(any):
    s_balance = (any.replace('$','').replace(',','')).strip()
    return float(s_balance or 0)
# Create an empty list of people.
people = []
# Define a class where each person is an object.
class Person:
    def __init__(self, id, first_name, last_name, birth_year, date_joined, is_
   active, balance):
        self.id = id
        self.first_name = first_name
        self.last_name = last_name
        self.birth_year = birth_year
```

```
        self.date_joined = date_joined
        self.is_active = is_active
        self.balance = balance

# Open CSV file with UTF-8 encoding, don't read in newline characters.
with open('sample.csv', encoding='utf-8', newline='') as f:
    # Set up a csv reader with a counter.
    reader = enumerate(csv.reader(f))
    # Skip the first row, which is column names.
    f.readline()
    # Loop through remaining rows one at a time, i is counter, row is
   entire row.
    for i, row in reader:
        # From each data row in the CSV file, create a Person object with unique
   id and appropriate data types, add to people list.
        people.append(Person(i, fname(row[0]), lname(row[0]), integer(row[1]),
   date(row[2]), boolean(row[3]), floatnum(row[4])))

# When above loop is done, show all objects in the people list.
for p in people:
    print(p.id, p.first_name, p.last_name, p.birth_year, p.date_joined, p.is_
   active, p.balance)
```

Here's how the code works: The first couple of lines are the required `import`s, followed by a number of functions to convert the incoming string data to Python data types. This code is similar to previous examples in this chapter. We just separated the conversion code out into separate functions to compartmentalize everything a bit:

```
import datetime as dt
import csv
# Use these functions to convert any string to appropriate Python data type.
# Get just the first name from full name.
def fname(any):
    try:
        nm = any.split(',')
        return nm[1]
    except IndexError:
        return ''

# Get just the last name from full name.
def lname(any):
    try:
        nm = any.split(',')
        return nm[0]
    except IndexError:
        return ''
```

```
# Convert string to integer or zero if no value.
def integer(any):
    return int(any or 0)

# Conver mm/dd/yyyy date to date or None if no valid date.
def date(any):
    try:
        return dt.datetime.strptime(any,"%m/%d/%Y").date()
    except ValueError:
        return None

# Convert any string to Boolean, False if no value.
def boolean(any):
    return bool(any)

# Convert string to float, or to zero if no value.
def floatnum(any):
    s_balance = (any.replace('$','').replace(',','')).strip()
    return float(s_balance or 0)
```

This next line creates an empty list named `people`. This just provides a place to store the objects that the program will create from the CSV file:

```
# Create an empty list of people.
people = []
```

Next, the code defines a class that will be used to generate each `Person` object from the CSV file:

```
# Define a class where each person is an object.
class Person:
    def __init__(self, id, first_name, last_name, birth_year, date_joined, is_
   active, balance):
        self.id = id
        self.first_name = first_name
        self.last_name = last_name
        self.birth_year = birth_year
        self.date_joined = date_joined
        self.is_active = is_active
        self.balance = balance
```

The actual reading of the CSV file starts in the next lines. Notice how the code opens the `sample.csv` file with encoding. The `newline=''` just prevents it from sticking the newline character that's at the end of each row to the last item of data in each row. The `reader` uses an enumerator to keep a count while reading the

rows. The `f.readline()` reads the first row, which is just column heads, so that the `for` that follows starts on the second row. The `i` variable in the `for` loop is just the incrementing counter, and the `row` is the entire row of data from the CSV file:

```
# Open CSV file with UTF-8 encoding, don't read in newline characters.
with open('sample.csv', encoding='utf-8', newline='') as f:
    # Set up a csv reader with a counter.
    reader = enumerate(csv.reader(f))
    # Skip the first row, which is column names.
    f.readline()
    # Loop through remaining rows one at a time, i is counter, row is
   entire row.
    for i, row in reader:
```

With each pass through the loop, this line creates a single `Person` object from the incrementing counter (`i`) and appends the data in the `row`. Notice how we've called upon the functions defined earlier in the code to do the data type conversions. This makes this code more compact and a little easier to read and work with:

```
# From each data row in the CSV file, create a Person object with unique id
    and appropriate data types, add to people list.
people.append(Person(i, fname(row[0]), lname(row[0]), integer(row[1]),
    date(row[2]), boolean(row[3]), floatnum(row[4])))
```

When the loop is complete, the next code simply displays each object on the screen to verify that the code worked correctly:

```
# When above loop is done, show all objects in the people list.
for p in people:
    print(p.id, p.first_name, p.last_name, p.birth_year, p.date_joined,
    p.is_active, p.balance)
```

Figure 1-12 shows the output from running this program. Of course, subsequent code in the program can do anything you need to do with each object; the printing is just there to test and verify that it worked.

Importing CSV to Python dictionaries

If you prefer to store each row of data from the CSV file in its own dictionary, you can use code that's similar to the preceding code for creating objects. You don't need the class definition code, because you won't be creating objects here. Instead of creating a people list, you can create an empty `people` dictionary to hold all the individual "person" dictionaries, like this:

```
# Create an empty dictionary of people.
people = {}
```

```
34  # Create an empty list of people.
35  people = []
36  # Define a class where each person is an object.
37  class Person:
38      def __init__(self, id, first_name, last_name, birth_year, date_joined, is_active, balance):
39          self.id = id
40          self.first_name = first_name
41          self.last_name = last_name
42          self.birth_year = birth_year
43          self.date_joined = date_joined
44          self.is_active = is_active
45          self.balance = balance
46
47  # Open CSV file with UTF-8 encoding, don't read in newline characters.
48  with open('sample.csv', encoding='utf-8', newline='') as f:
49      # Set up a csv reader with a counter.
50      reader = enumerate(csv.reader(f))
51      # Skip the first row, which is column names.
52      f.readline()
53      # Loop through remaining rows one at a time, i is counter, row is entire row.
54      for i, row in reader:
55          # From each data row in the CSV file, create a Person object with unique id and appropriate data types, add to people list.
56          people.append(Person(i, fname(row[0]), lname(row[0]), integer(row[1]), date(row[2]), boolean(row[3]), floatnum(row[4])))
57
58  # When above loop is done, show all objects in the people list.
59  for p in people:
60      print(p.id, p.first_name, p.last_name, p.birth_year, p.date_joined, p.is_active, p.balance)
61

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

0  Annie Angst 1982 2011-01-11 True 300.0
1  Barry Bónañas 1973 2012-02-11 True -123.45
2  Sandy Schadenfreude 2004 2003-03-03 True 0.0
3  Wanda Weltschmerz 1995 1994-04-24 True 999999.99
4  Mindy Malaise 2006 2005-05-05 True 454.01
5  Ollie O'Possum 1987 1997-07-27 True -1000.0
6   0 None False 0.0
7  Pamela Pusillanimity 1979 2008-08-08 True 12345.67
```

FIGURE 1-12: Reading a CSV file into a list of objects.

As far as the loop goes, again you can use an enumerator (`i`) to count rows, and you can also use this unique value as the key for each new dictionary you create. The line that starts with `newdict=` creates a dictionary with the data from one CSV file row, using the built-in Python `dict()` function. The next line assigns the value of `i` plus one (to start the first one at one rather than zero) to each newly created dictionary:

```
    # Loop through remaining rows one at a time, i is counter, row is entire row.
    for i, row in reader:
        # From each data row in the CSV file, create a Person object with unique
id and appropriate data types, add to people list.
        newdict = dict({'first_name': fname(row[0]), 'last_name': lname(row[0]),
'birth_year': integer(row[1]), \
                    'date_joined' : date(row[2]), 'is_active' : boolean(row[3]),
'balance' : floatnum(row[4])})
        people[i + 1] = newdict
```

To verify that the code ran correctly, you can loop through the dictionaries in the `people` dictionary and show the *`key`*:*`value`* pair for each item of data in each row. Figure 1-13 shows the result of running that code in VS Code:

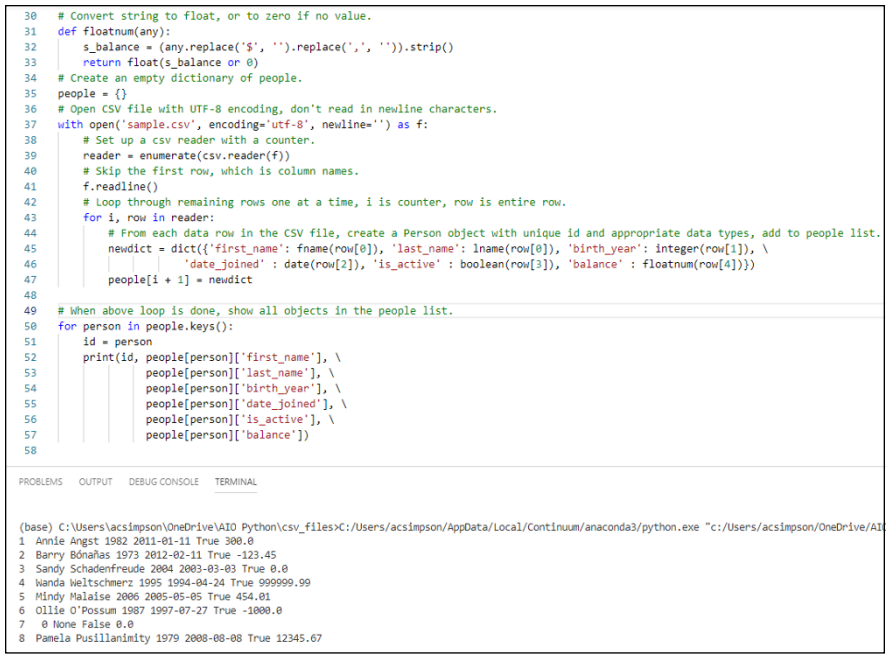


FIGURE 1-13: Reading a CSV file into a dictionary of dictionaries.

Here is all the code that reads the data from the CSV files into the dictionaries:

```
import datetime as dt
import csv
# Use these functions to convert any string to appropriate Python data type.
# Get just the first name from full name.
def fname(any):
    try:
        nm = any.split(',')
        return nm[1]
    except IndexError:
        return ''
# Get just the last name from full name.
def lname(any):
    try:
        nm = any.split(',')
        return nm[0]
    except IndexError:
        return ''
# Convert string to integer or zero if no value.
def integer(any):
    return int(any or 0)
# Convert mm/dd/yyyy date to date or None if no valid date.
def date(any):
    try:
        return dt.datetime.strptime(any, "%m/%d/%Y").date()
```

```
    except ValueError:
        return None
# Convert any string to Boolean, False if no value.
def boolean(any):
    return bool(any)
# Convert string to float, or to zero if no value.
def floatnum(any):
    s_balance = (any.replace('$', '').replace(',', '')).strip()
    return float(s_balance or 0)
# Create an empty dictionary of people.
people = {}
# Open CSV file with UTF-8 encoding, don't read in newline characters.
with open('sample.csv', encoding='utf-8', newline='') as f:
    # Set up a csv reader with a counter.
    reader = enumerate(csv.reader(f))
    # Skip the first row, which is column names.
    f.readline()
    # Loop through remaining rows one at a time, i is counter, row is
   entire row.
    for i, row in reader:
        # From each data row in the CSV file, create a Person object with
   unique id
        # and appropriate data types, add to people dictionary.
        newdict = dict({'first_name': fname(row[0]), 'last_name': lname(row[0]),
   'birth_year': integer(row[1]), \
                    'date_joined' : date(row[2]), 'is_active' : boolean(row[3]),
   'balance' : floatnum(row[4])})
        people[i + 1] = newdict

# When above loop is done, show all objects in the people list.
for person in people.keys():
    id = person
    print(id, people[person]['first_name'], \
              people[person]['last_name'], \
              people[person]['birth_year'], \
              people[person]['date_joined'], \
              people[person]['is_active'], \
              people[person]['balance'])
```

CSV files are widely used because it's easy to export data from spreadsheets and database tables to this format. Getting data from those files can be tricky at times, but you'll find Python's `csv` module a big help. It takes care of many of the details, makes it relatively easy to loop through one row at a time, and handles the data however you see fit within your Python app.

Similar to CSV for transporting and storing data in a simple textual format is JSON, which stands for JavaScript Object Notation. You learn all about JSON in the next chapter.

IN THIS CHAPTER

- Organizing JSON data
- Understanding serialization
- Loading data from JSON files
- Dumping Python data to JSON

Chapter 2
Juggling JSON Data

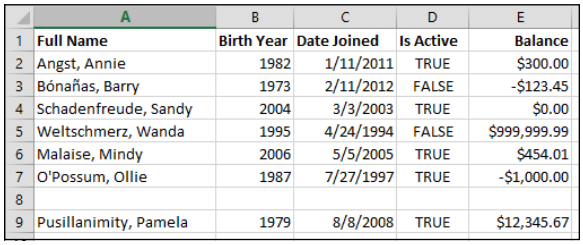
JSON (JavaScript Object Notation) is a common marshalling format for object-oriented data. That term, *marshalling format*, generally means a format used to send the data from one computer to another. However, some databases, such as the free Realtime Database at Google's Firebase, actually store the data in JSON format as well. The name *JavaScript* at the front sometimes throws people off a bit, especially when you're using Python, not JavaScript, to write your code. But don't worry about that. The format just got its start in the JavaScript world. It's now a widely known general purpose format used with all kinds of computers and programming languages.

In this chapter you learn exactly what JSON is, as well as how to export and import data to and from JSON. If you find that all the buzzwords surrounding JSON make you uncomfortable, don't worry. We'll get through all the jargon first. As you'll see, JSON data is formatted almost the same way as Python data dictionaries. So there won't be a huge amount of new stuff to learn. Also, you already have the free Python JSON module, which makes it even easier to work with JSON data.

Organizing JSON Data

JSON data is roughly the equivalent of a data dictionary in Python, which makes JSON files fairly easy to work with. It's probably easiest to understand when it's compared to tabular data. For instance, Figure 2-1 shows some tabular data in

an Excel worksheet. Figure 2-2 shows the same data converted to JSON format. Each row of data in the Excel sheet has simply been converted to a dictionary of *key*:*value* pairs in the JSON file. And there are, of course, lots of curly braces to indicate that it's dictionary data.



	A	B	C	D	E
1	**Full Name**	**Birth Year**	**Date Joined**	**Is Active**	**Balance**
2	Angst, Annie	1982	1/11/2011	TRUE	$300.00
3	Bónañas, Barry	1973	2/11/2012	FALSE	-$123.45
4	Schadenfreude, Sandy	2004	3/3/2003	TRUE	$0.00
5	Weltschmerz, Wanda	1995	4/24/1994	FALSE	$999,999.99
6	Malaise, Mindy	2006	5/5/2005	TRUE	$454.01
7	O'Possum, Ollie	1987	7/27/1997	TRUE	-$1,000.00
8					
9	Pusillanimity, Pamela	1979	8/8/2008	TRUE	$12,345.67

FIGURE 2-1: Some data in an Excel spreadsheet.

FIGURE 2-2: Excel spreadsheet data converted to JSON format.

That's one way to do a JSON file. You can also do a *keyed JSON file* where each chunk of data has a single key the uniquely identifies it (no other dictionary in the same file can have the same key). The key can be a number or some text; it doesn't really matter which, so long as it's unique to each item. When you're downloading JSON files created by someone else, it's not unusual for the file to be keyed. For example, on Alan's personal website he uses a free Google Firebase Realtime Database to count hits per page and other information about each page. This Realtime Database stores the data as shown in Figure 2-3. Those weird things that look like *-LAOqOxg6kmP4jhnjQXS* are all keys that the Firebase generates automatically for each item of data, to guarantee uniqueness. The + sign next to each key allows you to expand and collapse the information under each key.

CONVERTING EXCEL TO JSON

In case you're wondering, to convert that sample Excel spreadsheet to JSON, set your browser to `www.convertcsv.com/csv-to-json.htm` and follow these steps:

1. In Step 1, open the Choose File tab, set the Encoding to UTF-8, click the Browse button, select your Excel file, and click Open.
2. In Step 2, make sure the First Row Is Column Names option is checked and set Skip # of Lines to 1 to skip the column headings row.
3. In Step 5, click the CSV to JSON button.
4. Next to Save Your Result, type a filename then click the Download Result button.

The file should end up in your Downloads folder (or to whatever location you normally download) with a .json extension. It's a plain text file so you can open it with any text editor, or a code editor like VS Code. The converter automatically skips empty rows in Excel files, so your JSON file won't contain any data for empty rows in a spreadsheet. If you often work with Excel, CSV, JSON, and similar types of data, you may want to spend some time exploring the many tools and capabilities of that `http://www.convertcsv.com/` website.

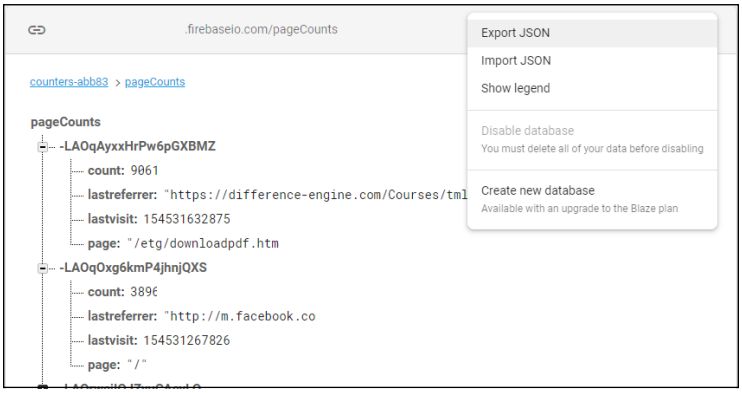


FIGURE 2-3: Some data in a Google Firebase Realtime Database.

As you can see in the image, Firebase also has an Export JSON option that downloads the data to a JSON file to your computer. We did this. Figure 2-4 shows how the data looks in that downloaded file. You can tell that one is a keyed JSON file because each chunk of data is preceded by a unique key, like *-LAOqAyxxHrPw6pGXBMZ* followed by a colon. You can work with both keyed and un-keyed JSON files in Python.

FIGURE 2-4: Google Firebase Realtime Database data exported to keyed JSON file.

Some readers may have noticed that the `Date Joined` field in the JSON file doesn't look like a normal *mm/dd/yyyy* date. The `lastvisit` field from the Firebase database is a `datetime`, even though it doesn't look like a date or time. But don't worry about that. You'll learn how to convert those odd-looking serial dates (as they're called) to human readable format later in this chapter.

Understanding Serialization

When it comes to JSON, the first buzzword you have to learn is *serialization*. Serialization is the process of converting an object (like a Python dictionary) into a stream of bytes (characters) that can be sent across a wire, stored in a file or database, or stored in memory. The main purpose is to save all the information contained within an object in a way that can easily be retrieved on any other computer. The process of converting it back to an object is called *deserialization*. To keep things simple you may just consider using these definitions:

- **Serialize:** Convert an object to a string.
- **Deserialize:** Convert a string to an object.

The Python standard library includes a `json` module that helps you work with JSON files. Because it's part of the standard library, you just have to put `import json` near the top of your code to access its capabilities. The four main methods for serializing and deserializing `json` are summarized in Table 2-1.

TABLE 2-1 Python JSON Methods for Serializing and Deserializing JSON Data

Method	Purpose
`json.dump()`	Write (serialize) Python data to a JSON file (or stream).
`json.dumps()`	Write (serialize) a Python object to a JSON string.
`json.load()`	Load (deserialize) JSON from a file or similar object.
`json.loads()`	Load (deserialize) JSON data from a string.

Data types in JSON are somewhat similar to data types in Python, but they're not exactly the same. Table 2-2 lists how data types are converted between the two languages when serializing and deserializing.

TABLE 2-2 Python and JSON Data Conversions

Python	JSON
dict	object
list, tuple	array
str	string
int and float	number
True	true
False	false
None	null

Loading Data from JSON Files

To load data form JSON files, make sure you import `json` near the top of the code. Then you can use a regular file `open()` method to open the file. As with other kinds of files, you can add `encoding = "utf-8"` if you think there are any foreign

characters in the data to preserve. You can also use `newline=""` to avoid bringing in the newline character at the end of each row, which isn't really part of the data. It's just a hidden character to end the line when displaying the data on the screen.

To load the JSON data into Python, come up with a variable name to hold the data (we'll use `people`) and then use `json.load()` to load the file contents into the variable, like this:

```
import json
# This is the Excel data (no keys)
filename = 'people_from_excel.json'
# Open the file (standard file open stuff)
with open(filename, 'r', encoding='utf-8', newline='') as f:
    # Load the whole json file into an object named people
    people = json.load(f)
```

Running this code doesn't display anything on the screen. However, you can explore the `people` object in a number of ways using `print()` un-indented print statements below this last line. For example this

```
    print(people)
```

. . . displays everything that's in the `p` variable. In the output, you can see it starts and ends with square brackets (`[]`), which tells you that `people` is a list. To verify this, you can run this line of code:

```
print(type(people))
```

When you do, Python displays `<class 'list'>`, which tells you that the object is an instance of the `list` class. In other words, it's a `list` object, although most people would just call it a list.

```
<class 'list'>
```

Because it's a list, you can loop through it. Within the loop you can display the type of each item, like this:

```
for p in people:
    print(type(p))
```

The output from this is:

```
<class 'dict'>
<class 'dict'>
<class 'dict'>
```

```
<class 'dict'>
<class 'dict'>
<class 'dict'>
<class 'dict'>
```

This's useful information because it tells you that each of the "people" (which we've abbreviated `p` in that code) in the list is a Python dictionary. So within the loop, you can isolate each value by its key. For example, take a look at this code:

```
for p in people:
    print(p['Full Name'], p['Birth Year'], p['Date Joined'], p['Is Active'],
              p['Balance'])
```

Running this code displays all the data in the JSON file, as in the following. Those data came from the Excel spreadsheet shown back in Figure 2-1.

```
Angst, Annie 1982 40554 True 300
Bónañas, Barry 1973 40950 False -123.45
Schadenfreude, Sandy 2004 37683 True 0
Weltschmerz, Wanda 1995 34448 False 999999.99
Malaise, Mindy 2006 38477 True 454.01
O'Possum, Ollie 1987 35638 True -1000
Pusillanimity, Pamela 1979 39668 True 12345.67
```

Converting an Excel date to a JSON date

You may be thinking "Hey, waitaminit . . . what's with those 40554, 40950, 37683 numbers in the Date Joined column?" Well, those are serial dates, but you can certainly convert them to Python dates. You'll need to import the xlrd (Excel reader) and `datetime` modules. Then, to convert that integer in the `p['Date Joined']` column to a Python date, use this code:

```
y, m, d, h, i, s = xlrd.xldate_as_tuple(p['Date Joined'],0)
joined = dt.date(y, m, d)
```

To display this date in a familiar format, you can use an f-string like this:

```
print(f"{joined:%m/%d/%Y}")
```

Here is all the code, including the necessary imports at the top of the file:

```
import json, xlrd
import datetime as dt
# This is the Excel data (no keys)
filename = 'people_from_excel.json'
```

```
# Open the file (standard file open stuff)
with open(filename, 'r', encoding='utf-8', newline='') as f:
    # Load the whole json file into an object named people
    people = json.load(f)

# Dictionaries are in a list, loop through and display each dictionary.
for p in people:
    name=p['Full Name']
    byear = p['Birth Year']
    # Excel date pretty tricky, use xlrd module.
    y, m, d, h, i, s = xlrd.xldate_as_tuple(p['Date Joined'],0)
    joined = dt.date(y, m, d)
    balance = '$' + f"{p['Balance']:,.2f}"
    print(f"{name:<22} {byear}  {joined:%m/%d/%Y} {balance:>12}")
```

Here is the output of this code, which, you can see, is fairly neatly formatted and looks more like the original Excel data than the JSON data. If you need to display the data in *dd/mm/yyyy* format just changes the pattern in the last line to `%d/%m/%Y`.

```
Angst, Annie           1982  01/11/2011      $300.00
Bónañas, Barry         1973  02/11/2012     $-123.45
Schadenfreude, Sandy   2004  03/03/2003        $0.00
Weltschmerz, Wanda     1995  04/24/1994  $999,999.99
Malaise, Mindy         2006  05/05/2005      $454.01
O'Possum, Ollie        1987  07/27/1997   $-1,000.00
Pusillanimity, Pamela  1979  08/08/2008   $12,345.67
```

Looping through a keyed JSON file

Opening and loading a keyed JSON file is the same as opening a non-keyed file. However, after it's loaded, the data tends to be a single dictionary rather than a list of dictionaries. For example, here is the code to open and load the data we exported from Firebase (the original is shown back in Figure 3-4). This data contains hit counts for pages in a website, including the page name, the number of hits to date, the last referred (the last page that sent someone to that page), and the date and time of the last visit. As you can see, the code for opening and loading the JSON data is basically the same. The JSON data loads to an object we named `hits`:

```
import json
import datetime as dt
# This is the Firebase JSON data (keyed).
filename = 'firebase_hitcounts.json'
```

```
# Open the file (standard file open stuff).
with open(filename, 'r', encoding='utf-8', newline='') as f:
    # Load the whole json file into an object named people
    hits = json.load(f)

print(type(hits))
```

When you run this code, the last line displays the data type of the `hits` object, into which the JSON data was loaded, as `<class 'dict'>`. This tells you that the `hits` object is one large dictionary rather than a list of individual dictionaries. You can loop through this dictionary using a simple loop like we did for the non-keyed JSON file, like this:

```
for p in hits:
    print(p)
```

The result of this, however, is that you don't see much data. In fact, all you see is the key for each sub-dictionary contained within the larger `hits` dictionary:

```
-LAOqAyxxHrPw6pGXBMZ
-LAOqOxg6kmP4jhnjQXS
-LAOrwciIQJZvuCAcyLO
-LAOs2nsVVxbjAwxUXxE
-LAOwqJsjfuoQx8WIS1X
-LAQ7ShbQPqOANbDmm3O
-LAQrS6avlv0PuJGNm6P
-LI0iPwZ7nu3IUgiQORH
-LI2DFNAxVnT-cxYzWR-
```

This is not an error or a problem. It's just how it works with nested dictionaries. But don't worry, it's pretty easy to get to the data inside each dictionary. You can, for instance, use two looping variables, which we'll call `k` (for *key*) and `v` (for *value*), to loop through `hits.items()`, like this:

```
for k, v in hits.items():
    print(k,v)
```

This gives you a different view where you see each key followed by the dictionary for that key enclosed in curly braces (the curly braces tell you that the data inside is in a dictionary). Figure 2-5 shows the output from this.

The values for each sub-dictionary are in the `v` object of this loop. If you want to access individual items of data, use `v` followed by a pair of square brackets with the key name (for the field) inside. For example, `v['count']` contains whatever

is stored as the `count:` in a given row. Take a look at this code in which we don't even bother with displaying the key:

```
for k, v in hits.items():
    # Store items in variables.
    key = k
    hits = v['count']
    last_visit=v['lastvisit']
    page = v['page']
    came_from=v['lastreferrer']
    print(f"{hits} {last_visit} {page:<28} {came_from}")
```

FIGURE 2-5: Output from looping through and displaying keys and values from sub-dictionaries.

The output from this is the data from each dictionary, formatted in a way that's a little easier to read, as shown in Figure 2-6.

```
9061 1545316328750 /etg/downloadpdf.html        https://difference-engine.com/Courses/tml-5-1118/
3896 1545312678263 /                            http://m.facebook.com
3342 1545311601815 /html_css/index.html         https://alansimpson.me/
2220 1545280814480 /html_css/codequickies/      http://alansimpson.me/html_css/codequickies/dropdownmenu.html
2194 1545308609977 /index.html                  https://alansimpson.me/firebase/hitcounter/
1154 1544974827730 /javascript/code_quickies/   https://alansimpson.me/javascript/code_quickies/clickdropdown/
1547 1545305365597 /how/
1439 1545315903301 /datascience/beginner/       https://www.youtube.com/
1643 1545226502089 /datascience/cheatsheets/
```

FIGURE 2-6: Output from showing one value at a time from each dictionary.

You may notice we've run into another weird situation with the `lastvisit` column. The date appears in the format *545316328750* rather than the more familiar *mm/dd/yyyy* format. This time we can't blame Excel because these dates were never in Excel. What you're seeing here is the Firebase timestamp of when the data item was last written to the database. This date is expressed as a UTC date, including the time down to the nanosecond. This's why the number is so long. Obviously, if you need people to be able to understand these dates, you need to translate them to Python dates, as we discuss next.

Converting firebase timestamps to Python dates

As always, the first thing you need to do when working with dates and times in a Python app is to make sure you've imported the `datetime` module, which we usually do using the code `import datetime as dt`, in which the `dt` is an optional alias (a nickname that easier to type than the full name).

Because we know that the Firebase datetime is UTC-based, we know that we can use the `datetime .utcfromtimestamp()` method to convert it to Python time. But there is a catch. If you went strictly by the documentation you would expect this to work:

```
last_visit = dt.datetime.utcfromtimestamp(v['lastvisit'])
```

However, in Windows, apparently that nanosecond resolution is a bit much and this code raises an OS Error exception. Fortunately, there's an easy workaround. Dividing that `lastvisit` number by 1,000 trims off the last few digits, which gets the number into a lower-resolution `datetime` that Windows can stomach. All we really care about in this application is the date of the last visit; we don't care at all about the time. So you can grab just the date and get past the error by writing the code like this:

```
    last_visit = dt.datetime.utcfromtimestamp(v['lastvisit']/1000).date()
```

What you end up with, then, in the `last_visit` variable is a simple Python date. So you can use a standard f-string to format the date however you like. For example, use this in your f-string to display that date:

```
{last_visit: %m/%d/%Y}
```

The dates will be in *mm/dd/yyyy* format in the output, like this:

```
12/20/2018
12/19/2018
12/17/2018
12/20/2018
11/30/2018
12/16/2018
12/20/2018
12/20/2018
12/19/2018
```

Loading unkeyed JSON from a Python string

The `load()` method we used in the previous examples loaded the JSON data from a file. However, JSON data is always delivered in a text file, thus you can copy/paste the whole thing into a Python string. Typically you give the whole string a variable name and set it equal to some docstring that starts and ends with triple quotation marks. Put all the JSON data inside the triple quotation marks as in the following code. (To keep the code short, we've included data for only a couple of people, but at least you can see how the data is structured.)

```
import json
# Here the JSON data is in a big string named json_string.
# It starts and the first triple quotation marks and extends
# down to the last triple quotation marks.
json_string = """
{
"people": [
    {
    "Full Name": "Angst, Annie",
    "Birth Year": 1982,
    "Date Joined": "01/11/2011",
    "Is Active": true,
    "Balance": 300
  },
 {
    "Full Name": "Schadenfreude, Sandy",
    "Birth Year": 2004,
    "Date Joined": "03/03/2003",
    "Is Active": true,
    "Balance": 0
  }
]
}
"""
```

Although it may be nice to be able to see all the data from within your code like that, there is one big disadvantage: You can't loop through a string to get to individual items of data. If you want to loop through, you need to load the JSON data from the string into some kind of object. To do this, use `json.loads()` (where the `s` is short for *from string*), as in the following code. As usual, `peep_data` is just a name we made up to differentiate the loaded JSON data from the data in the string:

```
# Load JSON data from the big json_string string.
peep_data = json.loads(json_string)
```

Now that you have an object with which to work (`peep_data`), you can loop through and work with the code one bit at a time, like this:

```
# Now you can loop through the peep_data collection.
for p in peep_data['people']:
    print(p["Full Name"], p["Birth Year"], p["Date Joined"],p['Is
   Active'],p['Balance'])
```

Figure 2-7 shows all the code and the result of running that code in VS Code.

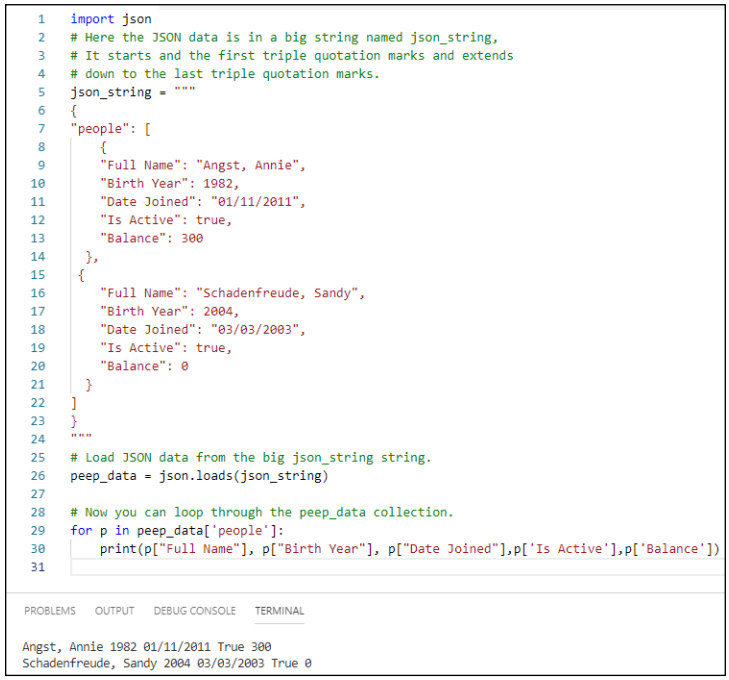


FIGURE 2-7: Output from showing one value at a time from each dictionary (see bottom of image).

Loading keyed JSON from a Python string

Keyed data can also be stored in a Python string. In the following example, we used `json_string` as the variable name again, but as you can see, the data inside the string is structured a little differently. The first item has a key of 1 and the second item has a key of 2. But again, the code uses `json.loads(json_string)` to load this date from the string into a JSON object:

```
import json
# Here the JSON data is in a big string named json_string,
# It starts and the first triple quotation marks and extends
```

```
# down to the last triple quotation marks.
json_string = """
  {
  "1": {
    "count": 9061,
    "lastreferrer": "https://difference-engine.com/Courses/tml-5-1118/",
    "lastvisit": "12/20/2018",
    "page": "/etg/downloadpdf.html"
  },
  "2": {
    "count" : 3342,
    "lastreferrer" : "https://alansimpson.me/",
    "lastvisit" : "12/19/2018",
    "page" : "/html_css/index.html"
  }
  }
"""
# Load JSON data from the big json_string string.
hits_data = json.loads(json_string)

# Now you can loop through the hits_data collection.
for k, v in hits_data.items():
    print(f"{k}. {v['count']:>5} - {v['page']}")
```

The loop at the end prints the key, hit count, and page name from each item in the format shown in the following code. Note that this loop uses the two variables named `k` and `v` to loop through `hits_data.items()`, which is the standard syntax for looping through a dictionary of dictionaries:

```
1.  9061 - /etg/downloadpdf.html
2.  3342 - /html_css/index.html
```

Changing JSON data

When you have JSON data in a data dictionary, you can use standard dictionary procedures (originally presented in Book 2, Chapter 4) to manipulate the data in the dictionary. As you're looping through the data dictionary with *key*, *value* variables, you can change the value of any *key*:*value* pair using the relatively simple syntax:

```
    value['key'] = newdata
```

The *key* and *value* are just the k and v variables from the loop. For example, suppose you're looping through a dictionary created from the Firebase database, which includes a lastvisit field shown as a UTC Timestamp number. You want to change this timestamp to a string in a more familiar Python format. Set up a loop as in the following code, in which the first line inside the loop creates a new variable named pydate that contains the date as a Python date. Then the second line replaces the content of v['lastvisit'] with this date in *mm/dd/yy* format:

```
for k, v in hits.items():
    # Convert the Firebase date to a Python date.
    pydate = dt.datetime.utcfromtimestamp(v['lastvisit']/1000).date()
    # In the dictionary, replace the Firebase date with string of Python date.
    v['lastvisit']= f"{pydate:%m/%d/%Y})"
```

When this loop is complete, all the values of the "lastvisit" column will be dates in *mm/dd/yyyy* format rather than the Firebase timestamp format.

Removing data from a dictionary

To remove data from a dictionary as you're going through the loop, use the syntax pop(*'keyname'*, None). Replace *'keyname'* with the name of the column you want to remove. For example, to remove all the lastreferrer key names and data from a dictionary created by the Firebase database JSON example, add v.pop('lastreferrer', None) to the loop.

Figure 2-8 shows an example where lines 1-8 import Firebase data into a Python object named hits. The line 10 starts a loop that goes through each key (k) and value (v) in the dictionary. Line 12 converts the timestamp to a Python date named pydate. Then line 16 replaces the timestring that was in the lastvisit column with that Python date as a string in *mm/dd/yyyy* format. Line 16, v.pop('lastreferrer', None), removes the whole lastreferrer *key:value* pair from each dictionary. The final loop shows what's in the dictionary after making those changes.

Keep in mind that changes you make to the dictionary in Python have no effect on the file or string from which you loaded the JSON data. If you want to create a new JSON string or file, use the json.dumps() or json.dump() methods discussed next.

```
1   import json
2   import datetime as dt
3   # This is the Firebase JSON data (keyed).
4   filename = 'firebase_hitcounts.json'
5   # Open the file (standard file open stuff).
6   with open(filename, 'r', encoding='utf-8', newline='') as f:
7       # Load the whole json file into an object named hits
8       hits = json.load(f)
9
10  for k, v in hits.items():
11      # Convert the Firebase date to a Python date.
12      pydate = dt.datetime.utcfromtimestamp(v['lastvisit']/1000).date()
13      # In the dictionary, replace the Firebase date with string of Python date.
14      v['lastvisit']= f"{pydate:%m/%d/%Y})"
15      # Remove the entire last referrer column.
16      v.pop('lastreferrer', None)
17
18  # Now look at the lastvisit date in the hits dictionary.
19  for k, v in hits.items():
20      print(k,v)
21
22
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

-LAOqAyxxHrPw6pGXBMZ {'count': 9061, 'lastvisit': '12/20/2018)', 'page': '/etg/downloadpdf.html'}
-LAOqOxg6kmP4jhnjQXS {'count': 3896, 'lastvisit': '12/20/2018)', 'page': '/'}
-LAOrwciIQJZvuCAcyLO {'count': 3342, 'lastvisit': '12/20/2018)', 'page': '/html_css/index.html'}
-LAOs2nsVVxbjAwxUXxE {'count': 2220, 'lastvisit': '12/20/2018)', 'page': '/html_css/codequickies/'}
-LAOwqJsjfuoQx8WISlX {'count': 2194, 'lastvisit': '12/20/2018)', 'page': '/index.html'}
-LAQ7ShbQPqOANbDmm3O {'count': 1154, 'lastvisit': '12/16/2018)', 'page': '/javascript/code_quickies/'}
-LAQrSGavlv0PuJGNm6P {'count': 1547, 'lastvisit': '12/20/2018)', 'page': '/how/'}
-LI0iPwZ7nu3IUgiQORH {'count': 1439, 'lastvisit': '12/20/2018)', 'page': '/datascience/beginner/'}
-LI2DFNAxVnT-cxYzWR- {'count': 1643, 'lastvisit': '12/19/2018)', 'page': '/datascience/cheatsheets/'}
```

FIGURE 2-8: Changing the value of one key in each dictionary, and removing an entire *key:value* pair from the dictionary.

Dumping Python Data to JSON

So far we've talked about bringing JSON data from the outside world into your app so Python can use its data. There may be times where you want to go the opposite direction, to take some data that's already in your app in a dictionary format and export it out to JSON to pass to another app, the public at large, or whatever. This is where the `json.dump()` and `json.dumps()` methods come into play. The `dumps()` method creates a JSON string of the data, which is still in memory where you can `print()` it to see it. For example, the previous code examples imported a Firebase database to a Python dictionary, then looped through that dictionary changing all the timestamps to *mm/dd/yyyy* dates, and also removing all the `lastreferrer` *key*:*value* pairs. So let's say you want to create a JSON string of this new dictionary. You could use dumps like this to create a string named `new_dict`, and you could also print that string to the console. The last two lines of code outside the loop would be:

```
#Looping is done, copy new dictionary to JSON string.
new_dict = json.dumps(hits)
print(new_dict)
```

The `new_dict` string would show in its native, not-very-readable format, which would look something like this:

```
{"-LAOqAyxxHrPw6pGXBMZ": {"count": 9061, "lastvisit": "12/20/2018)", "page":
                "/etg/downloadpdf.html"}, "-LAOqOxg6kmP4jhnjQXS": {"count": 3896,
                "lastvisit": "12/20/2018)", "page": "/"}, "-LAOrwciIQJZvuCAcyLO":
                {"count": 3342, "lastvisit": "12/20/2018)", "page":
                "/html_css/index.html"}, ... }}
```

Wee replaced some of the data with `...` because you don't need to see all the items to see how unreadable it looks.

Fortunately, the `.dumps()` method supports an `indent=` option in which you can specify how you want to indent the JSON data to make it more readable. Two spaces is usually sufficient. For example, add `indent=2` to the code above as follows:

```
#Looping is done, copy new dictionary to JSON string.
new_dict = json.dumps(hits, indent=2)
print(new_dict)
```

The output from this `print()` shows the JSON data in a much more readable format, as shown here:

```
{
  "-LAOqAyxxHrPw6pGXBMZ": {
    "count": 9061,
    "lastvisit": "12/20/2018)",
    "page": "/etg/downloadpdf.html"
  },
  "-LAOqOxg6kmP4jhnjQXS": {
    "count": 3896,
    "lastvisit": "12/20/2018)",
    "page": "/"
  },
 ...
}
```

If you use foreign or special characters in your data dictionary and you want to preserve them, add `ensure_ascii=False` to your code as follows:

```
new_dict = json.dumps(hits, indent=2, ensure_ascii=False)
```

In our example, the key names in each dictionary are already in alphabetical order (`count`, `lastvisit`, `page`), so we wouldn't need to do anything to put them that way. But in your own code, if you want to ensure the keys in each dictionary are in alphabetical order, add `sortkeys=True` to your `.dumps` method as follows:

```
new_dict = json.dumps(hits, indent=2, ensure_ascii=False, sort_keys=True)
```

If you want to output your JSON to a file, use `json.dump()` rather than `json.dumps()`. You can use `ensure_ascii=False` to maintain foreign characters, and `sort_keys = True` to alphabetize key names. You can also include an `indent=` option, although that would make the file larger and typically you want to keep files small to conserve space and minimize download time.

As an example, suppose you want to create a file named `hitcounts_new.json` (or if it already exists, open it to overwrite its content). You want to retain any foreign characters that you write to the file. Here's the code for that; the `'w'` is required to make sure the file opens for writing data into it:

```
with open('hitcounts_new.json', 'w', encoding='utf-8') as out_file:
```

Then, to copy the dictionary named `hits` as JSON into this file, use the name you assigned at the end of the code in the line above. Again, to retain any foreign characters and perhaps to alphabetize the key names in each dictionary, follow that line with this one, making sure this one is indented to be contained within the `with` block:

```
    json.dump(hits, out_file, ensure_ascii=False, sort_keys=True)
```

Figure 2-9 shows all the code starting with the data that was exported from Firebase, looping through the dictionary that the import created, changing and removing some content, and then writing the new dictionary out to a new JSON file named `hitcounts_new.json`.

```
1  import json
2  import datetime as dt
3  # This is the Firebase JSON data (keyed).
4  filename = 'firebase_hitcounts.json'
5  # Open the file (standard file open stuff).
6  with open(filename, 'r', encoding='utf-8', newline='') as f:
7      # Load the whole json file into an object named hits.
8      hits = json.load(f)
9
10 # Loop through the new hits data dictionary.
11 for k, v in hits.items():
12     # Convert the Firebase date to a Python date.
13     pydate = dt.datetime.utcfromtimestamp(v['lastvisit']/1000).date()
14     # In the dictionary, replace the Firebase date with string of Python date.
15     v['lastvisit'] = f"{pydate:%m/%d/%Y})"
16     # Remove the entire last referrer column.
17     v.pop('lastreferrer', None)
18
19 # Write the modifed data to a JSON file named hitcounts_new.json.
20 with open('hitcounts_new.json', 'w', encoding='utf-8') as out_file:
21     json.dump(hits, out_file, ensure_ascii=False, sort_keys=True)
22
23 print('Done')
```

FIGURE 2-9: Writing modified Firebase data to a new JSON file named `hitcounts_new.json`.

Figure 2-10 shows the contents of the `hitcounts_new.json` file after running the app. We didn't indent the JSON because files are really for storing or sharing, not for looking at, but you can still see the `datevisited` values are in the *mm/dd/yyyy* format and the `lastreferrer` *`key`*`:`*`value`* pair isn't in there, because earlier code removed that *`key`*`:`*`value`* pair.

FIGURE 2-10: Writing modified Firebase data to a new JSON file named `hitcounts_new.json`.

JSON is a very widely used format for storing and sharing data. Luckily Python has lots of built-in tools for consuming and creating JSON data. We've covered the most important capabilities here. But don't be shy about searching Google or YouTube for *python json* if you want to explore more.

IN THIS CHAPTER

- How the Web works
- Opening web pages from Python
- Posting to the Web with Python
- Web scraping with Python

Chapter 3
Interacting with the Internet

As you probably know, the Internet is home to virtually all the world's knowledge. Most of us use the World Wide Web (a.k.a. *the Web*) to find information all the time. We do so using a web browser like Safari, Google Chrome, Firefox, Opera, Internet Explorer, or Edge. To visit a website, you type a URL (uniform resource locator) into your browser's Address bar and press Enter, or you click a link that sends you to the page automatically.

As an alternative to browsing the Web with your web browser, you can access its content *programmatically*. In other words, you can use a programming language like Python to post information to the Web, as well as to access web information. In a sense, you make the Web your personal database of knowledge from which your apps can pluck information at will. In this chapter you learn about the two main modules for access the Web programmatically with Python: urllib and Beautiful Soup.

How the Web Works

When you open up your web browser and type in a URL or click a link, that action sends a *request* to the Internet. The Internet directs your request to the appropriate web server, which in turn sends a *response* back to your computer. Typically that

response is a *web page*, but it can be just about any file. Or it can be an error message if the thing you requested no longer exists at that location. But the important thing is that you the *user* (a human being), and your *user agent* (the program you're using to access the Internet) are on the *client* side of things. The *server*, which is just a computer, not a person, sends back its response, as illustrated in Figure 3-1.

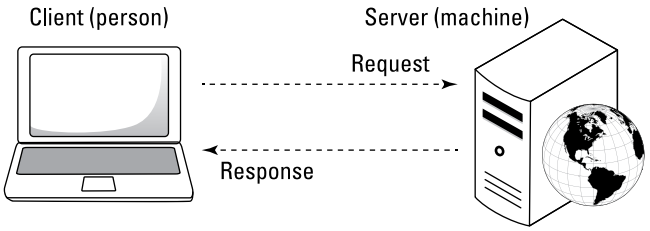


FIGURE 3-1: The client makes a request, and the server sends back a response.

Understanding the mysterious URL

The URL is a key part of the whole transaction, because that's how the Internet finds the resource you're seeking. On the Web, all resources use the Hypertext Transfer Protocol (HTTP), and thus their URLs start with `http://` or `https://`. The difference is that `http://` sends stuff across the wire in its raw form, which makes it susceptible to hackers and others who can "sniff out" the traffic. The `https` protocol is *secure* in that the data is *encrypted*, which means it's been converted to a secret code that's not so easy to read. Typically, any site with whom you do business and to whom you transmit sensitive information like passwords and credit card numbers, uses `https` to keep that information secret and secure.

The URL for any website can be relatively simple, such as Alan's URL of `https://AlanSimpson.me`. Or it can be complex to add more information to the request. Figure 3-2 shows parts of a URL, some of which you may have noticed in the past.

Note that the order matters. For example, it's possible for a URL to contain a path to a specific folder or page (starting with a slash right after the domain name). The URL can also contain a query string, which is always last and always starts with a question mark (?). After the question mark comes one or more *`name=value`* pairs, basically the same syntax you've seen in data dictionaries and JSON. If there are multiple *`name=value`* pairs, they are separated by ampersands.

TECHNICAL STUFF

A # followed by a name after the page name at the end of a URL is called a *fragment*, which indicates a particular place on the target page. Behind the scenes in the code of the page is usually a `<a id="name"></a>` tag that directs the browser to a spot on the page to which it should jump after it opens the page.

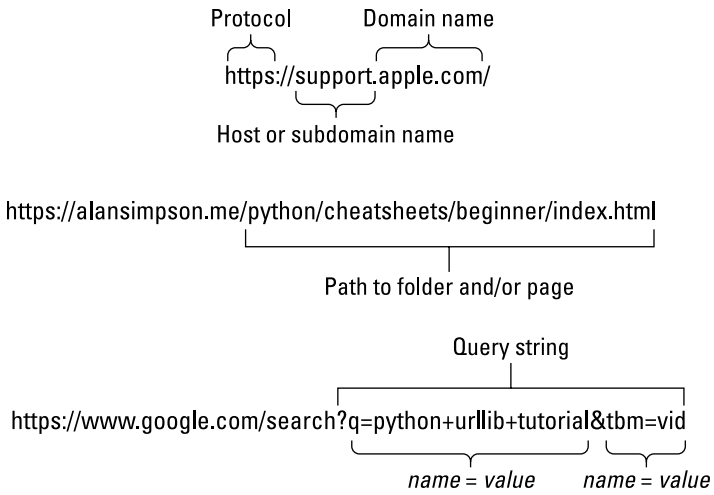


FIGURE 3-2: Different parts of URLs.

Exposing the HTTP headers

When you're using the Web, all you really care about is the stuff you see on your screen. At a deeper, somewhat hidden level, the two computers involved in the transaction are communicating with one another through *HTTP headers*. The headers are not normally visible to the human eye, but they are accessible to Python, your web browser, and other programs. You can choose to see the headers if you want, and actually doing so can be very handy when writing code to access the Web. The product we use most often to view the headers is called HTTP Headers, which is a Google Chrome extension. If you have Chrome and want to try it for yourself, use Chrome to browse to `https://www.esolutions.se/`, scroll down to Google Chrome Extensions, click HTTP Headers, and follow the instructions to install the extension. To see the headers involved whenever you've just visited a site, click the HTTP Headers icon in your Chrome toolbar (it looks like a cloud) and you'll see the HTTP header information as in Figure 3-3.

Two of the most important things in the HTTP Headers are right at the top, where you see GET followed by a URL. This tells you that a GET request was sent, meaning that the URL is just a request for information, nothing is being uploaded to the server. The URL after the word GET is the resource that was requested. Another type of response is POST, and that means there's some information you're sending to the server, such as when you *post* something on Facebook, Twitter, or any other site that accepts input from you.

The second line below the GET shows the status of the request. The first part indicates the protocol used. In the example in Figure 3-4, this is HTTP1.1, which just means it's a web request that's following the HTTP version 1.1 rules of communication. The 200 number is the status code, which in this case means "okay, everything went well." Common status codes are listed in Table 3-1.

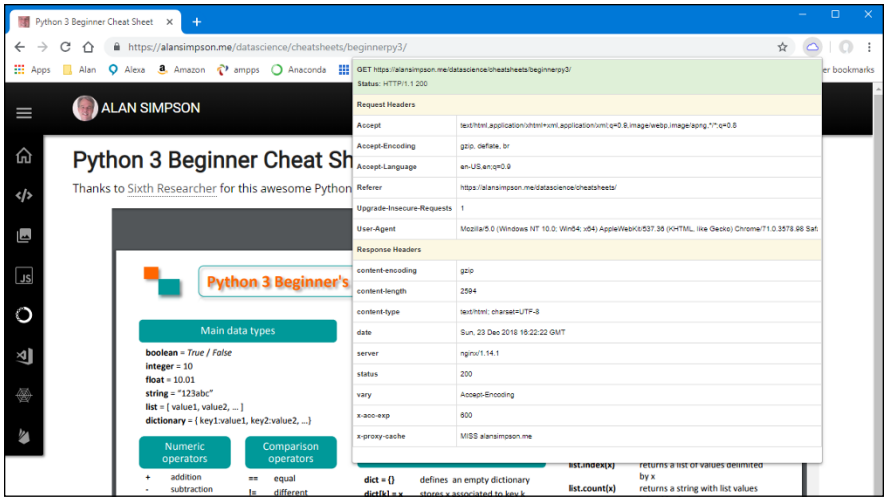


FIGURE 3-3: Inspecting HTTP headers with Google Chrome.

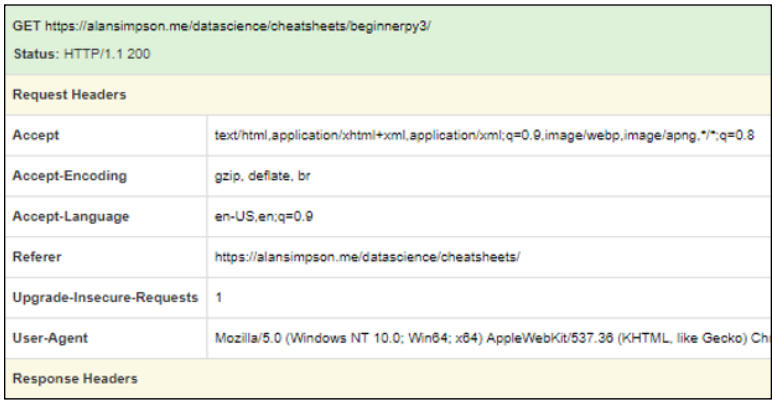


FIGURE 3-4: HTTP headers.

TABLE 3-1

Common HTTP Status Codes

Code	Meaning	Reason
200	Okay	No problems.
400	Bad Request	Server is available, but can't make sense of your request, usually because there's something wrong with your URL.
403	Forbidden	Site has detected you're accessing it programmatically, and doesn't allow that.
404	Not found	Either the URL is wrong, or the URL is right but the content that was there originally isn't there anymore.

All of what we've been telling you here matters because it's all related to accessing the Web programmatically with Python, as you'll see next.

Opening a URL from Python

To access the Web from within a Python program, you need to use aspects of the urllib package. The name *urllib* is short for *URL Lib*rary. This one library actually consists of modules, each of which provides capabilities that are useful for different aspects of accessing the Internet programmatically. Table 3-2 summarizes the packages.

TABLE 3-2 Packages from the Python urllib Library

Package	Purpose
request	Use this to open URLs
response	Internal code that handles the response that arrived; you don't need to work with that directly
error	Handles request exceptions
parse	Breaks up the url into smaller chunks
robotparser	Analyzes a site's robots.txt file, which grants permissions to bots that are trying to access the site programmatically

Most of the time you'll likely work with the request module, because that's the one that allows you to open resources from the Internet. The syntax for accessing a simple package from a library is

```
from library import module
```

. . . where *library* is the name of the larger library, and *module* is the name of the specific module. To access the capabilities of the response module of urllib, use this syntax at the top of your code (the comment above the code is optional):

```
# import the request module from urllib library.
from urllib import request
```

To open a web page, use this syntax:

```
variablename = request.urlopen(url)
```

Replace *variablename* with any variable name of your own choosing. Request *url* with the URL of the resource you want to access. You must enclose it in single- or double-quotation marks unless it's stored in a variable. If the URL is already stored in some variable, then just the variable name without quotation marks will work.

When running the code, the result will be an `HTTPResponse` object.

As an example, here is some code you can run in a Jupyter notebook or any .py file to access a sample HTML page Alan added to his own site just for this purpose:

```
# import the request module from urllib library.
from urllib import request
# URL (address) of the desired page.
sample_url = 'https://AlanSimpson.me/python/sample.html'
# Request the page and put it in a variables named the page.
thepage = request.urlopen(sample_url)
# Put the response code in a variable named status.
status = thepage.code
# What is the data type of the page?
print(type(thepage))
# What is the status code?
print(status)
```

Running this code displays this output:

```
<class 'http.client.HTTPResponse'>
200
```

This is telling you that the variable named `thepage` contains an `http.client.HTTPResponse` object . . . which is everything the server sent back in response to the request. The `200` is the status code that's telling you all went well.

Posting to the Web with Python

Not all attempts to access web resources will go as smoothly as the previous example. For example, type this URL into your browser's Address bar, and press Enter:

```
https://www.google.com/search?q=python web scraping tutorial
```

Google returns a search result of many pages and videos that contain the words *python web scraping tutorial.* If you look at the Address bar, you may notice that the

URL you typed has changed slightly and that blank spaces have all be replaced with `%20`, as in the following line of code:

```
https://www.google.com/search?q=python%20web%20scraping%20tutorial
```

That `%20` is the ASCII code, in hex, for a space, and the browser just does that to avoid sending the actual spaces in the URL. Not a big deal.

So now, let's see what happens if you run the same code as above but with the Google URL rather than the original URL. Here is that code:

```
from urllib import request
# URL (address) of the desired page.
sample_url = ' https://www.google.com/search?q=python%20web%20scraping%20
    tutorial'
# Request the page and put it in a variables named the page.
thepage = request.urlopen(sample_url)
# Put the response code in a variable named status.
status = thepage.code
# What is the data type of the page?
print(type(thepage))
# What is the status code?
print(status)
```

When you run this code, things don't go so smoothly. You may see several error messages, but the most important one is the one that usually reads something like this:

```
HTTPError: HTTP Error 403: Forbidden
```

The "error" isn't with your coding. Rather, it's an HTTP error. Specifically, it's error number 403 for "Forbidden." Basically your code worked. That is, the URL was sent to Google. But Google replied with "Sorry, you can search our site from your browser, but not from Python code like that." Google isn't the only site that does that. Many big sites reject attempts to access their content programmatically, in part to protect their rights to their own content, and in part to have some control over the incoming traffic.

The good news is, sites that don't allow you to post directly using Python or some other programming language often *do* allow you to post content. But you have to do so through their API (application programming interface). You can still use Python as your programming language. You just have to abide by their rules when doing so.

An easy way to find out whether a site has such an API is to simply Google your intention and language. For example, *post to facebook with python* or *post to twitter with python* or something like that. We won't attempt to provide an example here of actually doing such a thing, because they tend to change the rules often and anything we say may be outdated by the time you read this. But a Google search should get you want you need to know. If you get lots of results, focus on the ones that were posted most recently.

Scraping the Web with Python

Whenever you request a page from the Web, it's delivered to you as a *web page* usually consisting of HTML and *content*. The HTML is markup code that, in conjunction with another language called CSS, tells the browser *how* to display the content in terms of size, position, font, images, and all other such visual, stylistic matters. In our web browser, you don't see that HTML or CSS code. You see only the *content*, which is generally contained within blocks of HTML code in the page.

As a working example we're going to use the relatively simple page shown in Figure 3-5.

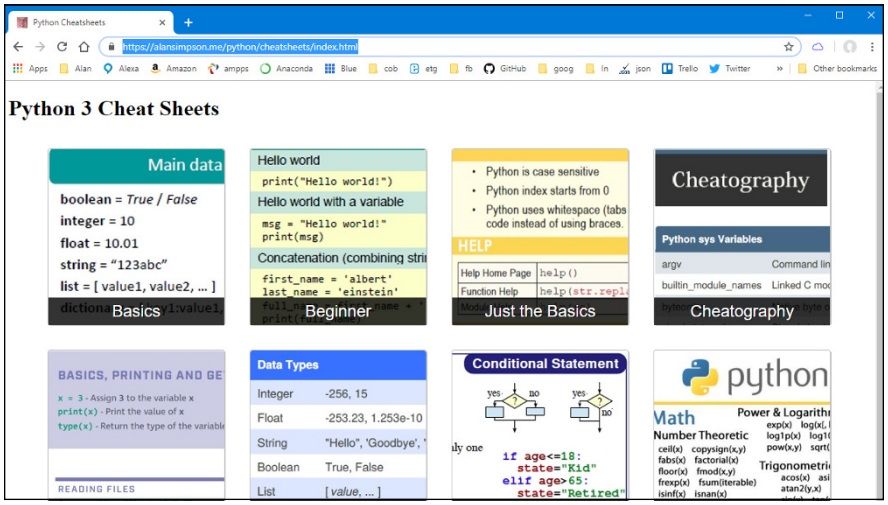


FIGURE 3-5: Sample page used for web scraping.

The code that tells the browser how to display that page's content isn't visible in the browser, unless you view the *source code*. In most browsers, you can do that by pressing the F12 key or by right-clicking an empty spot on the page and choosing View Source or Inspect or some other such option, depending on the brand and

version you're using. In most web pages, the real content — the stuff you see in the browser — is between the `<body> ... </body>` tags. Within the body of the page, there may be sections for a header, navigation bar footer, perhaps ads, or whatever. In that particular page, the real "meat" of the content is between `<article> ... </article>` tags. Each card that you see in the browser is defined as a link in `<a> ... </a>` tags.

Figure 3-6 shows some of the HTML code for the page in Figure 3-3. We're only showing code for the first two links in the page, but all the links follow the same structure. And they are all contained within the section denoted by a pair of `<article> ... </article>` tags.


```
<body>
    <h1>Python 3 Cheat Sheets</h1>
    <article>
        <a href="http://www.sixthresearcher.com/python-3-reference-cheat-sheet-for-beginners/">
            <img src="../../datascience/python/basics/basics256.jpg" alt="Python Basics Cheat Sheet">
            <span>Basics</span>
        </a>
        <a href="https://alansimpson.me/datascience/python/beginner/">
            <img src="../../datascience/python/beginner/beginner256.jpg" alt="Python Beginner Cheat Sheet">
            <span>Beginner</span>
        </a>
    </article>
</body>
```

FIGURE 3-6: Some of the code from the sample page for web scraping.

Notice that each link consists of several tags, as summarized here:

- **`<a> ... </a>`:** The a (sometimes called *anchor*) tags define where the browser should take the user when they click the link. The `href=` part of the `<a>` tag is the exact URL of the page to which the user should be taken.
- **`<img>`:** The `img` tag defines the image that shows for each link. The `src=` attribute in that tag defines the *source* of the image — in other words, the exact location and filename to show for that link.
- **`<span> ... </span>`:** At the bottom of the link is some text enclosed in `<span> ... </span>` tags. That text appears at the bottom of each link as white text against a black background.

The term *web scraping* refers to opening a web page, in order to pick its information apart programmatically for use in some other manner. Python has great web scraping capabilities, and this is a hot topic most people want to learn about. So for the first parts of this chapter we'll focus on that, using the sample page we just showed you as our working example.

The term *screen scraping* is also used as a synonym for *web scraping*. Though, as you'll see here, you're not actually scraping content from the computer screen. You're scraping it from the file that gets sent to the browser so that the browser can display the information on your screen.

In the Python code, you'll need to import two modules, both of which come with Anaconda so you should already have them. One of them is the `request` module from urllib (short for URL Library), which allows you to send a request out to the Web for a resource and to read what the Web serves back. The second is called `BeautifulSoup`, from a song in the book *Alice in Wonderland*. That one provides tools for parsing the web page that you've retrieved for specific items of data in which you're interested. So to get started, open up a Jupyter notebook or create a .py file in VS Code and type the first two lines as follows:

```
# Get request module from url library.
from urllib import request
# This one has handy tools for scraping a web page.
from bs4 import BeautifulSoup
```

Next, you need to tell Python where the page of interest is located on the Internet. In this case, the URL is

```
https://alansimpson.me/python/scrape_sample.html
```

You can verify this by typing the URL into the Address bar of your browser and pressing Enter. But to scrape the page, you'll need to put that URL in your Python code. You can give it a short name, like `page_url`, by assigning it to a variable like this:

```
# Sample page for practice.
page_url = 'https://alansimpson.me/python/scrape_sample.html'
```

To get the web page at that location into your Python app, create another variable, which we'll call `rawpage`, and use the `urlopen` method of the `request` module to read in the page. Here is how that code looks:

```
# Open that page.
rawpage = request.urlopen(page_url)
```

To make it relatively easy to parse that page in subsequent code, copy it over a `BeautifulSoup` object. We'll name the object `soup` in our code. You'll also have to tell `BeautifulSoup` how you want the page parsed. You can use `html5lib`, which also comes with Anaconda. So just add these lines:

```
# Make a BeautifulSoup object from the html page.
soup = BeautifulSoup(rawpage, 'html5lib')
```

Parsing part of a page

Most web pages contain lots of code for content in the header, footer, sidebars, ads, and whatever else is going on in the page. The main content is often just in one section. If you can identify just that section, your parsing code will run more quickly. In this example, in which we created the web page ourselves, we put all the main content between a pair of `<article> ... </article>` tags. In the following code, we assign that block of code to a variable named `content`. Later code in the page will parse only that part of the page, which can help improve speed and accuracy.

```
# Isolate the main content block.
content = soup.article
```

Storing the parsed content

You goal, when scraping a web page, is typically to collect just specific data of interest. In this case, we just want the URL, image source, and text for a number of links. We know there will be more than one line. An easy way to store these, for starters, would be to put them in a list. In this code we create an empty list named `links_list` for that purpose using this code:

```
# Create an empty list for dictionary items.
links_list = []
```

Next the code needs to loop through each link tag in the page content. Each of those starts and ends with an `<a>` tag. To tell Python to loop through each link individually, use the `find_all` method of `BeautifulSoup` in a loop. In the code below, as we loop through the links, we assign the current link to a variable named `link`:

```
# Loop through all the links in the article.
for link in content.find_all('a'):
```

Each link's code will look something like this, though each will have a unique URL, image source, and text:

```
<a href="https://alansimpson.me/datascience/python/lists/">
    <img src="../datascience/python/lists/lists256.jpg" alt="Python lists">
    <span>Lists</span>
</a>
```

The three items of data we want are:

- The link url, which is enclosed in quotation marks after the `href=` in the `<a>` tag.

- The image source, which is enclosed in quotation marks after `src=` in the `img` tag.
- The link text, which is enclosed in `<span> ... </span>` tags.

The following code teases out each of those components by using the `.get()` method on `BeautifulSoup` to isolate something inside the link (that is, in between the `<a>` and `</a>` tags that mark the beginning and end of each link). To get the URL portion of the link and put it in a variable named `url`, we use this code:

```
  url = link.get('href')
```

You have to make sure to indent that under the loop so it's executed for each link. To get the image source and put it in a variable named `img` we used this code:

```
img = link.img.get('src')
```

The text is between `<span> ... </span>` text near the bottom of the link. To grab that and put it into a variable named `text`, add this line of code:

```
text = link.span.text
```

You don't have to use `.get()` for that because the text isn't in an HTML attribute like `href=` or `src=`. It's just text between `<span> ... </span>` tags.

Finally, you need to save all that before going to the next link in the page. An easy way to do that is to append all three items of data to the `links_list` using this code:

```
links_list.append({'url' : url, 'img': img, 'text': text})
```

That's it for storing all the data for one link with each pass through the loop. There is one other caveat, though. Web browsers are very forgiving of errors in HTML code. So it's possible there will be mistyped code or missing code here and there that could cause the loop to fail. Typically this will be in the form of an attribute error, where Python can't find some attribute. If there's data missing, we prefer that the Python just skip the bad line and then keep going, rather then crash-and-burn leaving us with no data at all. So we should put the whole business of grabbing the parts in a `try:` block, which, if it fails, allows Python to just skip that one link and move onto the next. The code for that looks like this:

```
    # Try to get the href, image url, and text.
    try:
        url = link.get('href')
        img = link.img.get('src')
```

```
        text = link.span.text
        links_list.append({'url' : url, 'img': img, 'text': text})
    # If the row is missing anything...
    except AttributeError:
        #... skip it, don't blow up.
        pass
```

Figure 3-7 shows all the code as it stands right now. If you run it like that, you'd end up with the `link_list` being filled with all the data you scraped. But that doesn't do you much good. Chances are, you're going to want to save it as data to use elsewhere. You can do so by saving the data to a JSON file, a CSV file, or both, whatever is most convenient for you. In the sections that follow we show you how to do both.

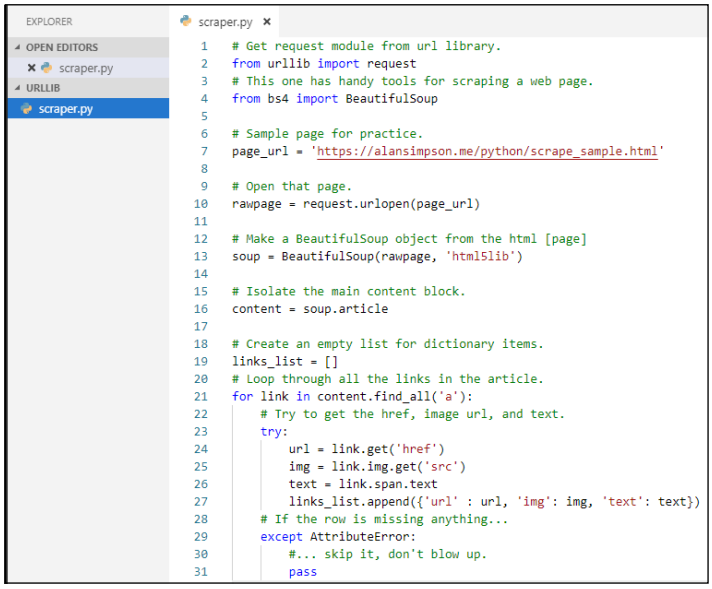


FIGURE 3-7: Web scraping code complete.

Saving scraped data to a JSON file

To save the scraped data to a JSON file, first import the `json` module near the top of your code, like this:

```
# If you want to dump data to json file.
import json
```

Then, below the loop (not indented, because you don't want to repeat the code for each pass through the loop), first open a file, for writing, using this code. You can

name you file anything you like. We've opted to name ours `links.json`, as you can see in the following code:

```
# Save as a JSON file.
with open('links.json', 'w', encoding='utf-8') as links_file:
```

Then finally, indented under that line, use `json.dump()` to dump the contents of `links_list` to that JSON file. We typically add `ensure_ascii=false` just to preserve any foreign characters, but that is entirely optional:

```
    json.dump(links_list, links_file, ensure_ascii=False)
```

That's it! After you run the code you'll have a file named `links.json` that contains all the scraped data in JSON format. If you open it from VS Code, it will look like one long line, because we didn't add any line breaks or spaces to indent. But when you see it as one long line, you can copy/paste the whole thing to a site like `jsonformatter.org`, which will display the data in a more readable format without changing the content of the file, as in Figure 3-8.

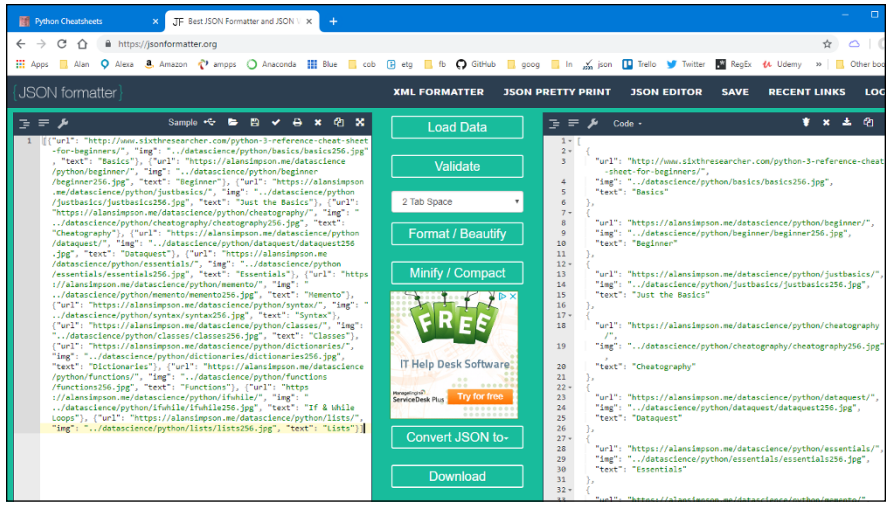


FIGURE 3-8: Web scraped data in a JSON file.

Saving scraped data to a CSV file

If, for whatever reason, you prefer to go straight to a CSV file with our scraped data, start by importing the `csv` module near the top of your code, like this:

```
# If you want to save to CSV.
import csv
```

Then, down below and outside of the loop that creates `links_list`, open in write mode a file with the filename of your choosing. In the following code we named ours `links.csv`. We also used `newline=''` to avoid putting in an extra newline at the end of each row.

```
# Save it to a CSV.
with open('links.csv', 'w', newline='') as csv_out:
```

Indented below that `open`, create a `csv` writer that targets the file based on the name you assigned at the end of the `with . . .` line:

```
    csv_writer = csv.writer(csv_out)
```

The first row of a CSV typically contains field names (or column headings, as they're also called). So the next step is to add that row to the table, using whatever names you want to apply to headings, like this:

```
    # Create the header row
    csv_writer.writerow(['url','img','text'])
```

Then you can write all the data from `link_list` to the CSV file by looping through `link_list` and writing the three items of data, separated by commas, to new rows. Here is that code:

```
    for row in links_list:
        csv_writer.writerow([str(row['url']),str(row['img']),str(row['text'])])
```

Running the code produces a file named `links.csv`. If you open that file in Excel or another spreadsheet app, you'll see that the data is neatly organized into a table with columns labeled `url`, `img`, and `text`, as in Figure 3-9.

	A	B	C
1	url	img	text
2	http://www.sixthresearcher.com/python-3-reference-cheat-sheet	../datascience/python/basics/basics256.jpg	Basics
3	https://alansimpson.me/datascience/python/beginner/	../datascience/python/beginner/beginner256.jpg	Beginner
4	https://alansimpson.me/datascience/python/justbasics/	../datascience/python/justbasics/justbasics256.jpg	Just the Basics
5	https://alansimpson.me/datascience/python/cheatography/	../datascience/python/cheatography/cheatography256.jpg	Cheatography
6	https://alansimpson.me/datascience/python/dataquest/	../datascience/python/dataquest/dataquest256.jpg	Dataquest
7	https://alansimpson.me/datascience/python/essentials/	../datascience/python/essentials/essentials256.jpg	Essentials
8	https://alansimpson.me/datascience/python/memento/	../datascience/python/memento/memento256.jpg	Memento
9	https://alansimpson.me/datascience/python/syntax/	../datascience/python/syntax/syntax256.jpg	Syntax
10	https://alansimpson.me/datascience/python/classes/	../datascience/python/classes/classes256.jpg	Classes
11	https://alansimpson.me/datascience/python/dictionaries/	../datascience/python/dictionaries/dictionaries256.jpg	Dictionaries
12	https://alansimpson.me/datascience/python/functions/	../datascience/python/functions/functions256.jpg	Functions
13	https://alansimpson.me/datascience/python/ifwhile/	../datascience/python/ifwhile/ifwhile256.jpg	If & While Loops
14	https://alansimpson.me/datascience/python/lists/	../datascience/python/lists/lists256.jpg	Lists

FIGURE 3-9: Web scraped data in Excel.

Figure 3-10 shows all the code for scraping both to JSON and CSV. We removed some comments from the top of the code to get it to all fit in one screenshot. But we just wanted to make sure you can see it all on one place. Seeing all the code in the proper order like that should help you debug your own code, if need be.

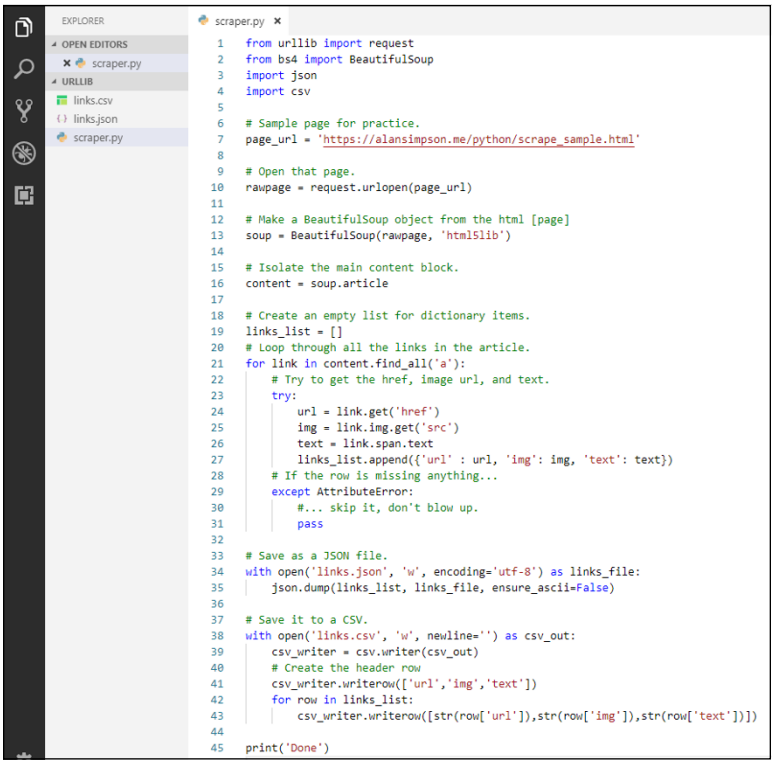


FIGURE 3-10: The entire `scraper.py` program.

Accessing the Web programmatically opens whole new worlds of possibilities for acquiring and organizing knowledge. In fact, there is a whole field of study, called data science, that's all about just that. There are many specialized tools available too, that go far beyond what you've learned here. These you'll learn about in Book 5 of this book.

But before launching into the more advanced and specialized applications of Python, there is just one more fundamental concept we need to discuss. Throughout these first chapters you've used many different kinds of libraries and modules and such. In the next chapter you learn just how far-reaching that is, and how to look for, and find, what you need when you need it.

IN THIS CHAPTER

- Understanding the standard library
- Exploring Python packages
- Importing Python modules
- Creating your own Python modules

Chapter 4
Libraries, Packages, and Modules

For the most part, all the chapters leading up to this one have focused on the core Python language, the elements of the language you'll need no matter how you intend to use Python in the future. But as you've seen, many programs start by importing one or more modules. Each module is essentially a collection of pre-written code that you can use in your own code without having to reinvent that wheel. The granddaddy of all this pre-written specialized code is called the Python standard library.

Understanding the Python Standard Library

The Python standard library is basically all the stuff you get when you get the Python languages. That includes all the Python data types like string, integer, float, and Boolean. Every instance of those data types is actually an instance of a class defined in the standard library. For this reason, the terms *type*, *instance*, and *object* are often used interchangeably. An integer is a whole number; it's also a data type in Python. But it exists because the standard library contains a class for integers, and every integer you create is actually an instance of that class and hence an object (because classes are the templates for things called objects).

The `type()` function in Python usually identifies the type of a piece of data. For example, run these two lines of code at a Python prompt, in a Jupyter notebook or a .py file:

```
x = 3
print(type(x))
```

The output is:

```
<class 'int'>
```

This is telling you that `x` is an integer, and also that it's an instance of the `int` class from the standard library. Running this code:

```
x = 'howdy'
print(type(x))
```

Produces this output:

```
<class 'str'>
```

That is, x contains data that's the string data type, created by the Python `str` class. The same thing works for a float (a numeric value with a decimal point, like 3.14) and for Booleans (`True` or `False`).

Using the dir() function

The Python standard library offers a `dir()` method that displays a list of all the attributes associated with a type. For example, in the previous example the result `<class 'str'>` tells you that the data is the `str` data type. So you know that's a type, and thus in instance of a class called `str` (short for *`string`*). Entering this command:

```
dir(str)
```

Displays something like this:

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
   '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
   '__getnewargs__', '__gt__', '__hash__', '__init__','__init_subclass__',
   '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
   '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
   '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize',
   'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
```

```
    'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii',
    'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
    'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
    'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
    'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
    'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

The dunder named items (the names surrounded by double-underlines) usually represent something that's built into Python and that plays some role in the language that you don't necessarily access directly. These are often referred to as *special variables* or *magic methods.* For example, there's an `__add__` method that's actually invoked by using the `+` (addition) operator to add two numbers or join together two strings.

The regular functions don't have the double underscores and are typically followed by parentheses. For example, take a look at these lines of code:

```
x = "Howdy"
print(type(x), x.isalpha(), x.upper())
```

The output from that code is:

```
<class 'str'> True HOWDY
```

The first part, `<class 'str'>` tells you that `x` contains a string. As such, you can use any of the attributes shown in the output of `dir(str)` on it. For example, the `True` is the output from `x.isalpha()` because `x` does contain alphabetic characters. The `HOWDY` is the output of `x.upper()`, which converts the string to all uppercase letters.

Beginners often wonder what good seeing a bunch of names like `'capitalize'`, `'casefold'`, `'center'`, `'count'`, `'encode'`, `'endswith'`, `'expandtabs'`, `'find'`, `'format'`, and so forth does for you when you don't know what the names mean or how to use them. Well, they don't really help you much if you don't pursue it any further. You can get some more detailed information by using `help()` rather than `dir`.

Using the help() function

The Python prompt also offers a `help()` function with the syntax:

```
help(object)
```

To use it, replace *object* with the object type with which you're seeking help. For example, to get help with `str` objects (strings, which come from the `str` class) enter this command at the Python prompt:

```
help(str)
```

The output will be more substantial information about the topic in the parentheses. For example, where `dir(str)` lists the names of attributes of that type, `help(dir)` provides more detail about each item. For example, whereas `dir(str)` tells you that there's a thing called `capitalize` in the `str` class, `help` tells you a bit more about it, as follows:

```
capitalize(self, /)
 Return a capitalized version of the string.
 More specifically, make the first character have upper case and the rest lower
   case.
```

The word `self` there just means that whatever word you pass to `capitalize` is what gets capitalized. The `/` at the end marks the end of positional-only parameters, meaning that you can't use keywords with parameters after that like you can when defining your own functions. What usually works best for most people is a more in-depth explanation and one or more examples. For those, Google or a similar search engine is usually your best bet. Start the search with the word *Python* (so it knows what the search is in reference too) followed by the exact word with which your seeking assistance. For example, searching Google for

```
python capitalize
```

. . . provides links to lots of different resources for learning about the `capitalize` attribute of the `str` object, including examples of its use.

TIP

If you get tired of pressing any key to get past More . . . at the end of every page in help, just press Ctrl+C. This gets you back to the Python prompt.

Of course, a really good (albeit technical) resource for the Python standard library is the standard library documentation itself. This is always available at `https://docs.python.org/` usually under the link Library Reference. But even that wording may change, so if in doubt, just google *python standard library*. Just be forewarned that it is huge and very technical. So don't expect to memorize or even understand it all right off the bat. Use it as an ongoing resource to learn about things that interest you as your knowledge of Python develops.

The documentation that appears at `docs.python.org` will generally be for the current stable version. Links to older versions, and to any newer versions that may be in the works when you visit, are available from links at the left side of the page.

Exploring built-in functions

Both `dir()` and `help()` are examples of Python built-in functions. These are functions that are always available to you in Python, in any app you're creating as well as at the Python command prompt. These built-in functions are also a part of the standard library. In fact, if you google *Python built-in functions*, some of the search results will point directly to the Python documentation. Clicking that link will open that section of the standard library documentation and displays a table of all the built-in functions, as in Figure 4-1. On that page, you can click the name of any function to learn more about it.

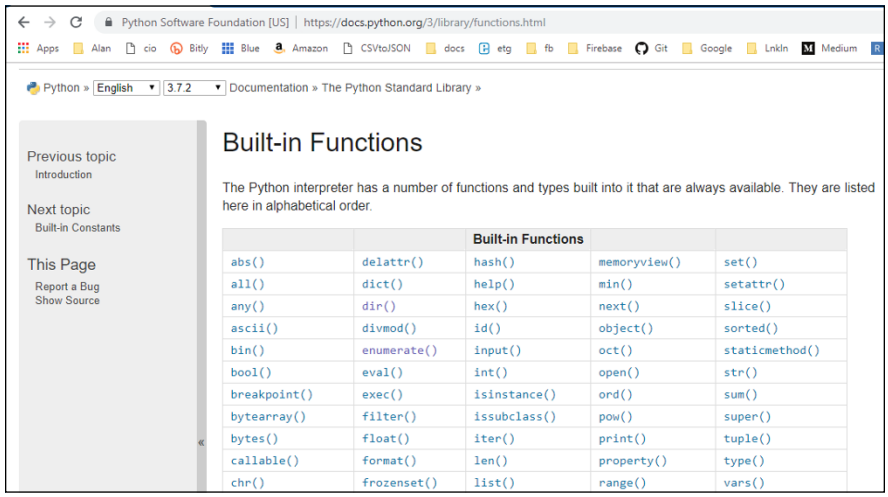


FIGURE 4-1: Python's built-in functions.

Exploring Python Packages

The Python language supports *modular programming*, where the idea is that there's no need for every individual programmer to reinvent everything herself. Even the largest projects can be broken down into smaller, more manageable components or modules. And in fact, a large project may actually require some components that already exist out there in the world somewhere, because somebody already needed the functionality and spent the time to create, test, and debug it.

Any large project, whether you're working alone or as a team member, can simplified and streamlined if some components can use tried-and-true code that's already been written, tested, debugged, and deemed reliable by members of the Python programming community. The *packages* you hear about in relation to Python are exactly that kind of code — code that's been developed and nurtured, is trustworthy, and generic enough that it can be used as a component of some large project that you're working on.

There are literally thousands of packages available for Python. A good resource for finding packages is PyPi, a clever name that's easy to remember and short for *Py*thon *P*ackage *I*ndex. You can check it out at any time using any browser and the URL `https://pypi.org/`. There is also a program named pip, another clever name, which stands for *P*ip *I*nstalls *P*ackages. It's commonly referred to as a *package manager* because you can also use it to explore, update, and remove existing packages.

To use pip, you have to get to your operating system's command prompt, which is Terminal on a Mac, or `cmd.exe` or Powershell in Windows. If you're using VS Code, the simplest way to get there is to open VS Code and choose View➪Terminal from its menu bar.

If you already have pip, typing this command at a command prompt will tell you which version of pip is currently installed:

```
pip --version
```

The result will likely look something like this (but with your version's numbers and names):

```
pip 18.1 from C:\Users\...\AppData\Local\Continuum\anaconda3\lib\site-packages\
   pip (python 3.7)
```

To see what packages you already have installed, enter this at the operating system command prompt:

```
pip list
```

Most people are surprised at how many packages they already have installed. It's a very lengthy list, and you have to scroll up and down quite a bit to see all the names. However, one of the advantages of installing Python with Anaconda is that you get lots of great packages in the mix. And you don't have to rely on `pip list` to find their names. Instead, just open Anaconda Navigator and click Environments in the left column. You'll see a list of all your installed packages, along with a brief description and version number for each, as shown in the example in Figure 4-2.

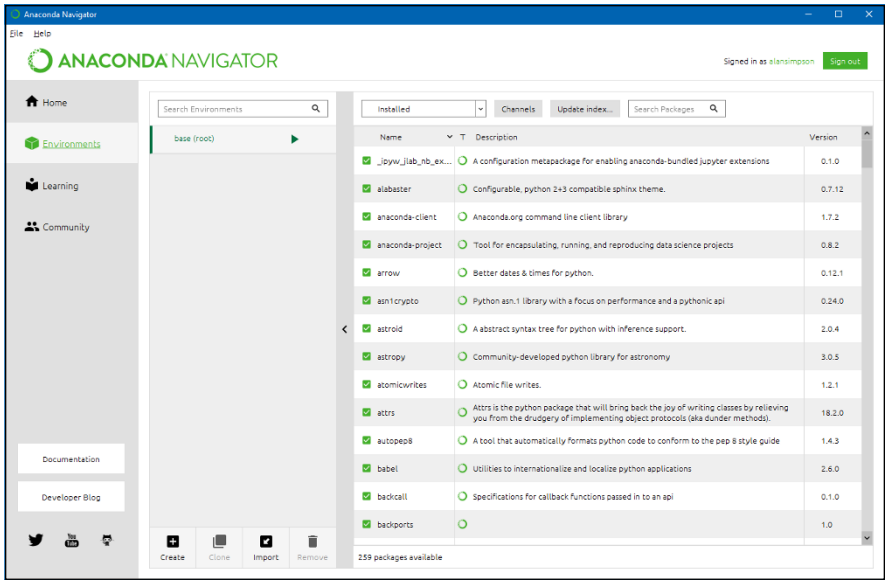


FIGURE 4-2: Installed packages as viewed in Anaconda.

TIP

See Book 1 for more information on installing and using Anaconda.

Although it's perfectly okay to use pip to install any packages that you don't already have, the one disadvantage is that those packages may not show up in Anaconda Navigator's list of installed packages. To get around that, any time you see an instruction to *pip* something to install it, you can try replacing the word *pip* with the word *conda* (short for Anaconda). This adds the package to your collection, so it will show up both when you do `pip list` and when you look at the list in Anaconda Navigator.

Importing Python Modules

You'll hear the word *module* used in conjunction with Python all the time. If you think of the standard library as an actual physical library, and a package as being, perhaps, one book in that library, then you can think of a module as being one chapter in one book. In other words, a package may contain many modules, and a library may contain many packages. The module is a big part of what makes Python a modular language, because code is grouped together according to function. So in the one hand, you don't have to import everything including the kitchen sink to use *some* code. By the same token, if you need to use several related things, such as functions for working with dates and times, you don't have to import them all one at a time. Typically, importing just the whole module will get you what you need. For example, to work with dates and times, you don't have

to import every Python module out there. Nor do you need to import every possible little function and object one at a time. You just import the whole module, like `datetime`, to get lots of handy things for working with dates and times.

There are actually a few ways to import functionality from modules. One of the most common is to simply import the whole module. To do that, you just follow the `import` command with the name of the module you want to import. For example, this imports the entire `math` module, which has lots of functions and stuff for doing math:

```
import math
```

After you import a module, the `dir()` and `help()` functions work on that too. For example, if you tried doing `dir(math)` or `help(math)` before `import math`, you'd get an error. That's because that `math` package isn't part of the standard library. However, if you do `import math` first, and then `help(math)`, then it all works.

There may be times where you don't really need the whole kit-and-caboodle. In those cases, you can import just what you need using a syntax like this:

```
from math import pi
```

In this example, you're just importing one thing (`pi`), so you're not bringing in unnecessary stuff. The second example is also handy because in your code you can refer to `pi` as just `pi`, you don't have to use `math.pi`. Here is some stuff you can try at the Python prompt, such as in a VS Code Terminal window, to see for yourself:

Enter the command `print(pi)` and press Enter. Most likely you'll get an error that reads:

```
NameError: name 'pi' is not defined
```

In other words, `pi` isn't part of the standard library that's always available to your Python code. To use it, you have to import the `math` module. There are two ways to do that. You can import the whole thing by typing this at the Python prompt:

```
import math
```

But if you do that and then enter

```
print(pi)
```

. . . you'll get the same error again, even though you imported the `math` package. The reason for that is when you import an entire module and you want to use a

part of it, you have to precede the part you want to use with the name of the module and a dot. For example, if you enter this command:

```
print(math.pi)
```

. . . you get the correct answer:

```
3.141592653589793
```

Be aware that when you import just part of a module, the `help()` and `dir()` functions for the whole module won't work. For example, if you've only executed `from math import pi` in your code and you attempt to execute a `dir(math)` or `help(math)` function, it won't work, because Python doesn't have the entire module at its disposal. It only has at its disposal that which you imported, `pi` in this example.

Usually `help()` and `dir()` are just things you use at the Python prompt for a quick lookup. They're not the kinds of things you're likely to use when actually writing an app. So using `from` rather than `import` is actually more efficient because you're only bringing in what you need. As an added bonus, you don't have to precede the function name with the module name and a dot. For example, when you import only `pi` from the `math` module, like this:

```
from math import pi
```

Then you can refer to `pi` in your code is simply `pi`, not `math.pi`. In other words, if you execute this function:

```
print(pi)
```

. . . you'll get the right answer:

```
3.141592653589793
```

This is because Python now "knows" of `pi` as being the thing named `pi` from the `math` module; you don't have to specifically tell it to use `math.pi`.

You can also import multiple items from a package by listing their names, separated by commas, at the end of the `from ...` command. For example, suppose you need `pi` and square roots in your app. You could import just those into your app using this syntax:

```
from math import pi, sqrt
```

Once again, because you used the `from` syntax for the import, you can refer to `pi` and `sqrt()` in your code by name without the leading module name. For example, after executing that `from` statement, this code:

```
print(sqrt(pi))
```

. . . displays

```
1.7724538509055159
```

. . . which, as you may have guessed, is the square root of the number pi.

You may also notice people importing a module like this:

```
from math import *
```

The asterisk is short for "everything." So in essence, that command is exactly the same as `import math`, which also imports the entire `math` module. But this is a subtle difference: When you do `from math import *` you associate the name of everything in the `math` module with that module. So you can use those names without the `math.` prefix. In other words, after you execute this:

```
from math import *
```

. . . you can do a command like `print(pi)` and it will work, even without using `print(math.pi)`. Although it does seem like a smart and convenient thing to do, we should point out that many programmers think that sort of thing isn't very Pythonic. If you're importing lots of modules and using lots of different pieces of each, avoiding module names in code can make it harder for other programmers to read and make sense of that code. So although in a *Zen of Python* sense it may be frowned upon, technically it works and is an option for you.

Making Your Own Modules

For all the hoopla about modules, a module is actually a pretty simple thing. In fact, it's just a file with a .py extension that contains Python code. That's it. So any time you write Python code and save it in a .py file, you've basically created a module. That's not to say you always have to use that code as a module. It can certainly be treated as a standalone app. But if you *wanted* to create your own module, with just code that you need often in your own work, you could certainly do so. We explain the whole process in this section.

A module is also just a file with a .py filename extension. The name of the module is the same as the filename (without the .py). Like any .py file, the module contains Python code. As a working example, let's suppose you want to have three functions to simplify formatting dates and currency values. You can make up any name you like for each function. For our working example, we'll use these three names:

- **to_date(*any_str*):** Lets you pass in any string (*any_str*) date in *mm/dd/yy* or *mm/dd/yyyy* format and sends back a Python `datetime.date` that you can use for date calculations.
- **mdy(*any_date*):** Lets you pass in any Python date or datetime, and returns a string date formatted in *mm/dd/yyyy* format for display on the screen.
- **to_curr(*any_num, len*):** Lets you pass in any Python float or integer number and returns a string with a leading dollar sign, commas in thousands places, and two digits for the pennies. The *len* is an optional number for length. If provided, the return value will be padded on the left with spaces to match the length specified

So here is all the code for that:

```
# Contains custom functions for dates and currency values.
import datetime as dt

def to_date(any_str):
    """ Convert mm/dd/yy or mm/dd/yyyy string to datetime.date, or None if
   invalid date. """
    try:
        if len(any_str) == 10:
            the_date = dt.datetime.strptime(any_str,'%m/%d/%Y').date()
        else:
            the_date = dt.datetime.strptime(any_str,'%m/%d/%y').date()
    except (ValueError, TypeError):
            the_date = None
    return the_date

def mdy(any_date):
    """ Returns a string date in mm/dd/yyyy format. Pass in Python date or
   string date in mm/dd/yyyy format """
    if type(any_date) == str:
        any_date = to_date(anydate)
    # Make sure its a dateime being forwarded
    if isinstance(any_date,dt.date):
        s_date = f"{any_date:'%m/%d/%Y'}"
    else:
        s_date = "Invalid date"
    return s_date
```