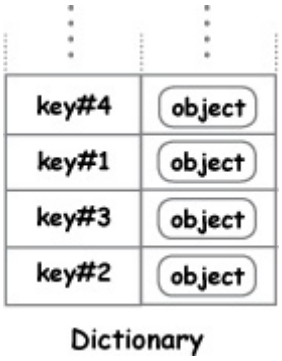
Chapter 3. Structured Data: Working with Structured Data

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

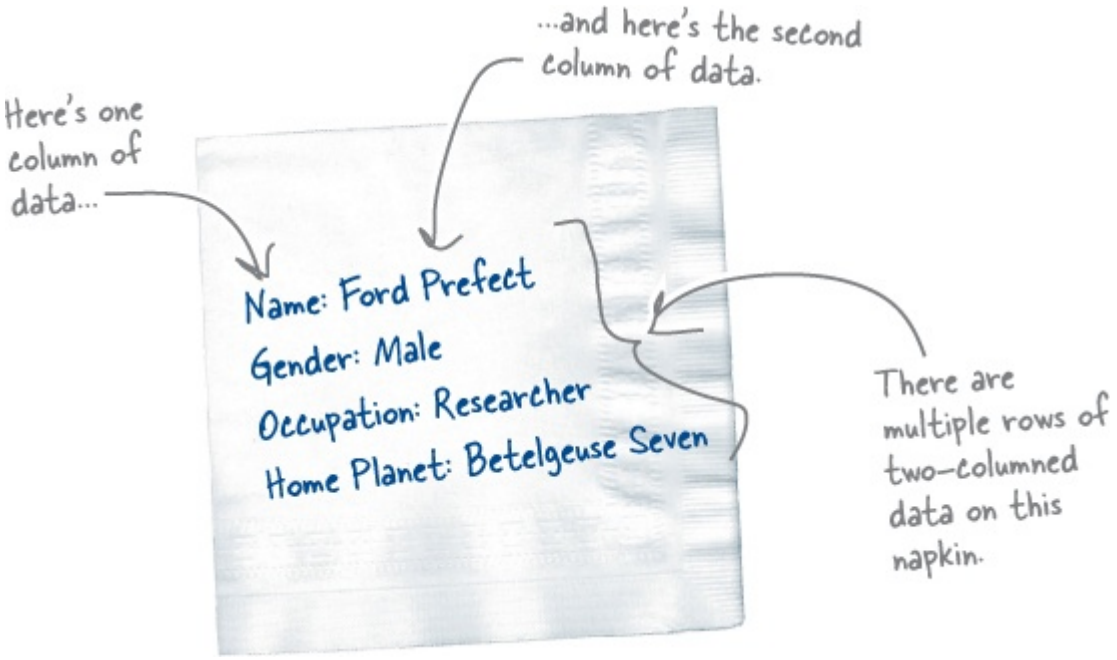
Unlike a list, which is a collection of related objects, the **dictionary** is used to hold a collection of **key/value pairs**, where each unique *key* has a *value* associated with it. The dictionary is often referred to as an *associative array* by computer scientists, and other programming languages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

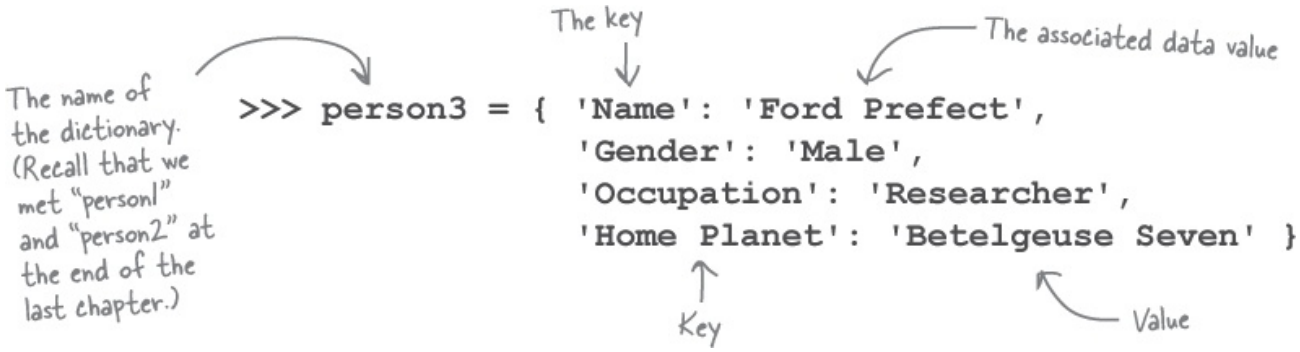
Data that conforms to the dictionary model is easy to spot: there are **two columns**, with potentially **multiple rows** of data. With this in mind, take another look at our "data napkin" from the end of the last chapter:

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



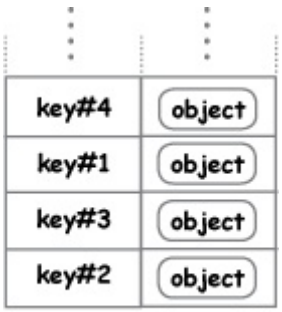
It looks like the data on this napkin is a perfect fit for Python's dictionary.

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:



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:



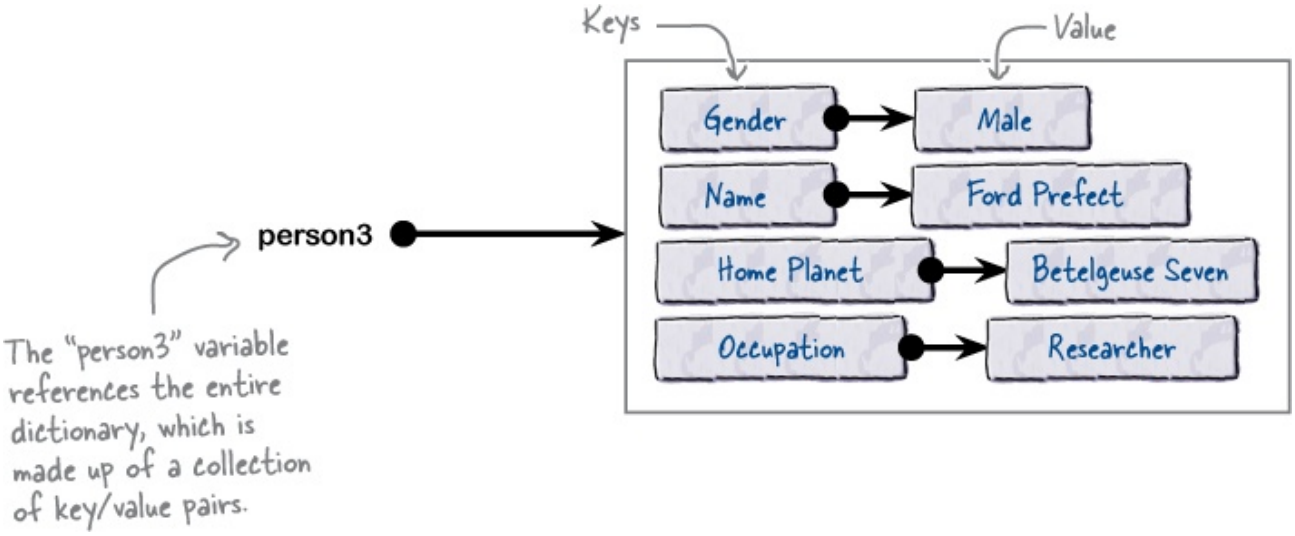
Dictionary

```
>>> 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:

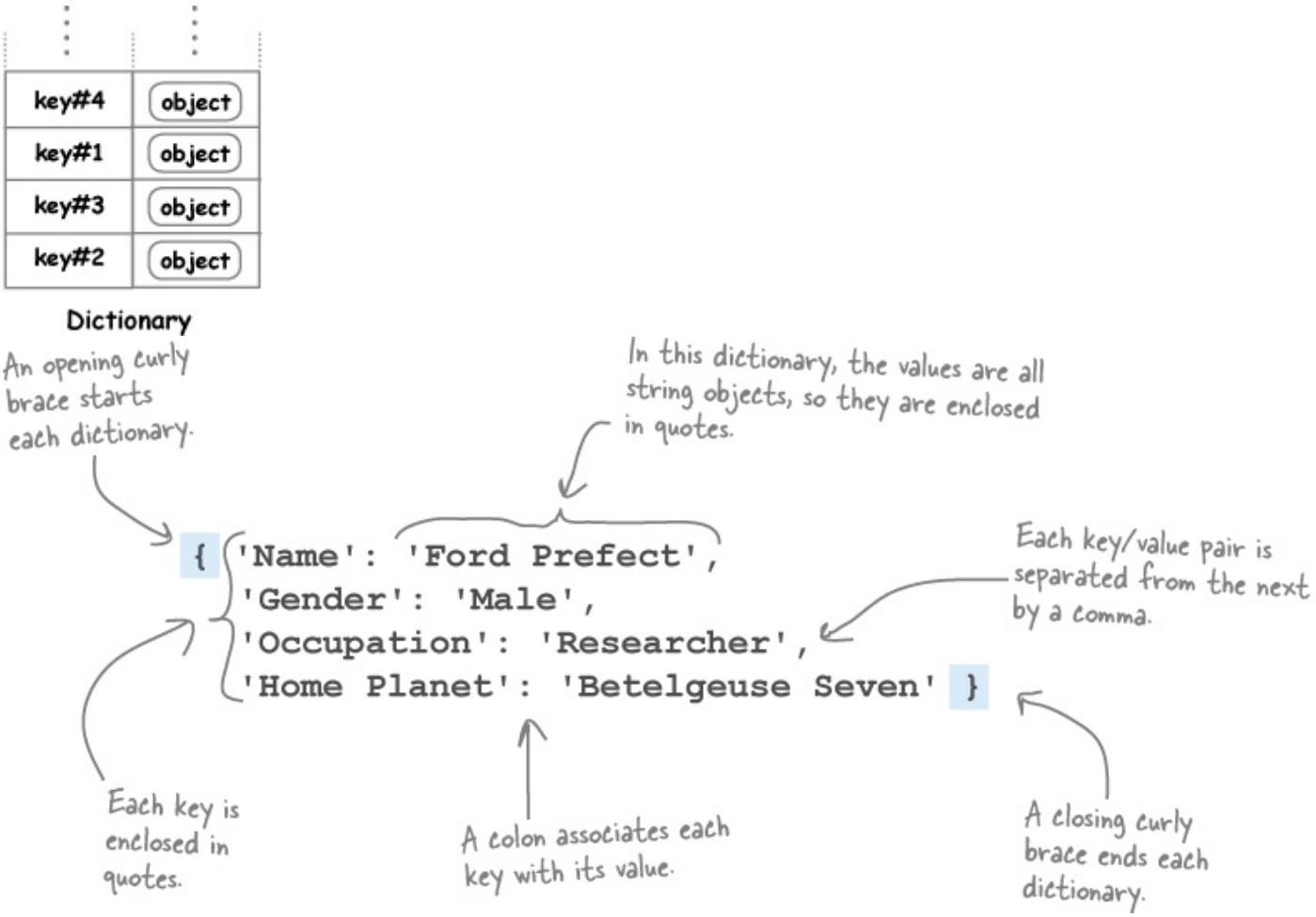


This is a more complicated structure than the array-like list. If the idea behind Python's dictionary is new to you, it's often useful to think of it as a **lookup table**. The key on the left is used to *look up* the value on the right (just like you look up a word in a paper dictionary).

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**:



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.

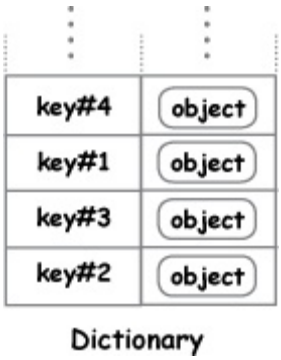
WHAT HAPPENED TO THE INSERTION ORDER?

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.

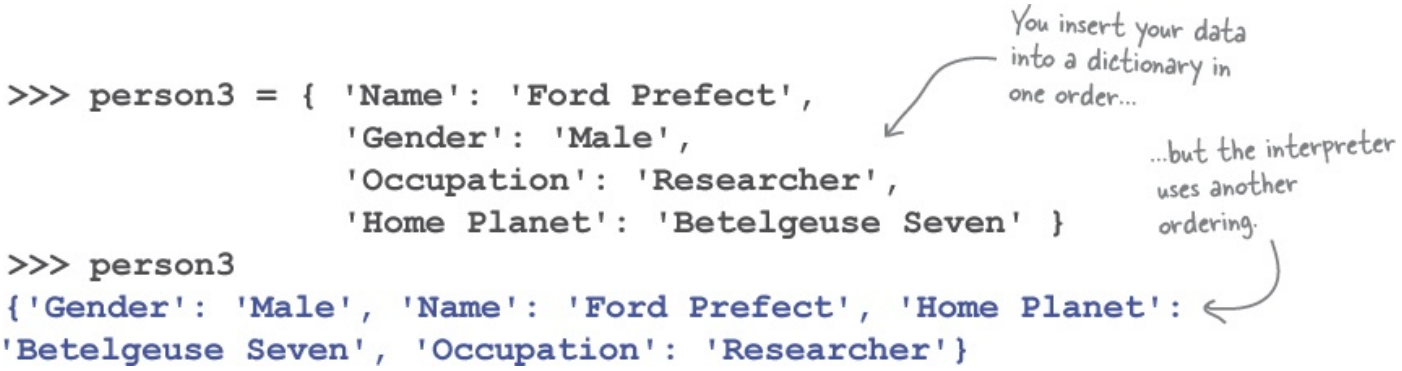
What's going on here? Why did the ordering change?

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:

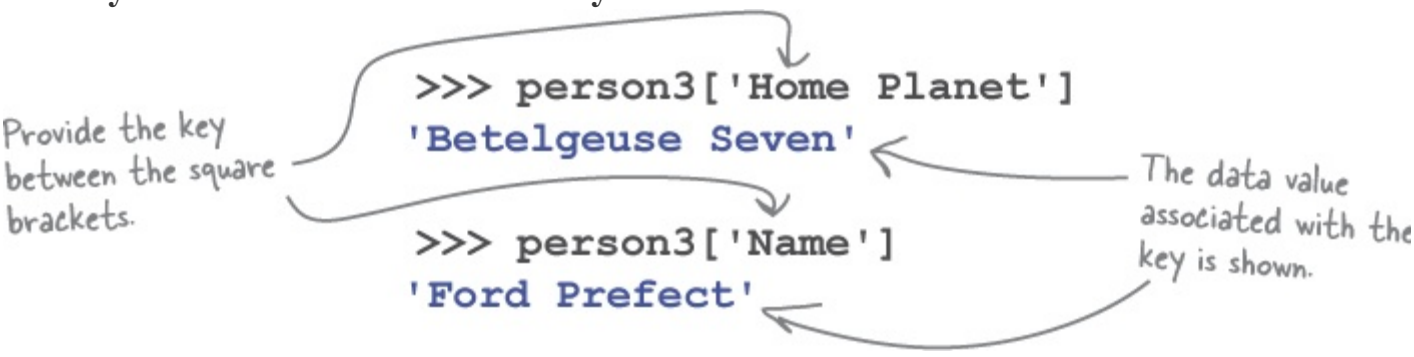


If you're scratching your head and wondering why you'd want to trust your precious data to such an unordered data structure, don't worry, as the ordering rarely makes a difference. When you select data stored in a dictionary, it has nothing to do with the dictionary's order, and everything to do with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

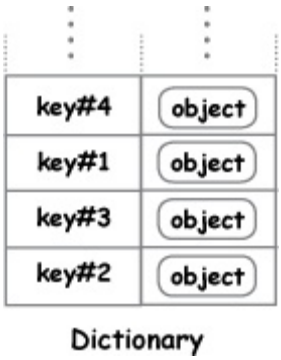
Use keys to access data in a dictionary.



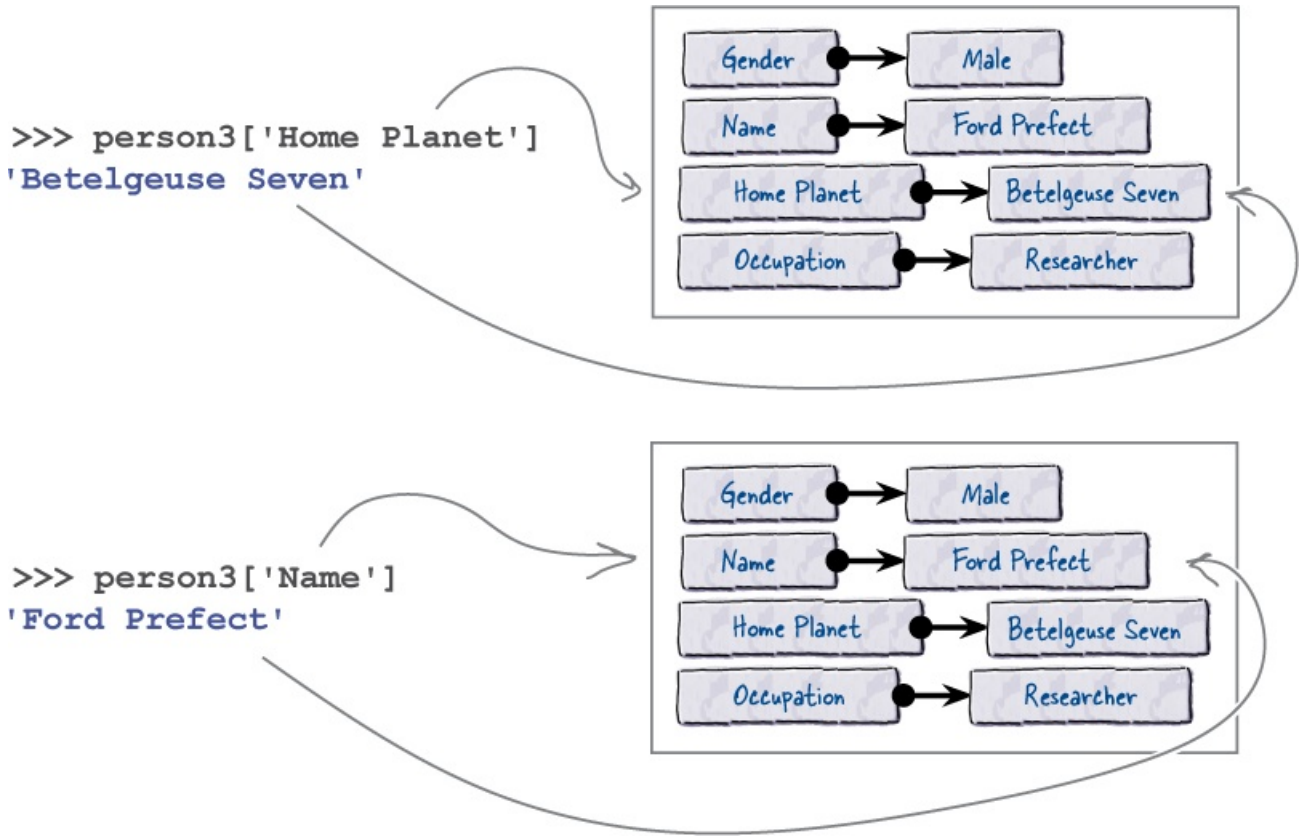
When you consider you can access your data in this way, it becomes apparent that it does not matter in what order the interpreter stores your data.

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:



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.

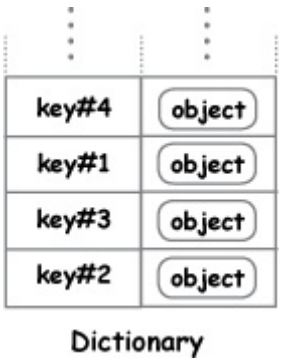
DICTIONARY LOOKUP IS FAST!

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.

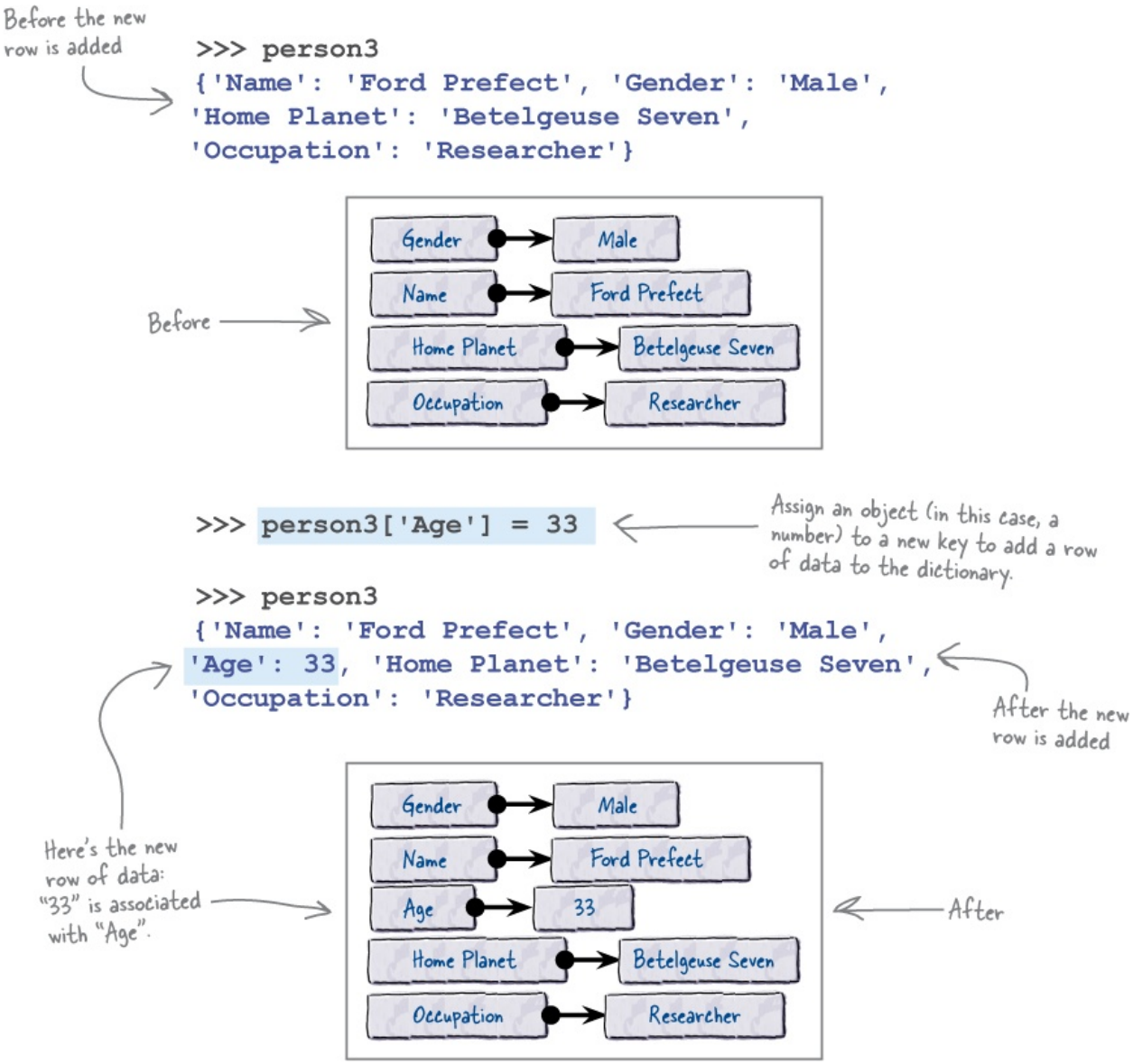
It does not matter in what order the dictionary is stored. All that matters is that the interpreter can access the value associated with a key *quickly* (no matter how big your dictionary gets). The good news is that the interpreter does just that, thanks to the employment of a highly optimized *hashing 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.

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:



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.

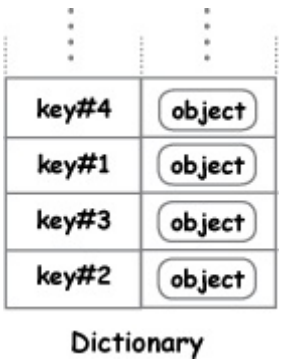
THERE ARE NO DUMB QUESTIONS

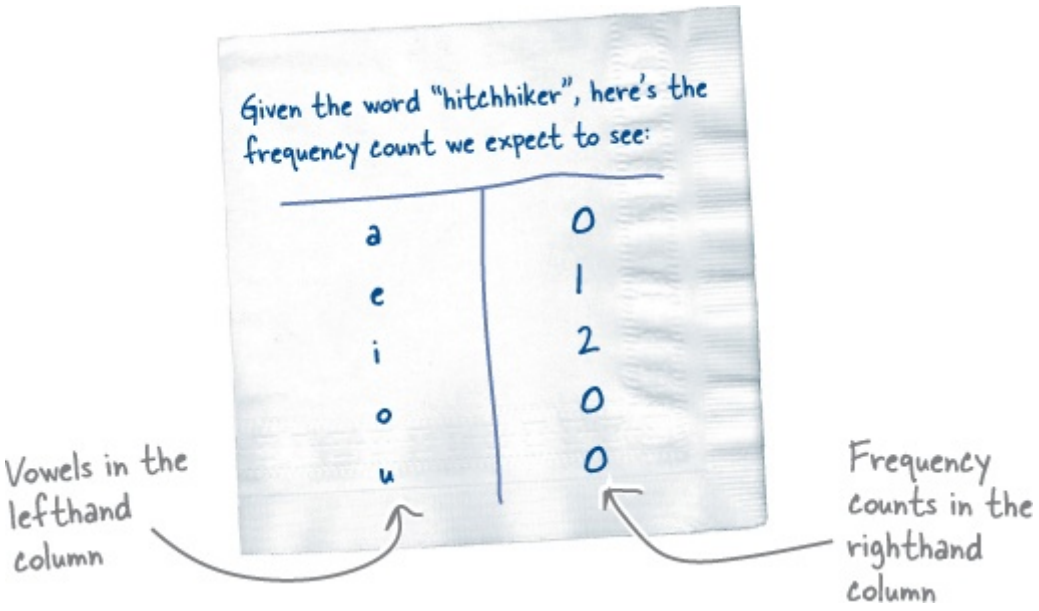
Q: **Q: Is it just me, or is the word "dictionary" a strange name for something that's basically a table?**

A: **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:

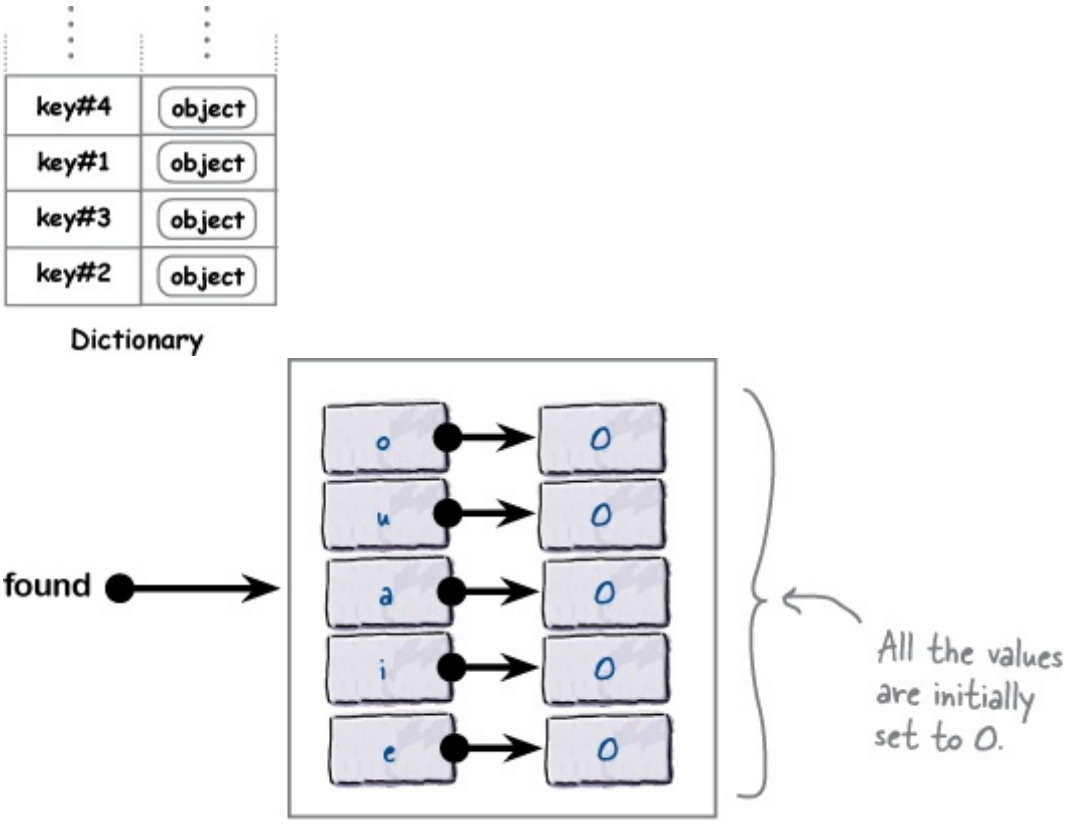
```
>>> found['a'] = 0
>>> found['e'] = 0
>>> found['i'] = 0
>>> found['o'] = 0
>>> found['u'] = 0
>>> found
{'o': 0, 'u': 0, 'a': 0, 'i': 0, 'e': 0}
```

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

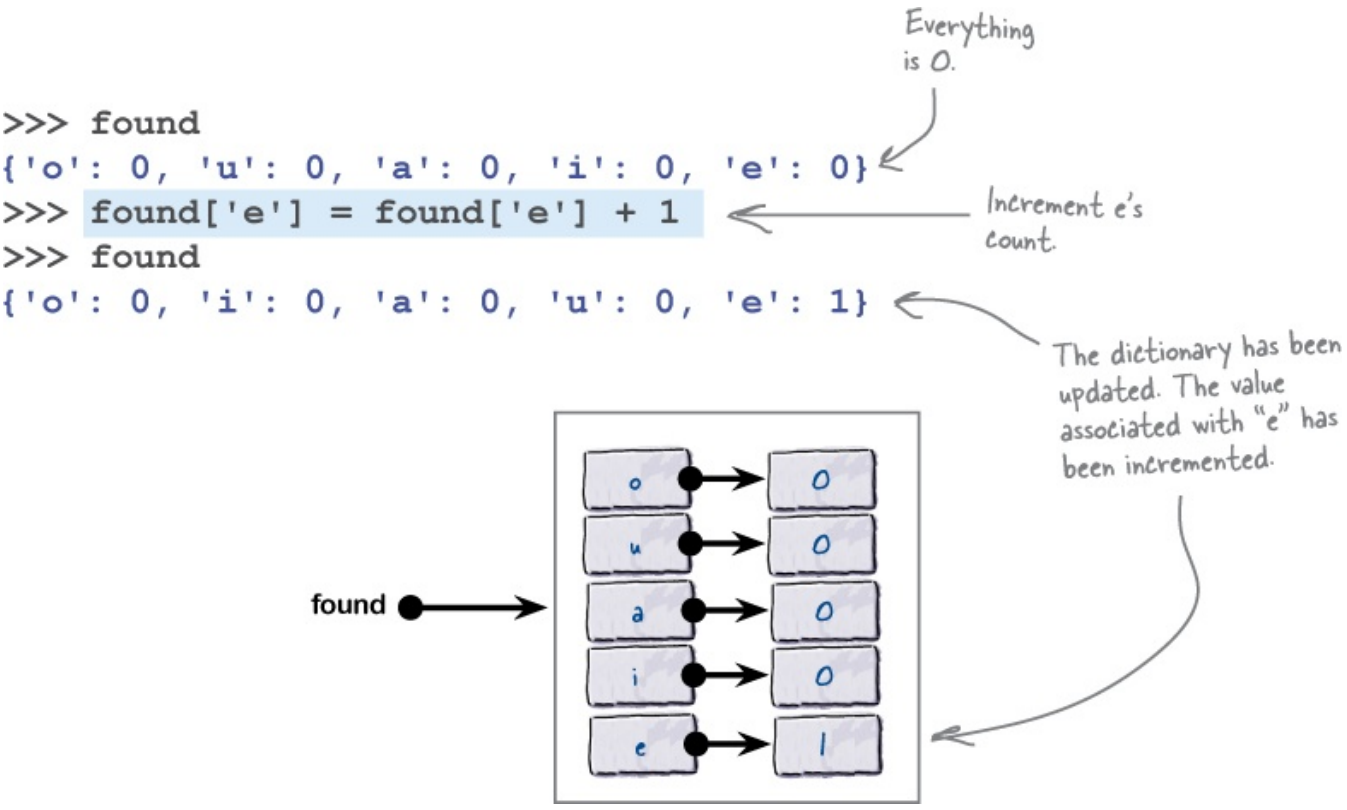
All we need to do now is find a vowel in a given word, then update these frequency counts as required.

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:



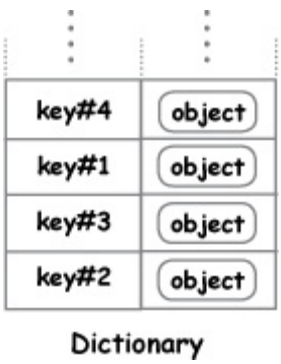
With the frequency counts initialized to 0, it's not difficult to increment any particular value, as needed. For instance, here's how to increment `e`'s frequency count:



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:

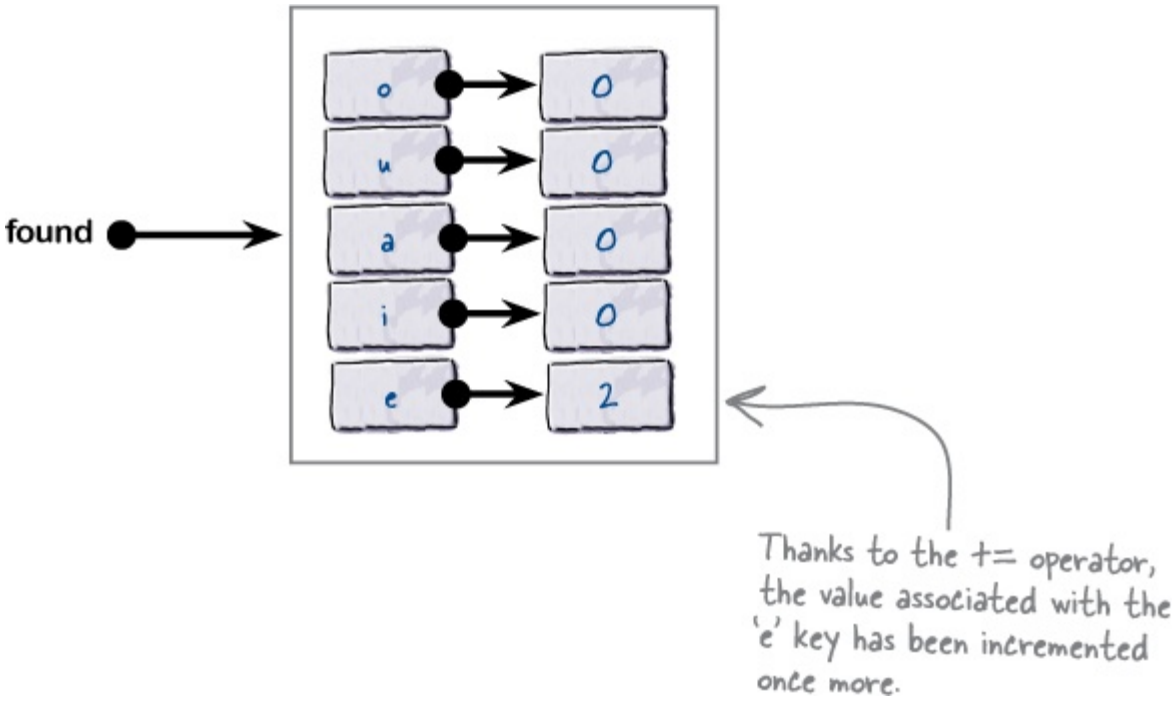


```
>>> found['e'] += 1
>>> found
{'o': 0, 'i': 0, 'a': 0, 'u': 0, 'e': 2}
```

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:



THERE ARE NO DUMB QUESTIONS

Q: **Q: Does Python have ++?**

A: **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.*

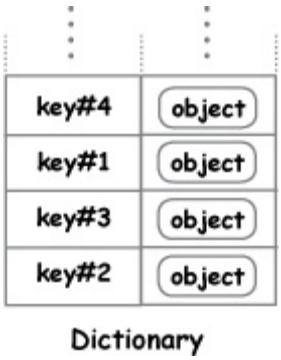
Q: **Q: Is there a handy list of operators?**

A: **A:** *Yes. Head over to https://docs.python.org/3/reference/lexical_analysis.html#operators for a list, and then*

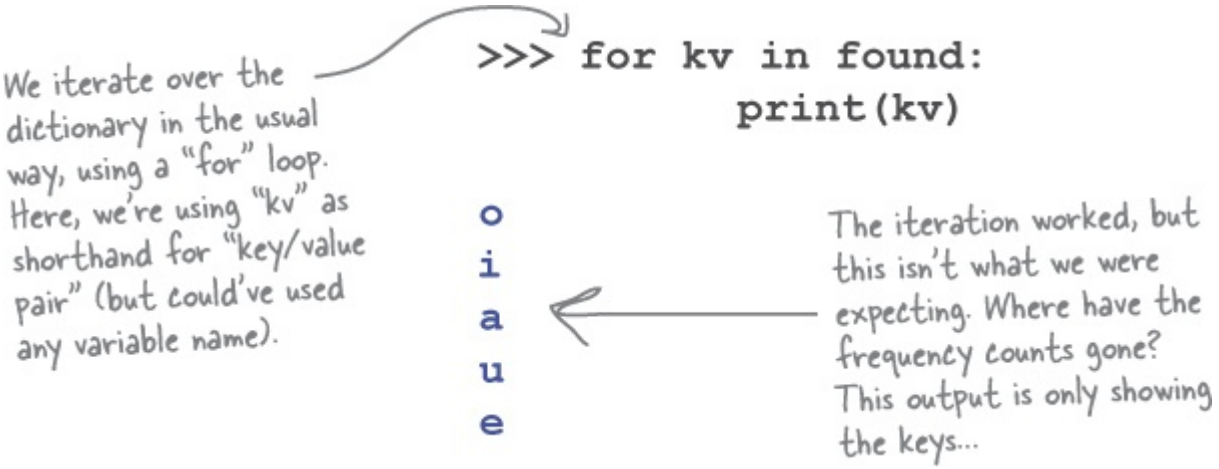
see https://docs.python.org/3/library/stdtypes.html for a detailed explanation of their usage in relation to Python's built-in types.

Iterating Over a Dictionary

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.



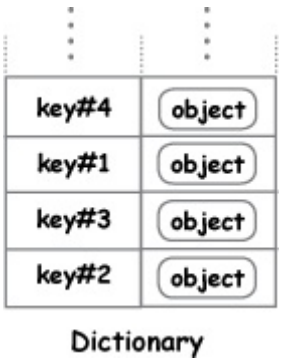
You'd be forgiven for thinking that all we need to do here is use the dictionary with a `for` loop, but doing so produces unexpected results:



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

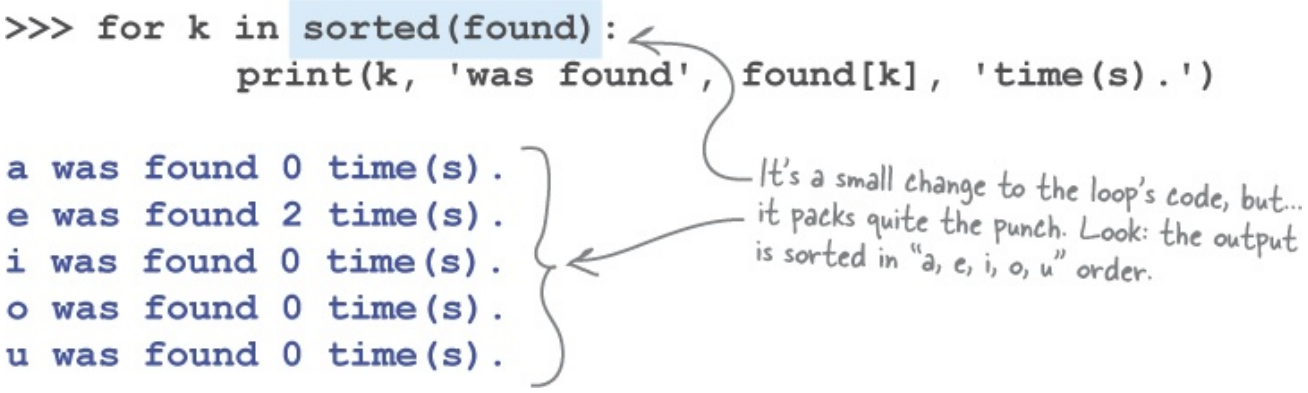
Here's what we know about Python's dictionary data structure so far:

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:



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"

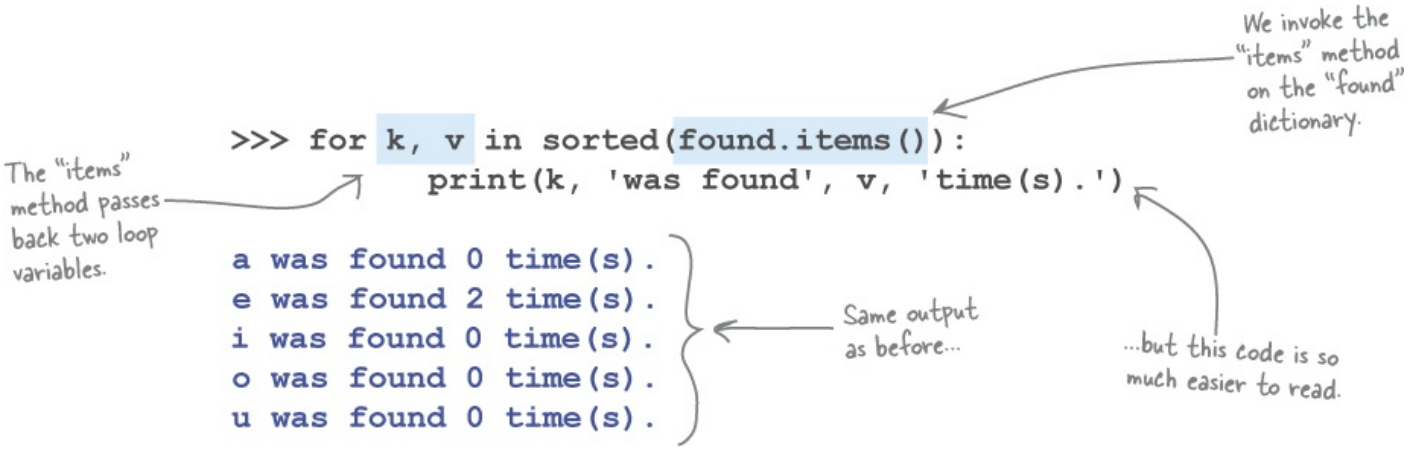
We've seen that it's possible to iterate over the rows of data in a dictionary using this code:

```
>>> 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:



THERE ARE NO DUMB QUESTIONS

Q: **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: **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.*

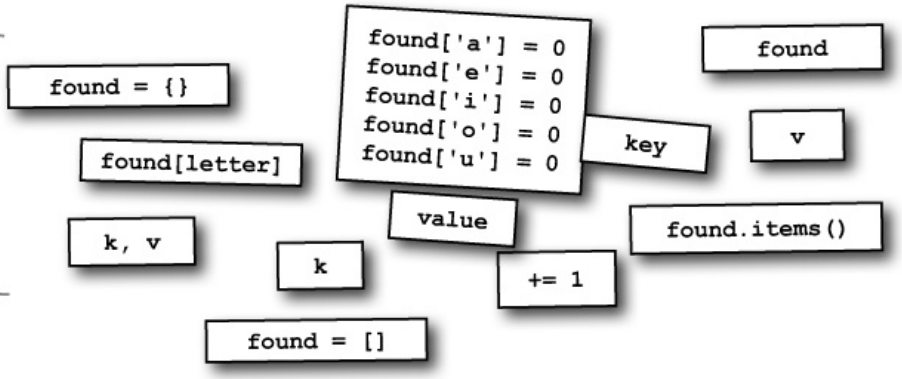
FREQUENCY COUNT MAGNETS

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.

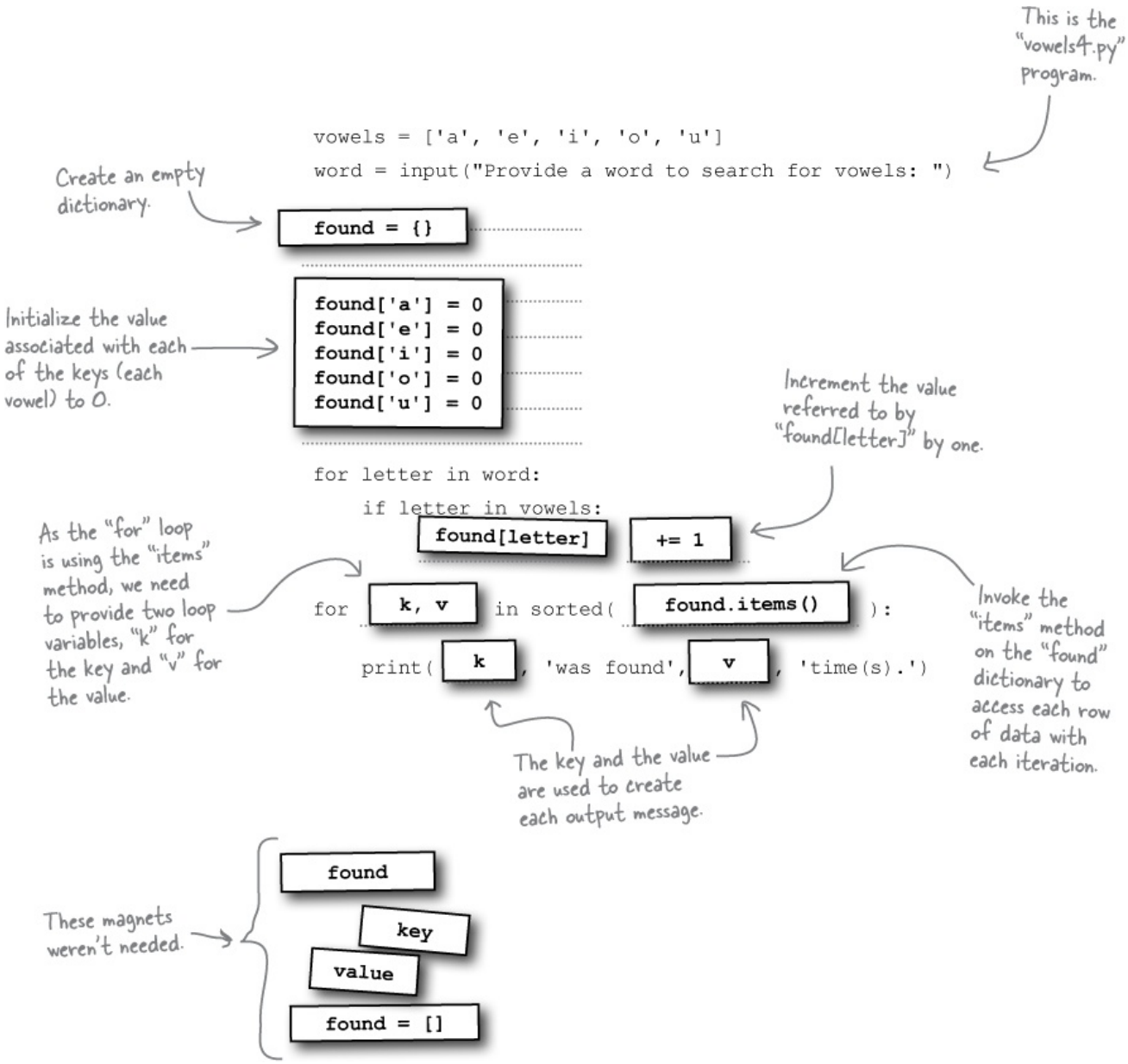


Once you've placed the magnets where you think they should go, bring `vowels3.py` into IDLE's edit window, rename it `vowels4.py`, and then apply your code changes to the new version of this program.

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.



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).
>>>
```

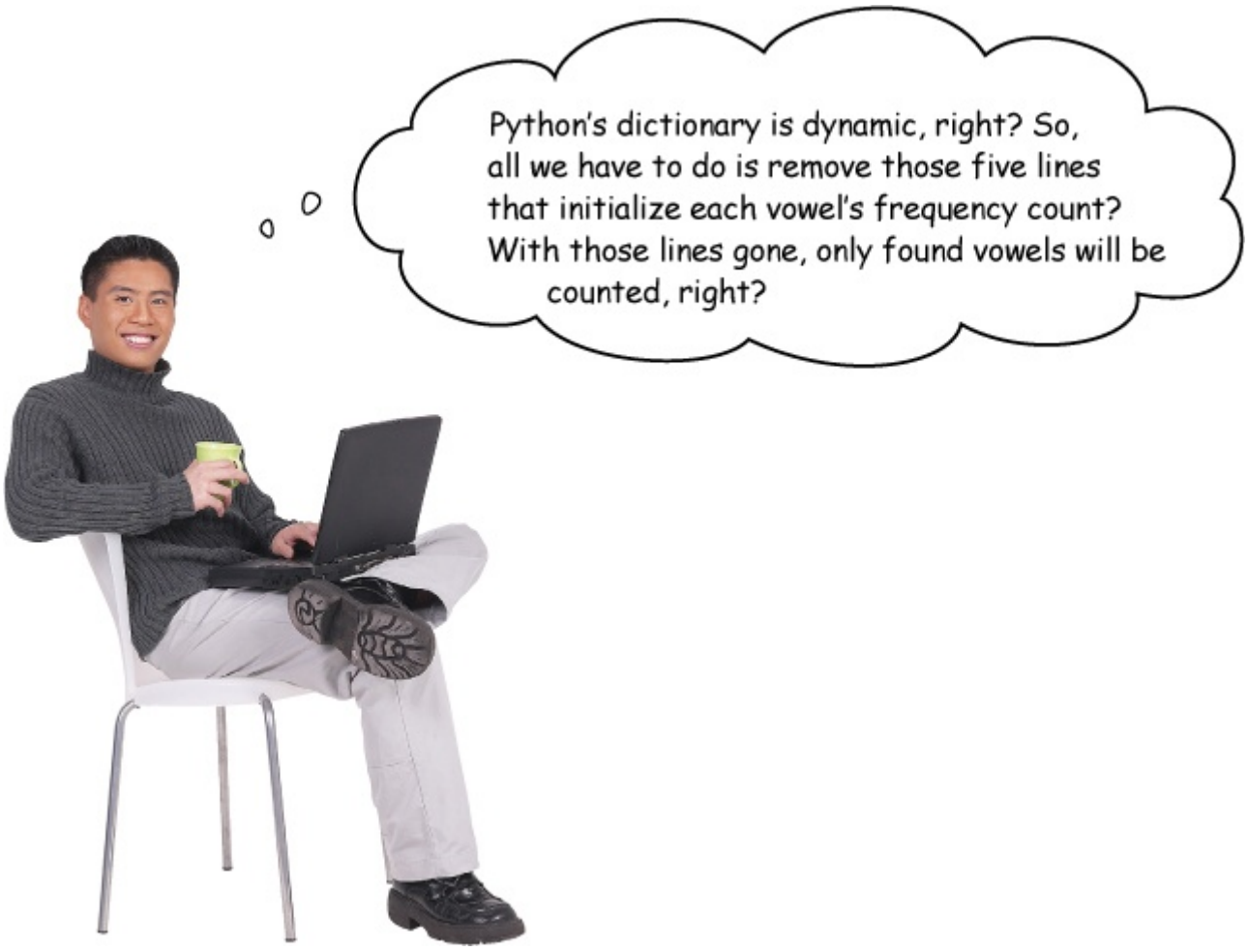
These three "runs" produce the output we expect them to.

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?



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.

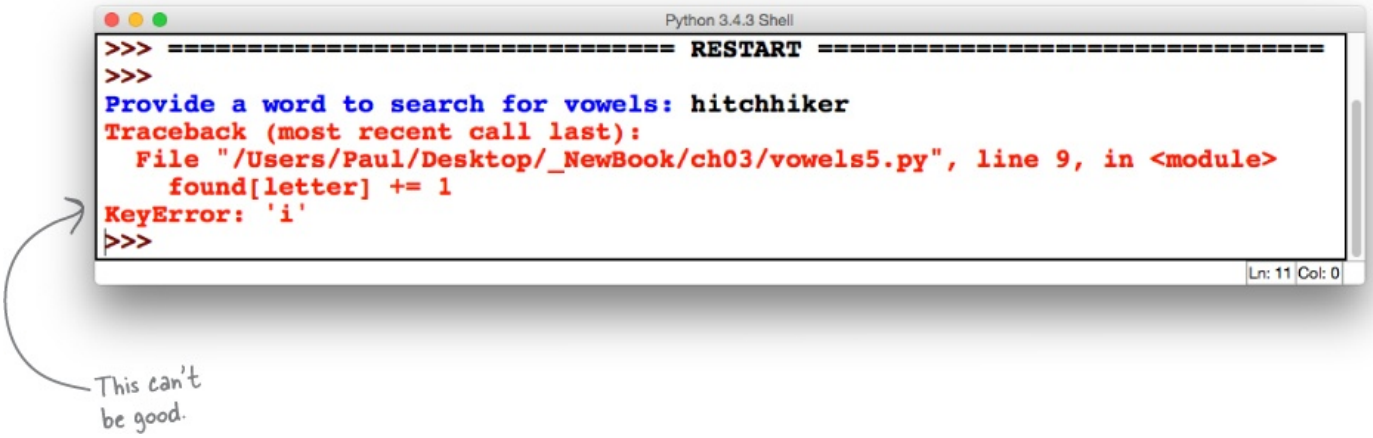
vowels5.py - /Users/Paul/Desktop/_NewBook/ch03/vowels5.py (3.4.3)

```
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:



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

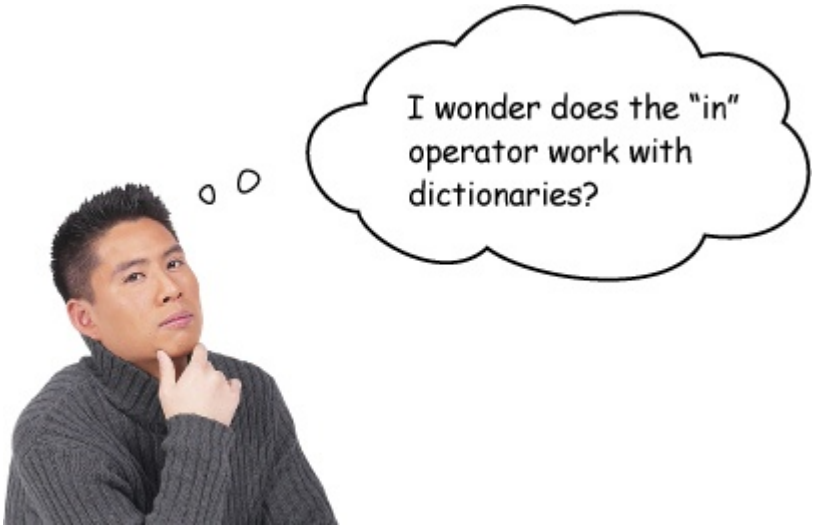
Removing the initialization code has resulted in a runtime error, specifically a `KeyError`, which is raised when you try to access a value associated with a nonexistent key. Because the key can't be found, the value associated with it can't be found either, and you get an error.

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.

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.



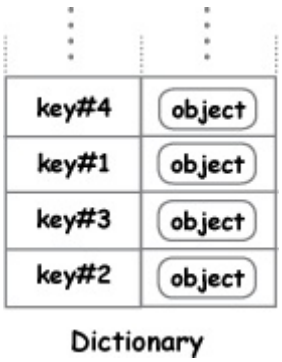
That's a great question.

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.



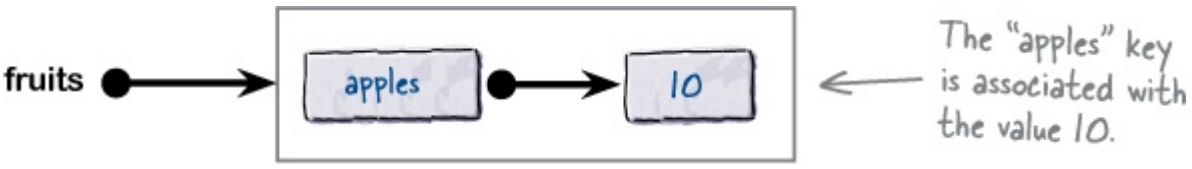
Let's use this fact to avoid that `KeyError` exception, because it can be annoying when your code stops as a result of this error being raised during an attempt to populate a dictionary at runtime.

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:



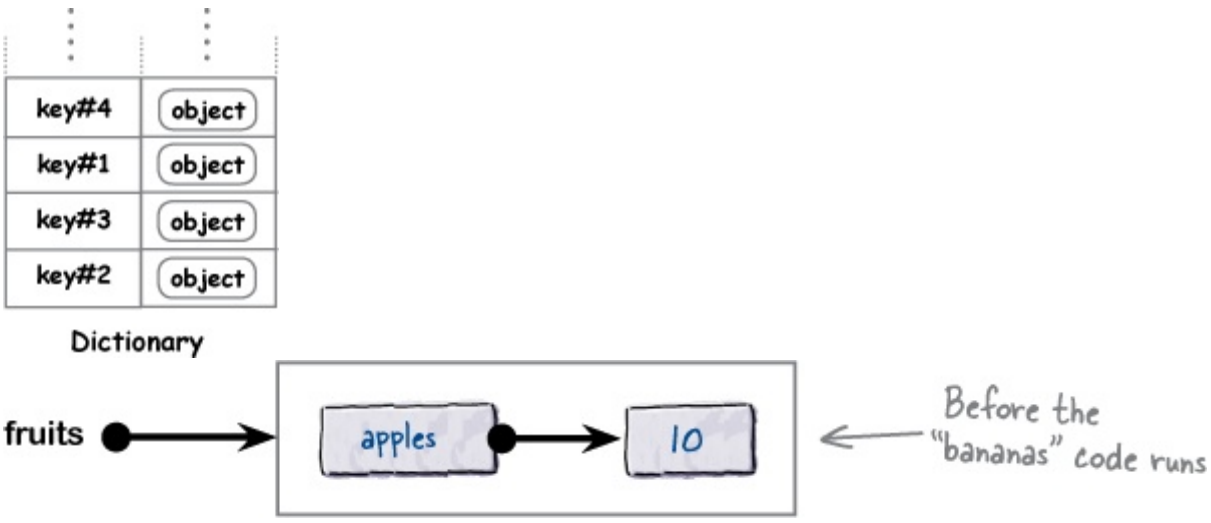
THERE ARE NO DUMB QUESTIONS

Q: **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: **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`.



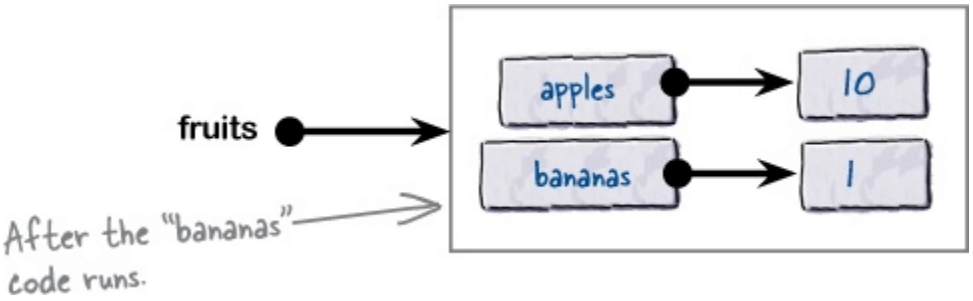
In the code that follows, the `in` operator in conjunction with an `if` statement avoids any slip-ups with `bananas`, which—as wordplays go—is pretty bad (even for us):

```
>>> 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:



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
```

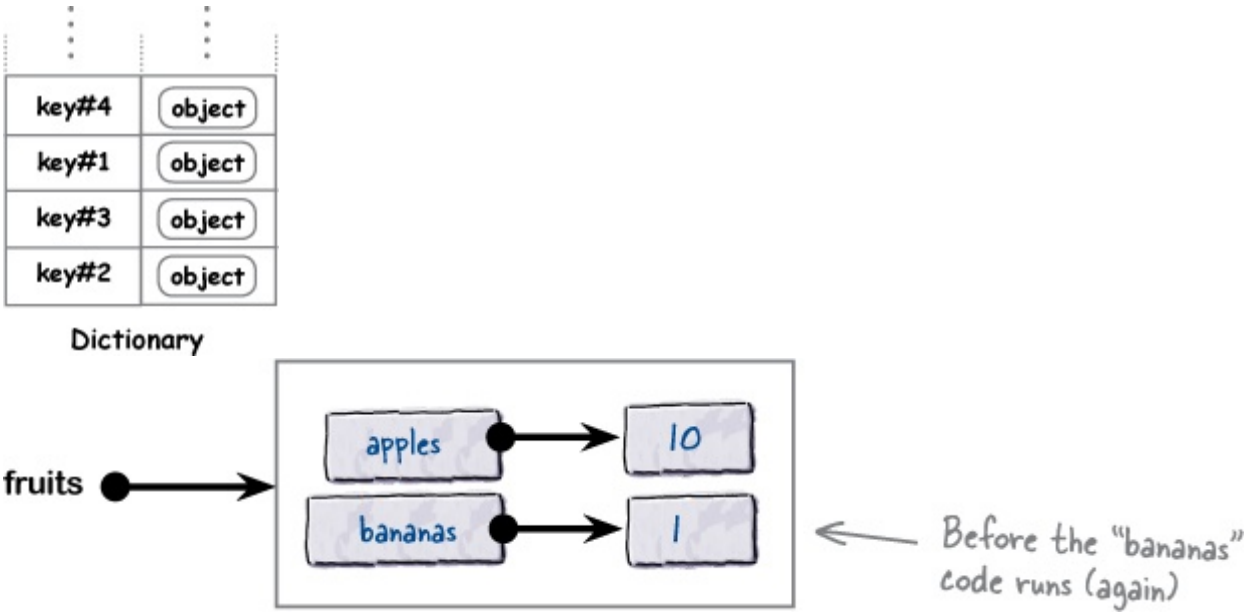
to set **`x`** to either **`10`** or **`20`** depending on whether or not the value of **`y`** is greater than **`3`**. That said, most Python programmers frown on its use, as the equivalent **`if... else...`** statements are considered easier to read.

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.

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:



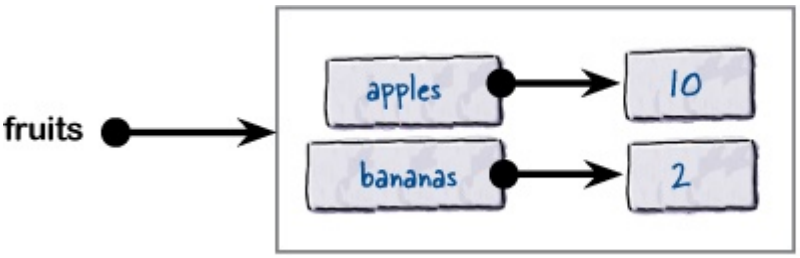
To run this code again, press *Ctrl-P* (on a Mac) or *Alt-P* (on Linux/Windows) to cycle back through your previously entered code statements while at IDLE's `>>>` prompt (as using the up arrow to recall input doesn't work at IDLE's `>>>` prompt). Remember to press Enter *twice* to execute the code once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



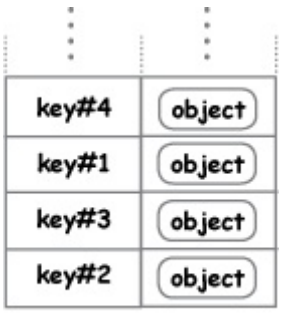
After the "bananas" code runs, the value associated with "bananas" has increased.

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:



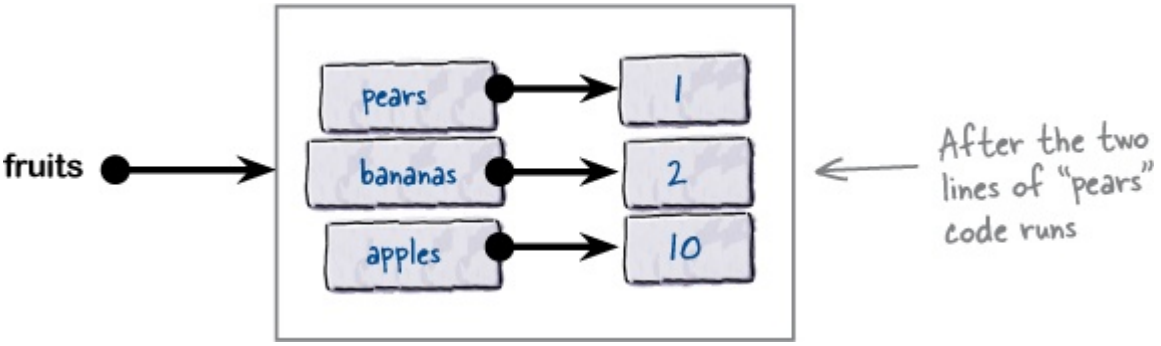
Dictionary

```
>>> 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.

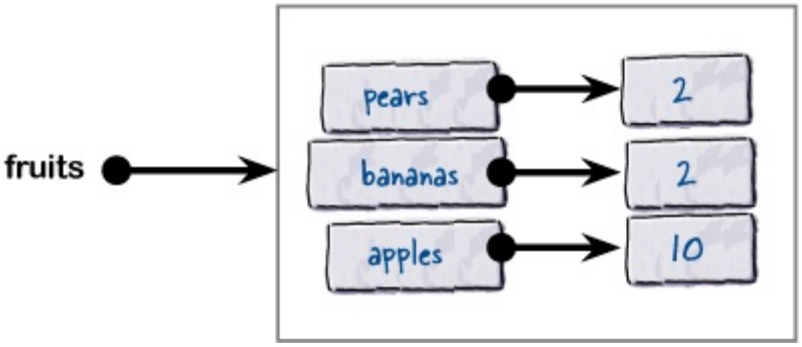
Here's the equivalent of the `pears` code from the top of the page rewritten to use `setdefault`:

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

Initialize (if needed).

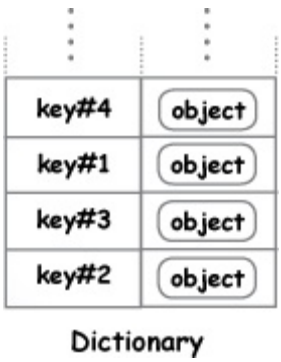
Increment.

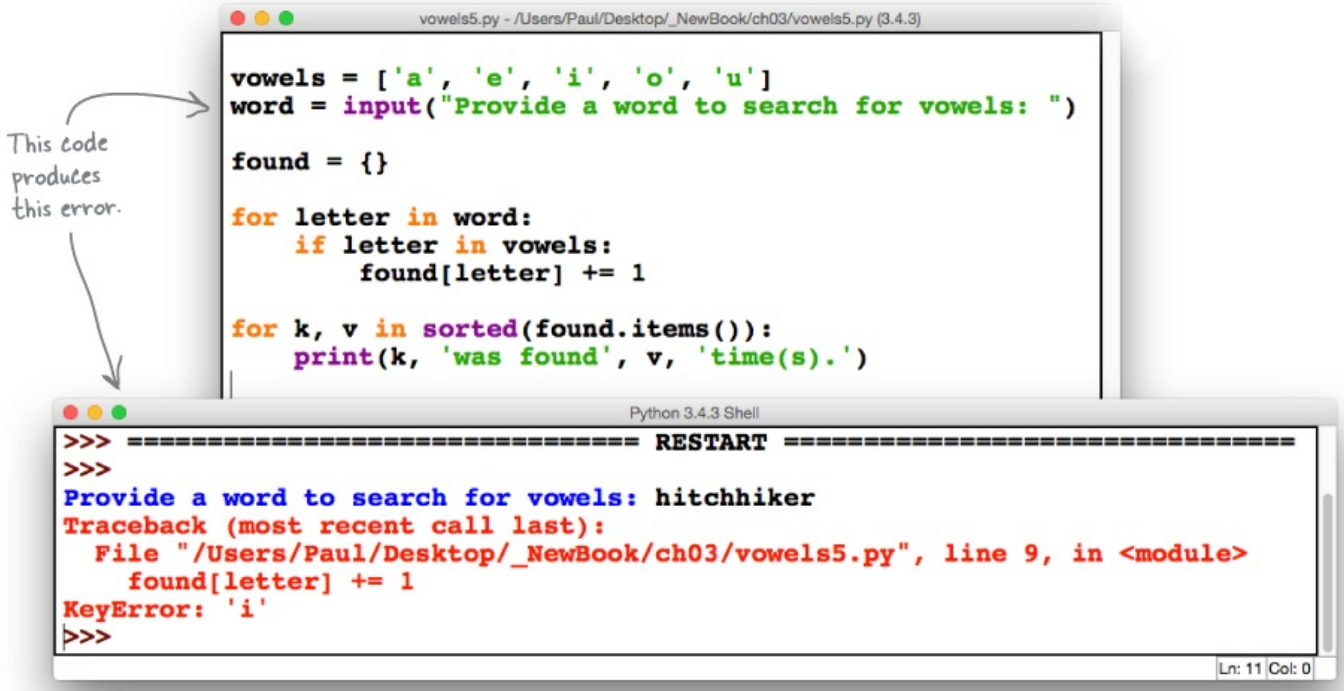
The single call to `setdetfault` has replaced the two-line `if`/`not in` statement, and its usage guarantees that a key is always initialized to a starter value before it's used. Any possibility of a `KeyError` exception is negated. The current state of the `fruits` dictionary is shown here (on the right) 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:





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 s*pecific 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.

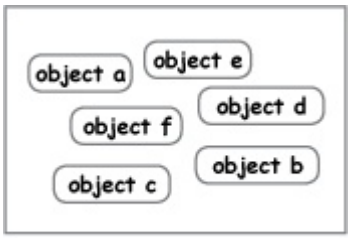
THERE ARE NO DUMB QUESTIONS

Q: **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: **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

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*.



Set

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.

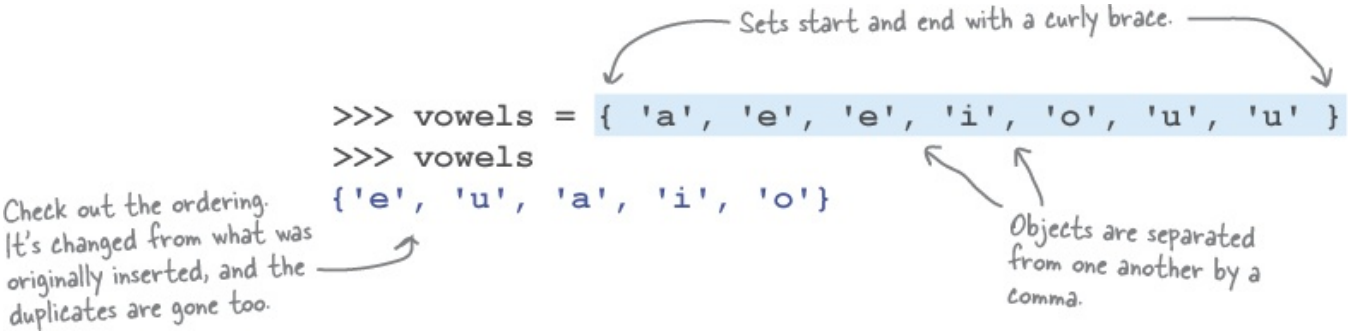
Python's set data structure is optimized for very speedy lookup, which makes using a set much faster than its equivalent list when lookup is the primary

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:



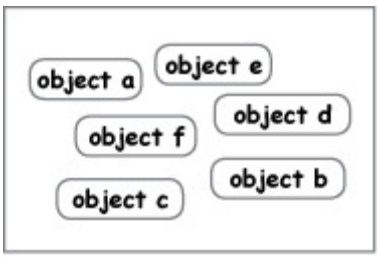
The fact that a set is enclosed in curly braces can often result in your brain mistaking a set for a dictionary, which is *also* enclosed in curly braces. The key difference is the use of the colon character (`:`) in dictionaries to separate keys from values. The colon never appears in a set, only commas.

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

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)
```

We never allow duplicates in the "found" list.

Ln: 11 Col: 0

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'}
```

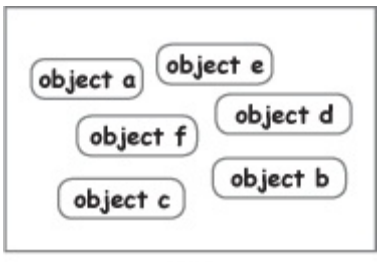
These two lines of code do the same thing: both assign a new set object to a variable.

```
>>> vowels2 = set('aeeiouu')
>>> vowels2
{'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:

Taking Advantage of Set Methods

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.



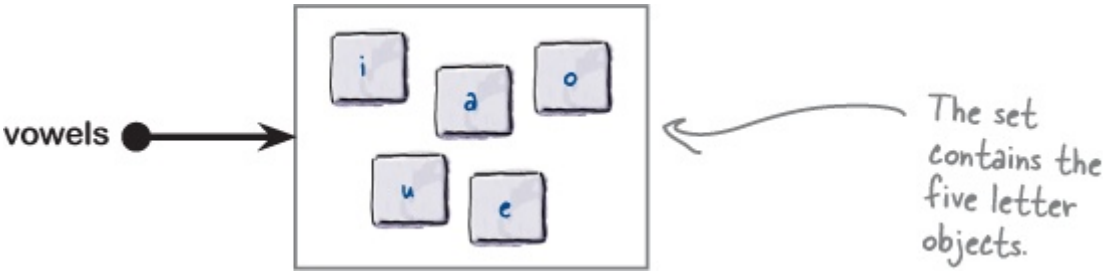
Set

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).

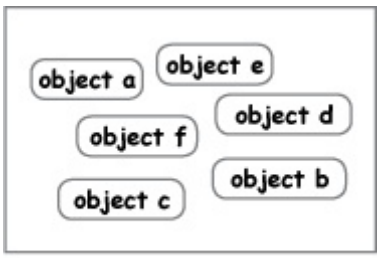
```
>>> u = vowels.union(set(word))
```

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

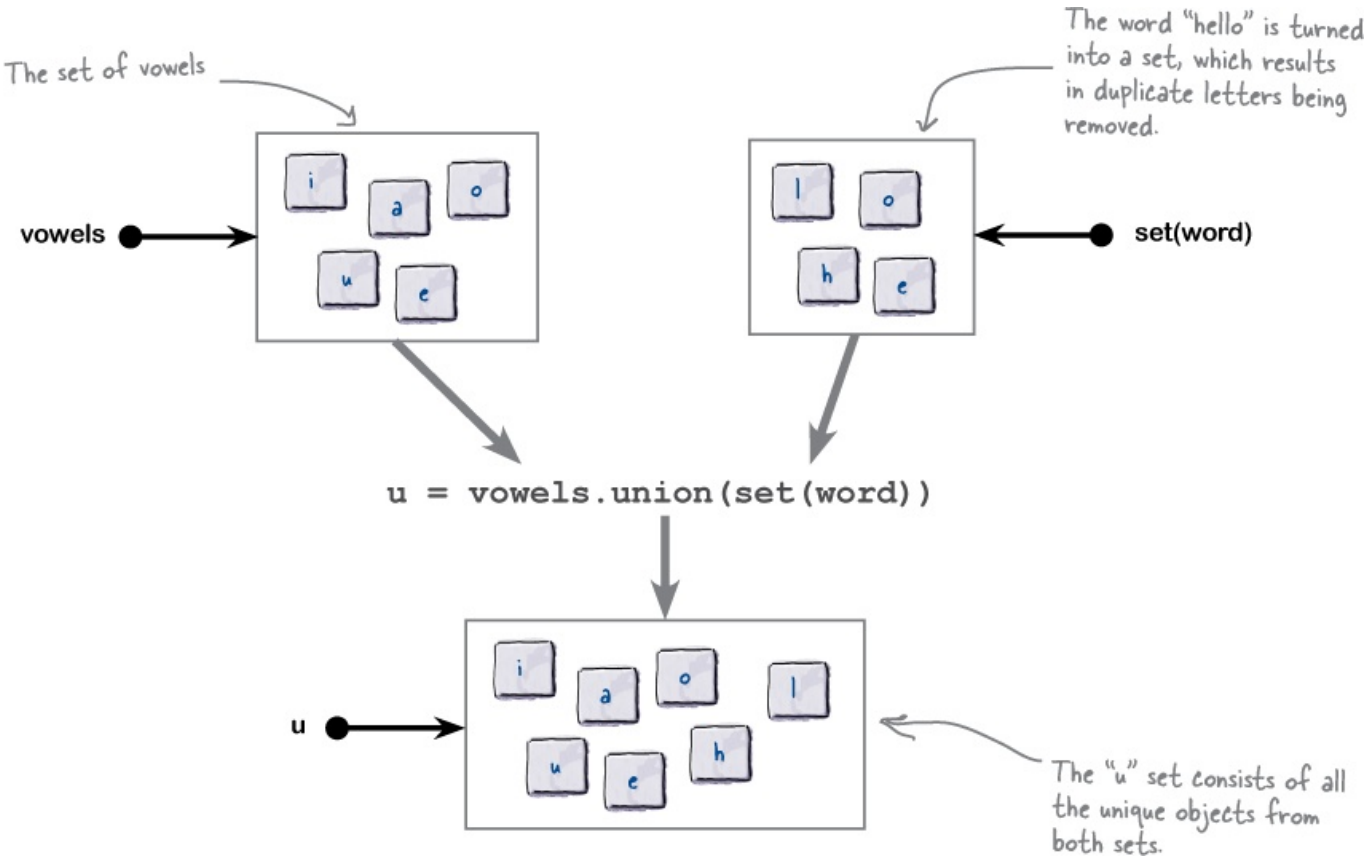
After this call to the `union` method, what do the *`vowels`* and *`u`* sets look like?

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.

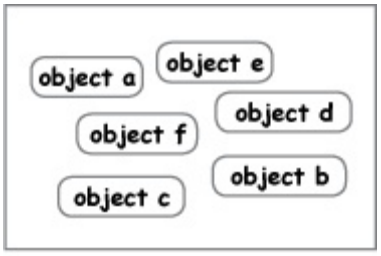
A common requirement (now that we've created the union) is to turn the resulting set into a sorted list. Doing so is trivial, thanks to the `sorted` and `list` functions:

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

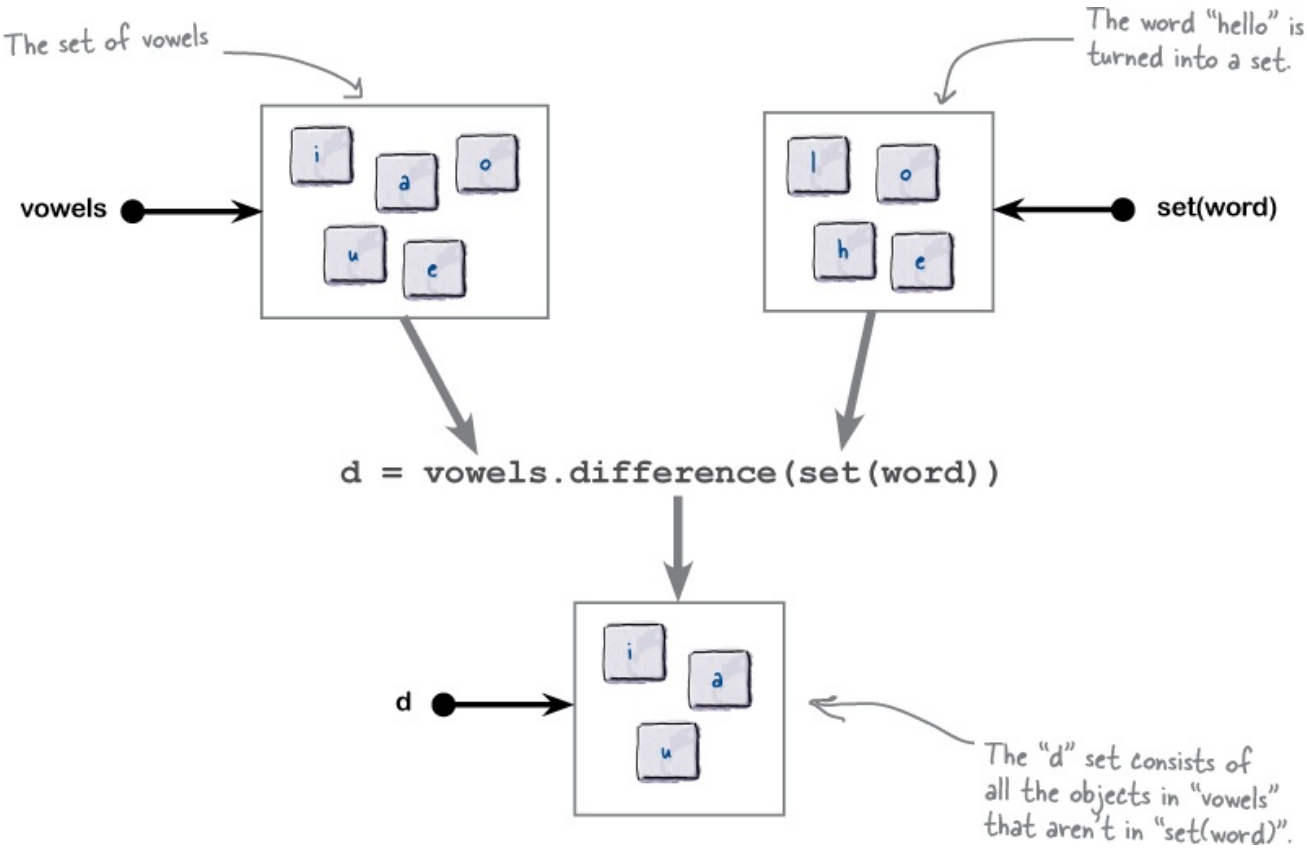
Another set method is `difference`, which, given two sets, can tell you what's in one set but not the other. Let's use `difference` in much the same way as we did with `union` and see what we end up with:




```
>>> 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:

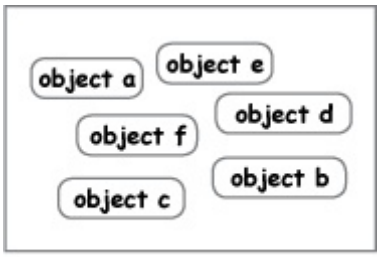


We once again draw your attention to the fact that this outcome has been accomplished *without* using a `for` loop. The `difference` function does all the grunt work here; all we did was state what was required.

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

intersection Reports on Commonality

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.

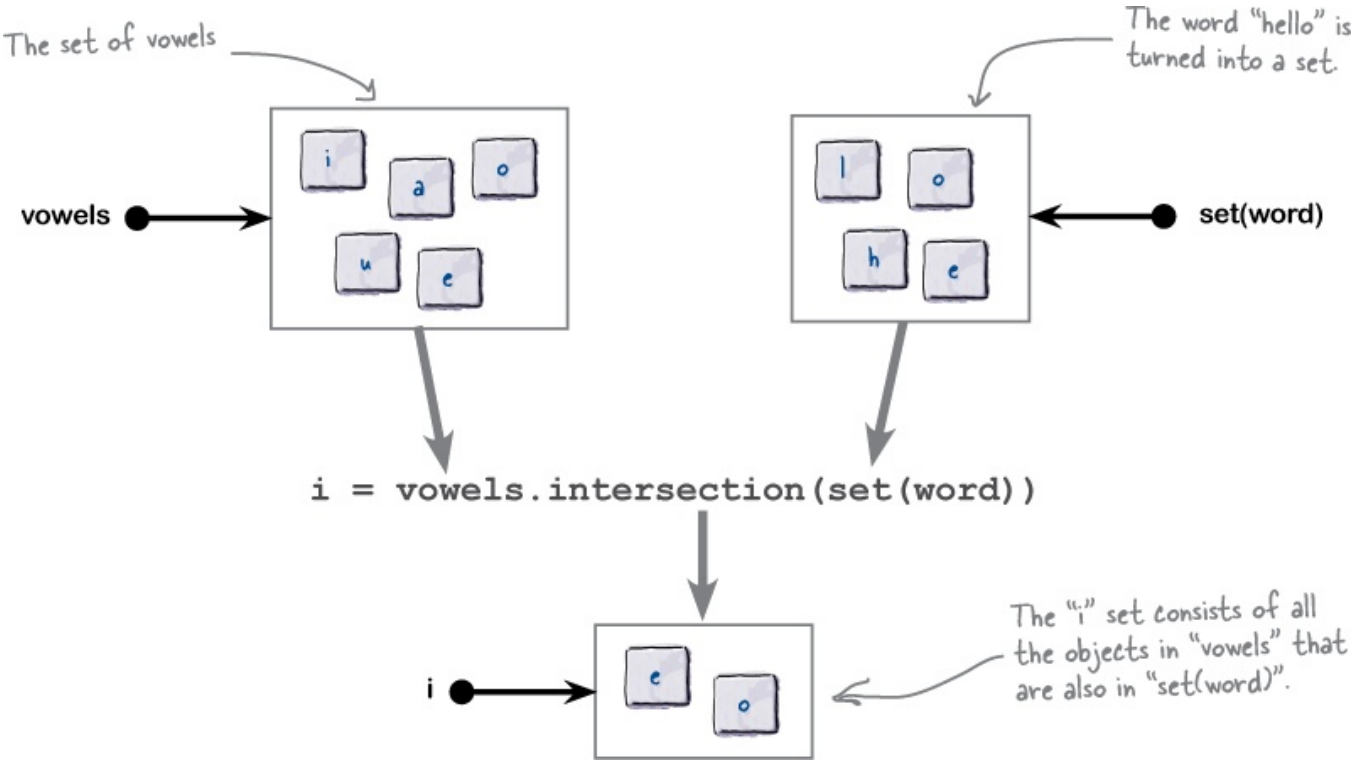


In relation to the requirements that we have with `vowels7.py`, what the `intersection` method does sounds very promising, as we want to know which of the letters in the user's word are vowels.

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)


.....................................................................
..
```

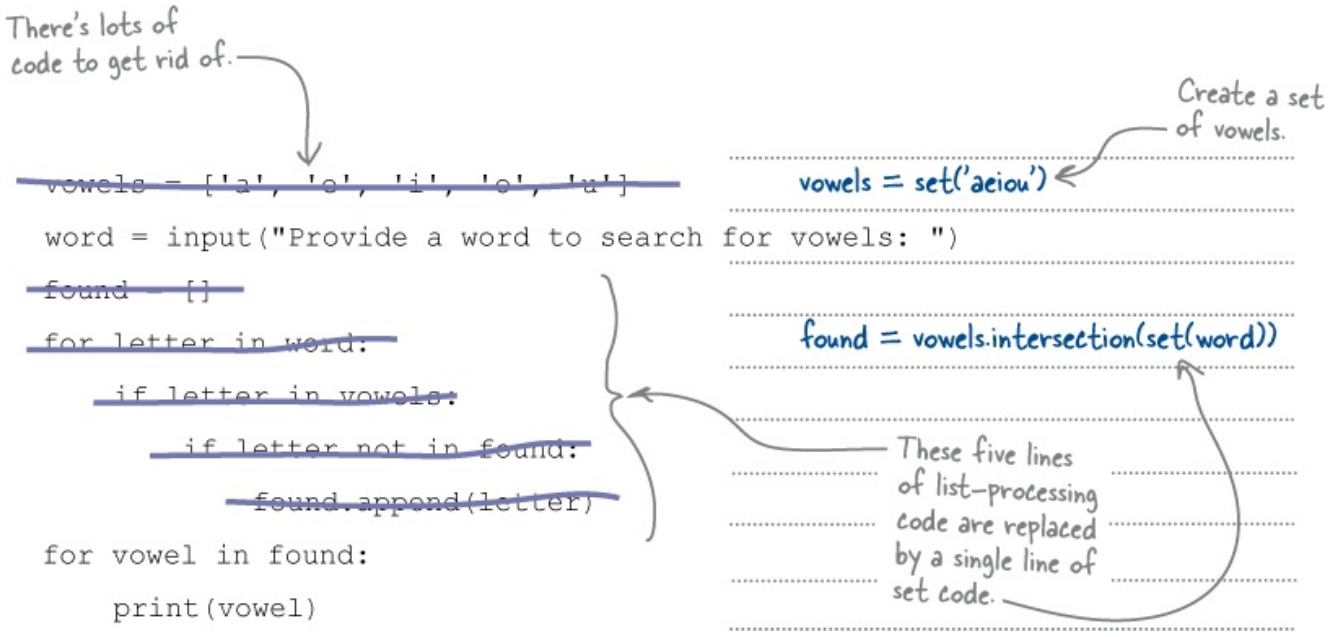
When you're done, be sure to rename your file `vowels7.py`.

SHARPEN YOUR PENCIL SOLUTION

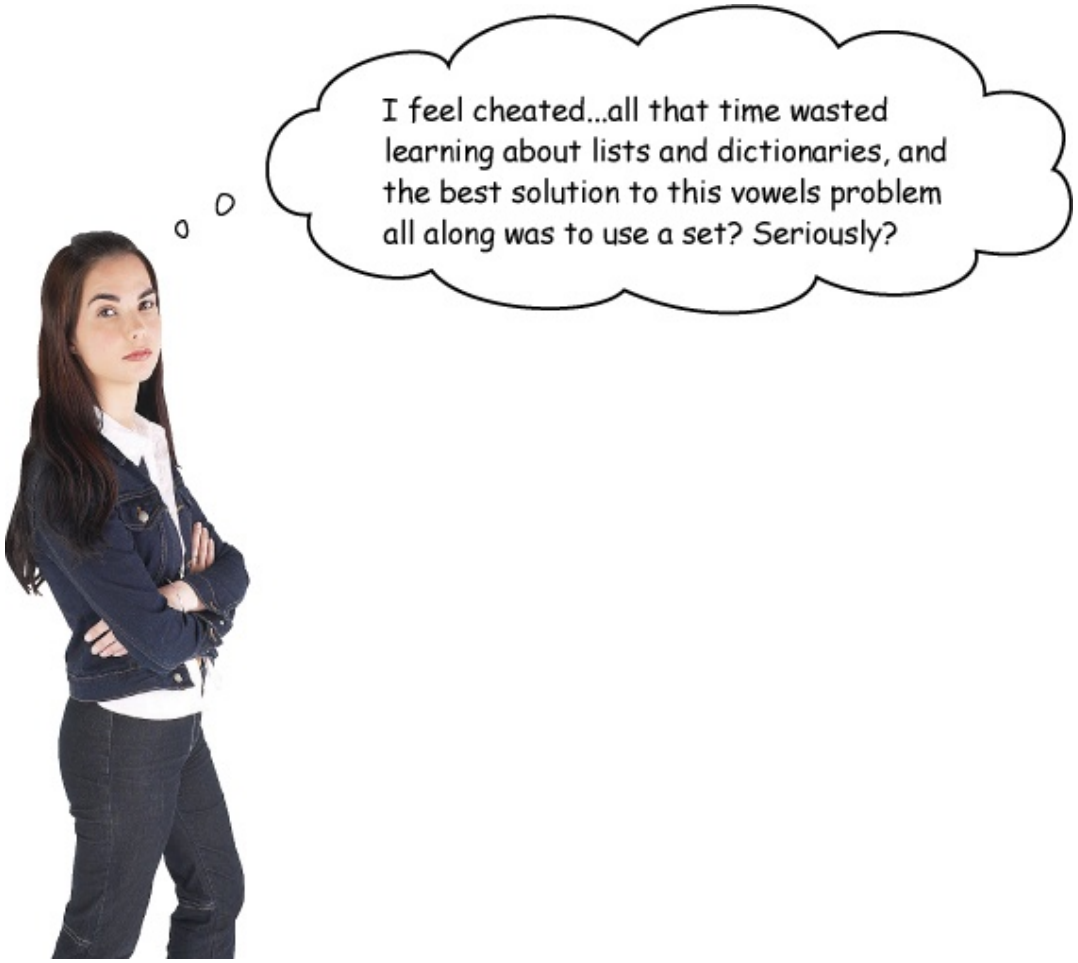
Here is the code to the `vowels3.py` program once more.

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.



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

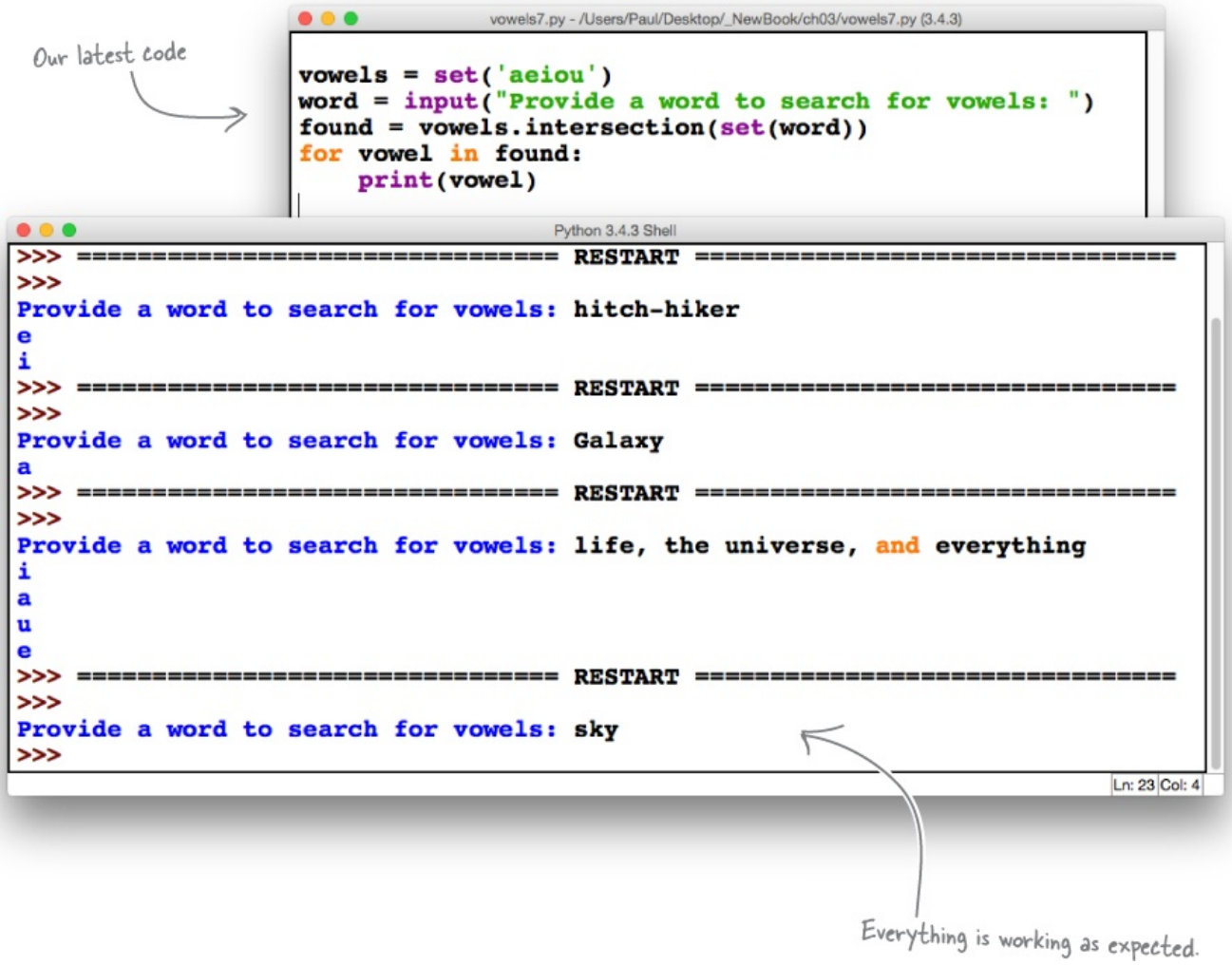


It wasn't a waste of time.

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:



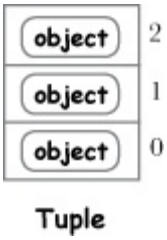
USING A SET WAS THE PERFECT CHOICE HERE...

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

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).

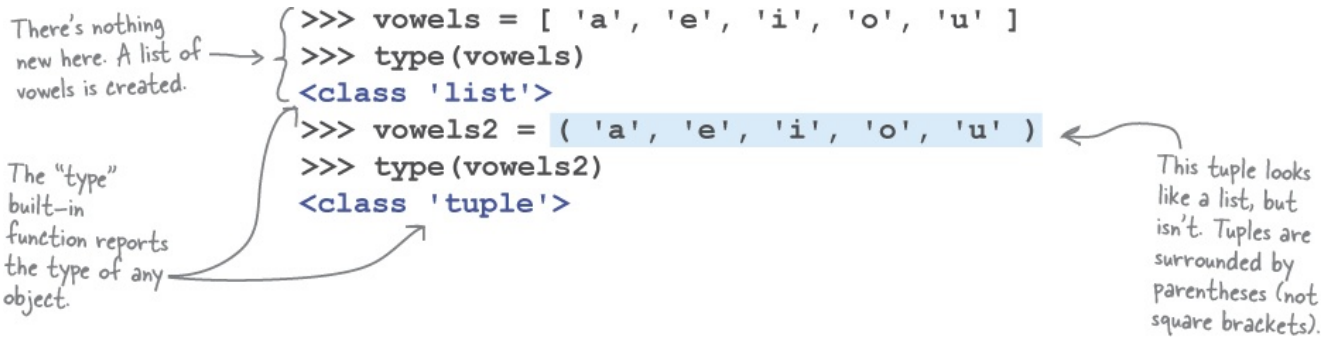
THERE ARE NO DUMB QUESTIONS

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

A: **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:



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:

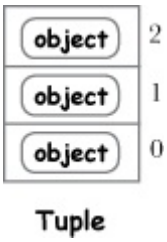
```
>>> vowels
['a', 'e', 'i', 'o', 'u']
>>> vowels2
('a', 'e', 'i', 'o', 'u')
```

The parentheses indicate that this is a tuple.

But what happens if we try to change a tuple?

Tuples Are Immutable

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.

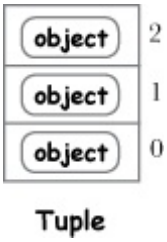
If the data in your structure never changes, put it in a tuple.

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.

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: **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: **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

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

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

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.

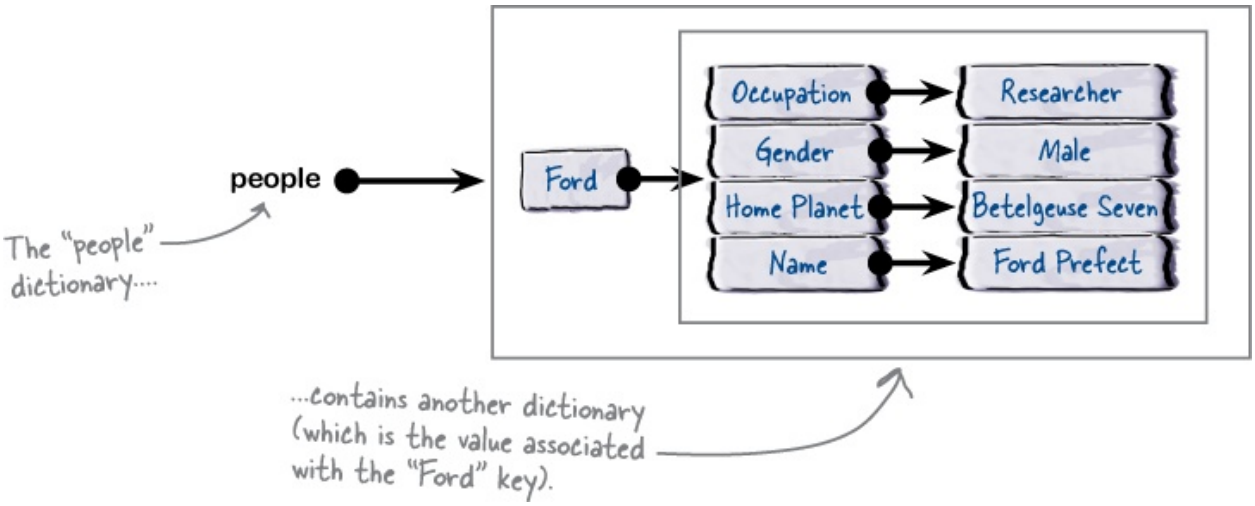
A Dictionary Containing a Dictionary

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:

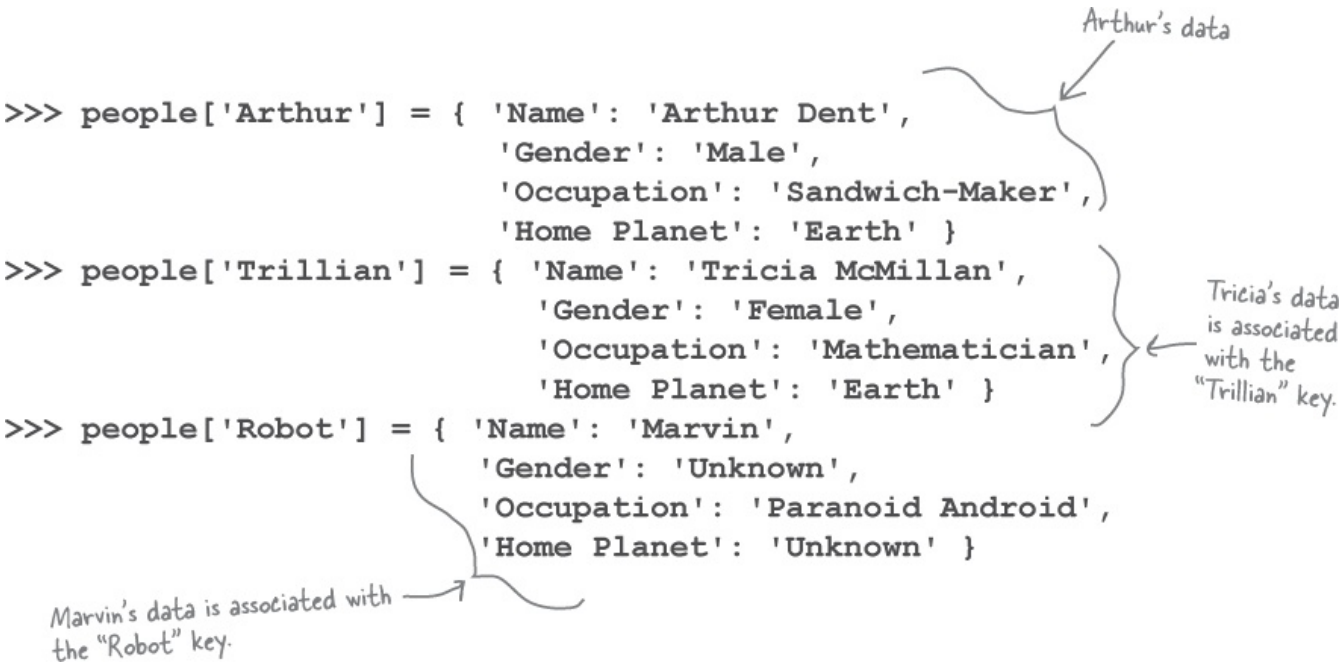
A dictionary embedded in a dictionary—note the extra curly braces.

```
>>> 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:



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

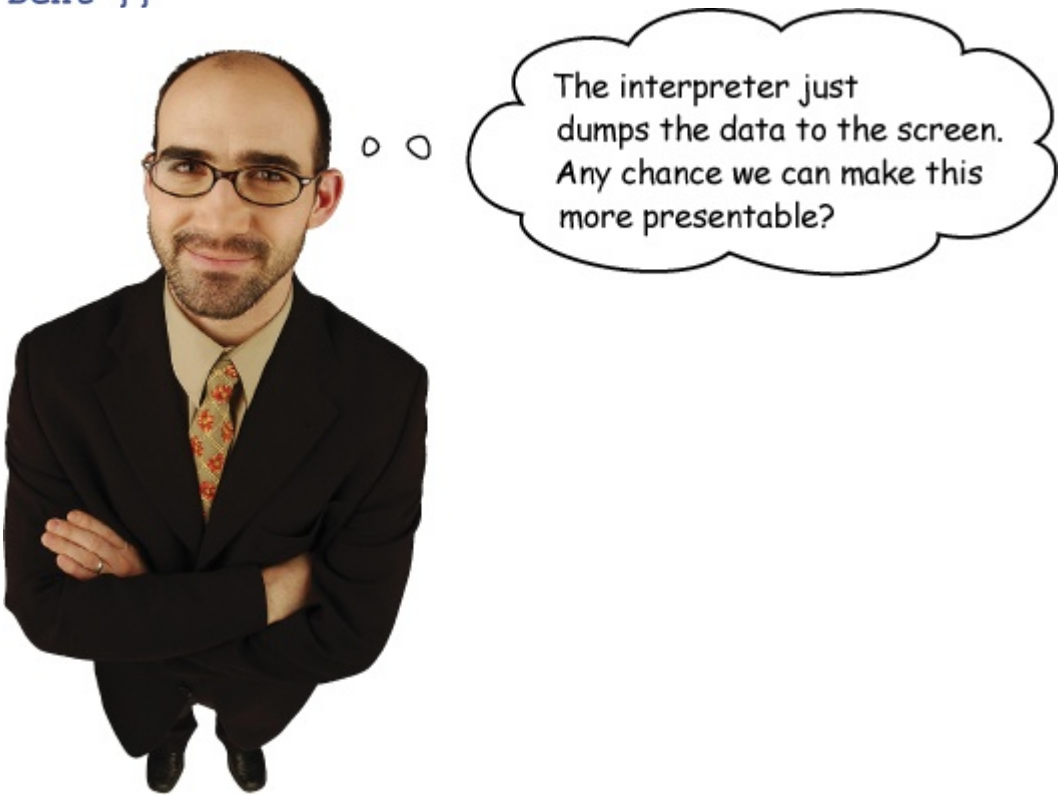
With the `people` dictionary populated with four embedded dictionaries, we can ask the interpreter to display the `people` dictionary at the `>>>` prompt.

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'}}
```



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

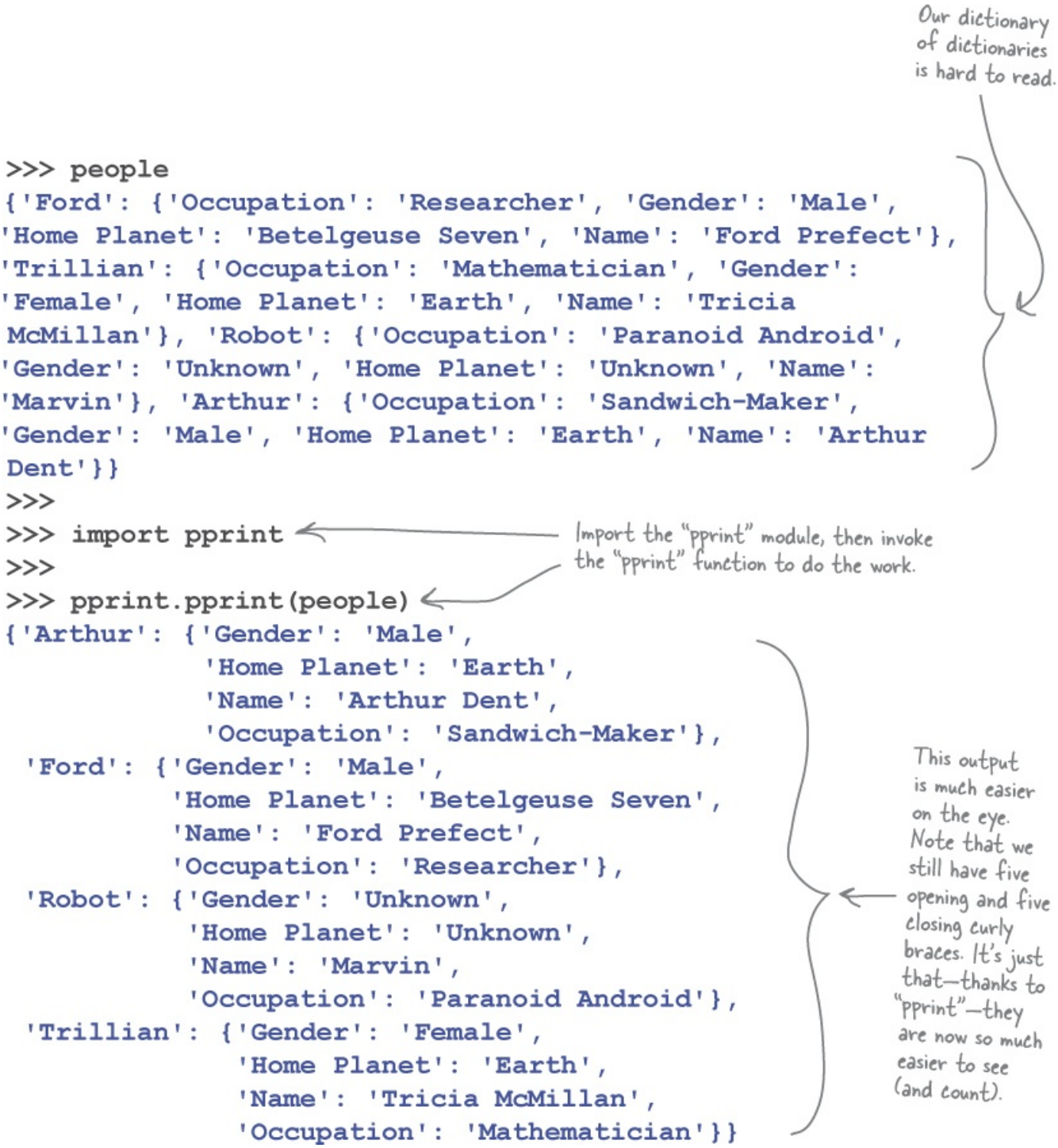
nested `for` loop could process each of the embedded dictionaries, being sure to output something easier to read on screen.

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

Pretty-Printing Complex Data Structures

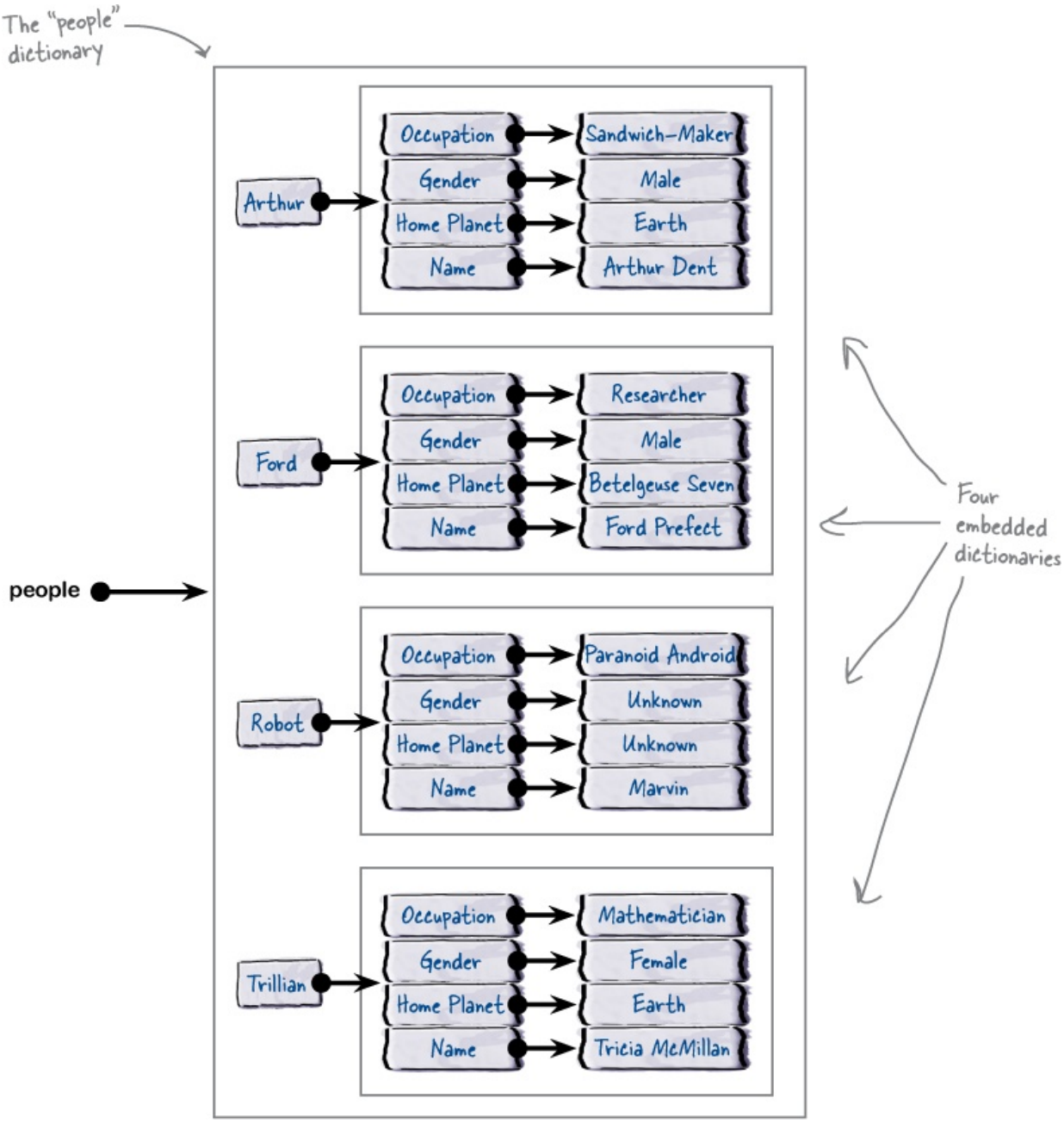
The standard library includes a module called `pprint` that can take any data structure and display it in a easier-to-read format. The name `pprint` is a shorthand for "pretty print."

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:



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

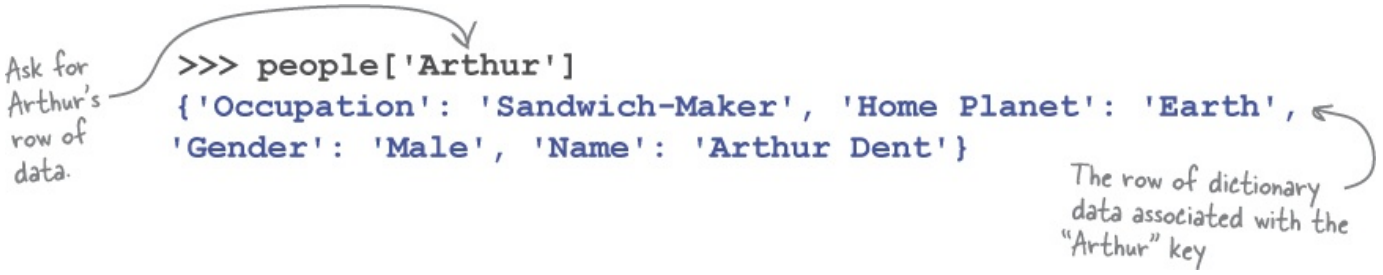
We now have our table of data stored in the `people` dictionary. Let's remind ourselves of what the original table of data looked like:

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

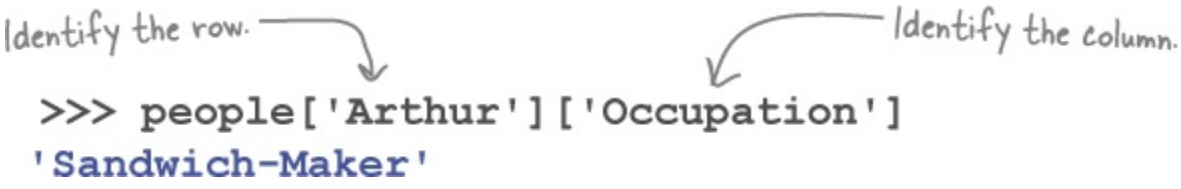
If we were asked to work out what Arthur does, we'd start by looking down the **Name** column for Arthur's name, and then we'd look across the row of data until we arrived at the **Occupation** column, where we'd be able to read "Sandwich-Maker."

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:



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:



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.