O'REILLY®

Third Edition

Think Python

How To Think Like a Computer Scientist

Early Release

RAW & UNEDITED

Allen B. Downey

Think Python

THIRD EDITION

How To Think Like a Computer Scientist

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Allen B. Downey

Beijing • Boston • Farnham • Sebastopol • Tokyo

Think Python

by Allen B. Downey

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (https://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Brian Guerin

Development Editor: Jeff Bleiel

Production Editor: Kristen Brown

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

August 2012: First Edition

December 2015: Second Edition

July 2024: Third Edition

Revision History for the Early Release

- 2023-10-09: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098155438 for release

details.

978-1-098-15543-8

[LSI]

Chapter 1. Lists

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jbleiel@oreilly.com*.

This chapter presents one of Python's most useful built-in types, lists. You will also learn more about objects and what can happen when you have more than one name for the same object.

In the exercises at the end of the chapter, we'll make a word list and use it to search for special words like palindromes and anagrams.

A list is a sequence

Like a string, a **list** is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called **elements**.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets (`[` and `]`). For example, here is a list of two integers.

```
In [1]: numbers = [42, 123]
```

And here's a list of three strings.

```
In [2]: cheeses = ['Cheddar', 'Edam', 'Gouda']
```

The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and even another list.

```
In [3]: t = ['spam', 2.0, 5, [10, 20]]
```

A list within another list is **nested**.

A list that contains no elements is called an empty list; you can create one with empty brackets, `[]`.

```
In [4]: empty = []
```

The `len` function returns the length of a list.

```
In [5]: len(cheeses)
```

```
Out[5]: 3
```

The length of an empty list is `0`.

The following figure shows the state diagram for `cheeses`, `numbers` and `empty`.

Lists are represented by boxes with the word “list” outside and the numbered elements of the list inside.

Lists are mutable

To read an element of a list, we can use the bracket operator. The index of the first element is `0`.

```
In [6]: cheeses[0]
```

```
Out[6]: 'Cheddar'
```

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
In [7]: numbers[1] = 17
        numbers


Out[7]: [42, 17]
```

The second element of `numbers`, which used to be `123`, is now `17`.

List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an `IndexError`.
- If an index has a negative value, it counts backward from the end of the list.

The `in` operator works on lists – it checks whether a given element appears anywhere in the list.

```
In [8]: 'Edam' in cheeses


Out[8]: True


In [9]: 'Wensleydale' in cheeses


Out[9]: False
```

Although a list can contain another list, the nested list still counts as a single element – so in the following list, there are only four elements.

```
In [10]: t = ['spam', 2.0, 5, [10, 20]]
         len(t)
```

```
Out[10]: 4
```

And `10` is not considered to be an element of `t` because it is an element of a nested list, not `t`.

```
In [11]: 10 in t
```

```
Out[11]: False
```

List slices

The slice operator works on lists the same way it works on strings. The following example selects the second and third elements from a list of four letters.

```
In [12]: letters = ['a', 'b', 'c', 'd']
         letters[1:3]
```

```
Out[12]: ['b', 'c']
```

If you omit the first index, the slice starts at the beginning.

```
In [13]: letters[:2]
```

```
Out[13]: ['a', 'b']
```

If you omit the second, the slice goes to the end.

```
In [14]: letters[2:]
```

```
Out[14]: ['c', 'd']
```

So if you omit both, the slice is a copy of the whole list.

```
In [15]: letters[:]
```

```
Out[15]: ['a', 'b', 'c', 'd']
```

Another way to copy a list is to use the `list` function.

```
In [16]: list(letters)
```

```
Out[16]: ['a', 'b', 'c', 'd']
```

Because `list` is the name of a built-in function, you should avoid using it as a variable name.

List operations

The `+` operator concatenates lists.

```
In [17]: t1 = [1, 2]
         t2 = [3, 4]
         t1 + t2
```

```
Out[17]: [1, 2, 3, 4]
```

The `*` operator repeats a list a given number of times.

```
In [18]: ['spam'] * 4
```

```
Out[18]: ['spam', 'spam', 'spam', 'spam']
```

No other mathematical operators work with lists, but the built-in function `sum` adds up the elements.

```
In [19]: sum(t1)
```

```
Out[19]: 3
```

And `min` and `max` find the smallest and largest elements.

```
In [20]: min(t1)
```

```
Out[20]: 1
```

```
In [21]: max(t2)
```

```
Out[21]: 4
```

These functions only work if the elements of the list are numbers.

List methods

Python provides methods that operate on lists. For example, `append` adds a new element to the end of a list:

```
In [22]: letters.append('e')
         letters
```

```
Out[22]: ['a', 'b', 'c', 'd', 'e']
```

`extend` takes a list as an argument and appends all of the elements:

```
In [23]: letters.extend(['f', 'g'])
         letters
```

```
Out[23]: ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

There are two ways to remove elements from a list. If you know the index of the element you want, you can use `pop`.

```
In [24]: t = ['a', 'b', 'c']
         t.pop(1)
```

```
Out[24]: 'b'
```

The return value is the element that was removed. And we can confirm that the list has been modified.

```
In [25]: t
```

```
Out[25]: ['a', 'c']
```

If you know the element you want to remove (but not the index), you can use `remove`:

```
In [26]: t = ['a', 'b', 'c']
         t.remove('b')
```

The return value from `remove` is `None`. But we can confirm that the list has

been modified.

```
In [27]: t
```

```
Out[27]: ['a', 'c']
```

If the element you ask for is not in the list, that's a ValueError.

```
In [28]: t.remove('d')
```

```
Out[28]: ValueError: list.remove(x): x not in list
```

Lists and strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use the `list` function.

```
In [29]: s = 'spam'
         t = list(s)
         t
```

```
Out[29]: ['s', 'p', 'a', 'm']
```

The `list` function breaks a string into individual letters. If you want to break a string into words, you can use the `split` method:

```
In [30]: s = 'pining for the fjords'
         t = s.split()
         t
```

```
Out[30]: ['pining', 'for', 'the', 'fjords']
```

An optional argument called a **delimiter** specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter.

```
In [31]: s = 'ex-parrot'
         t = s.split('-')
         t
```

```
Out[31]: ['ex', 'parrot']
```

If you have a list of strings, you can concatenate them into a single string using `join`. `join` is a string method, so you have to invoke it on the delimiter and pass the list as an argument.

```
In [32]: delimiter = ' '
         t = ['pining', 'for', 'the', 'fjords']
         s = delimiter.join(t)
         s
```

```
Out[32]: 'pining for the fjords'
```

In this case the delimiter is a space character, so `join` puts a space between words. To join strings without spaces, you can use the empty string, `''`, as a delimiter.

Looping through a list

You can use a `for` statement to loop through the elements of a list.

```
In [33]: for cheese in cheeses:
             print(cheese)
```

```
Out[33]: Cheddar
```

```
Edam
Gouda
```

For example, after using `split` to make a list of words, we can use `for` to loop through them.

```
In [34]: s = 'pining for the fjords'

         for word in s.split():
             print(word)


Out[34]: pining
         for
         the
         fjords
```

A `for` loop over an empty list never runs the body.

```
In [35]: for x in []:
             print('This never happens.')
```

Sorting lists

Python provides a built-in function called `sorted` that sorts the elements of a list.

```
In [36]: scramble = ['c', 'a', 'b']
         sorted(scramble)


Out[36]: ['a', 'b', 'c']
```

The original list is unchanged.

```
In [37]: scramble
```

```
Out[37]: ['c', 'a', 'b']
```

`sorted` works with any kind of sequence, not just lists. So we can sort the letters in a string like this.

```
In [38]: sorted('letters')
```

```
Out[38]: ['e', 'e', 'l', 'r', 's', 't', 't']
```

The result it a list. To convert the list to a string, we can use `join`.

```
In [39]: ''.join(sorted('letters'))
```

```
Out[39]: 'eelrstt'
```

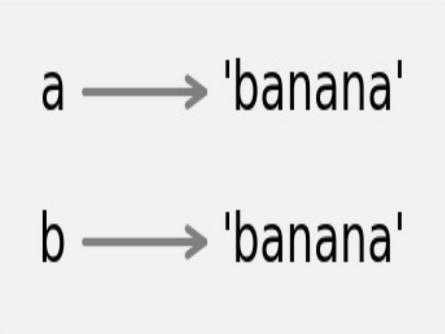
With an empty string as the delimiter, the elements of the list are joined with nothing between them.

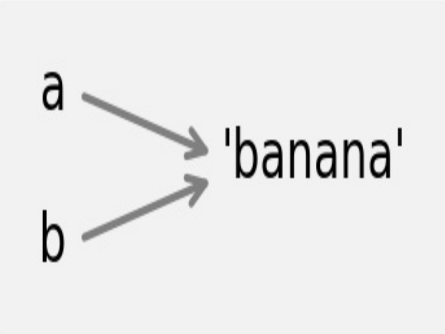
Objects and values

If we run these assignment statements:

```
In [40]: a = 'banana'
         b = 'banana'
```

We know that `a` and `b` both refer to a string, but we don't know whether they refer to the *same* string. There are two possible states, shown in the following figure.





In the diagram on the left, `a` and `b` refer to two different objects that have the same value. In the diagram on the right, they refer to the same object. To check whether two variables refer to the same object, you can use the `is` operator.

```
In [41]: a = 'banana'
         b = 'banana'
         a is b
```

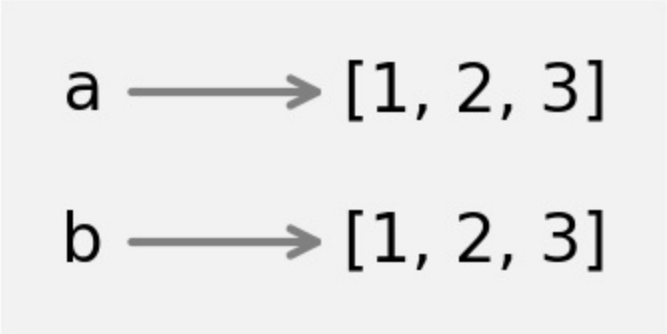
```
Out[41]: True
```

In this example, Python only created one string object, and both `a` and `b` refer to it. But when you create two lists, you get two objects.

```
In [42]: a = [1, 2, 3]
         b = [1, 2, 3]
         a is b
```

```
Out[42]: False
```

So the state diagram looks like this.



In this case we would say that the two lists are **equivalent**, because they have the same elements, but not **identical**, because they are not the same object. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

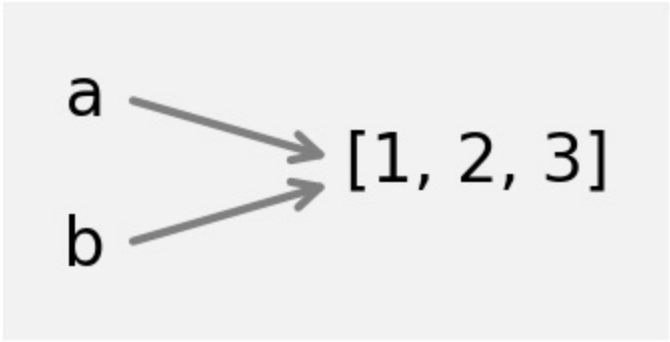
Aliasing

If `a` refers to an object and you assign `b = a`, then both variables refer to the same object.

```
In [43]: a = [1, 2, 3]
         b = a
         b is a


Out[43]: True
```

So the state diagram looks like this.



The association of a variable with an object is called a **reference**. In this example, there are two references to the same object.

An object with more than one reference has more than one name, so we say the object is **aliased**. If the aliased object is mutable, changes made with one name affect the other. In this example, if we change the object `b` refers to, we are also changing the object `a` refers to.

```
In [44]: b[0] = 5
         a


Out[44]: [5, 2, 3]
```

So we would say that `a` "sees" this change. Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.

For immutable objects like strings, aliasing is not as much of a problem. In this example:

```
In [45]: a = 'banana'
         b = 'banana'
```

It almost never makes a difference whether `a` and `b` refer to the same string or not.

List arguments

When you pass a list to a function, the function gets a reference to the list. If the function modifies the list, the caller sees the change. For example, `pop_first` uses the list method `pop` to remove the first element from a list.

```
In [46]: def pop_first(lst):
             return lst.pop(0)
```

We can use it like this.

```
In [47]: letters = ['a', 'b', 'c']
         pop_first(letters)
```

```
Out[47]: 'a'
```

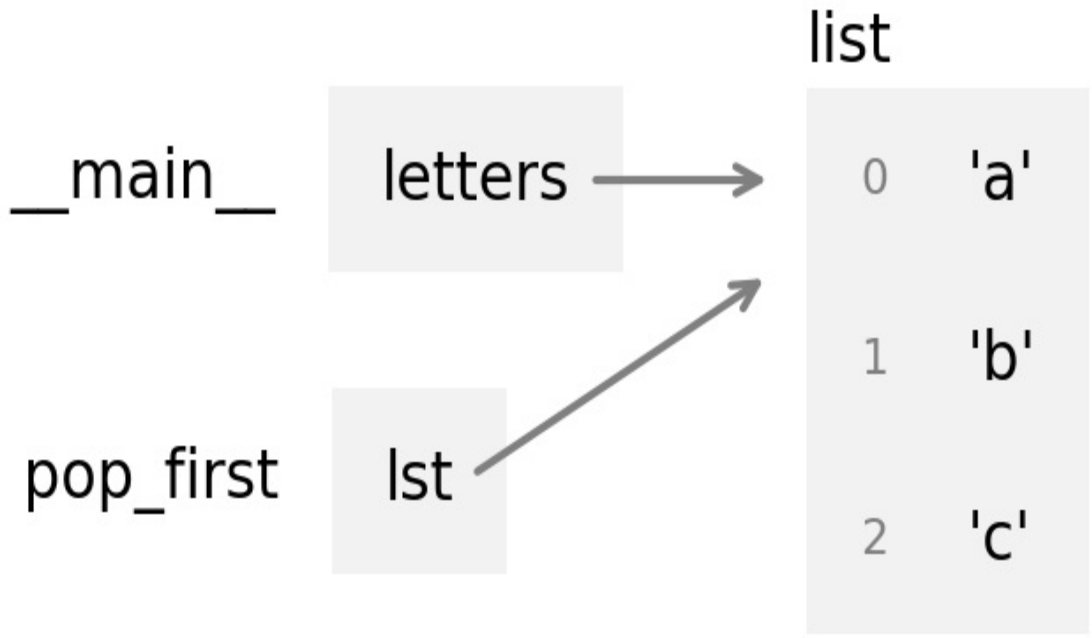
The return value is the first element, which has been removed from the list – as we can see by displaying the modified list.

```
In [48]: letters
```

```
Out[48]: ['b', 'c']
```

In this example, the parameter `lst` and the variable `letters` are aliases for the same object, so the stack diagram looks like this:

```
Out[48]: [2.04, 1.24, 1.06, 0.85]
```



Passing a reference to an object as an argument to a function creates a form of aliasing. If the function modifies the object, those changes persist after the function is done.

Making a word list

In the previous chapter, we read the file `words.txt` and searched for words with certain properties, like using the letter `e`. But we read the entire file many times, which is not efficient. It is better to read the file once and put the words in a list. The following loop shows how.

```
In [49]: word_list = []
         for line in open('words.txt'):
             word = line.strip()
             word_list.append(word)

         len(word_list)
```

```
Out[49]: 113783
```

Before the loop, `word_list` is initialized with an empty list. Each time through the loop, the `append` method adds a word to the end. When the loop is done, there are more than 113,000 words in the list.

Another way to do the same thing is to use `read` to read the entire file into a string.

```
In [50]: string = open('words.txt').read()
         len(string)
```

```
Out[50]: 1016511
```

The result is a single string with more than a million characters. We can use the `split` method to split it into a list of words.

```
In [51]: word_list = string.split()
         len(word_list)
```

```
Out[51]: 113783
```

Now, to check whether a string appears in the list, we can use the `in` operator. For example, `demotic` is in the list.

```
In [52]: 'demotic' in word_list
```

```
Out[52]: True
```

But `contrafibularities` is not.

```
In [53]: 'contrafibularities' in word_list
```

```
Out[53]: False
```

And I have to say, I'm anaspeptic about it.

Debugging

Note that most list methods modify the argument and return `None`. This is the opposite of the string methods, which return a new string and leave the original alone.

If you are used to writing string code like this:

```
In [54]: word = 'plumage!'
         word = word.strip('!')
         word


Out[54]: 'plumage'
```

It is tempting to write list code like this:

```
In [55]: t = [1, 2, 3]
         t = t.remove(3)            # WRONG!
```

`remove` modifies the list and returns `None`, so next operation you perform with `t` is likely to fail.

```
In [56]: t.remove(2)


Out[56]: AttributeError: 'NoneType' object has no attribute
'remove'
```

This error message takes some explaining. An **attribute** of an object is a variable or method associated with it. In this case, the value of `t` is `None`, which is a `NoneType` object, which does not have a attribute named `remove`, so the result is an `AttributeError`.

If you see an error message like this, you should look backward through the program and see if you might have called a list method incorrectly.

Exercises

Ask a virtual assistant

In this chapter, I used the words “contrafibularities” and “anaspeptic”, but they are not actually English words. They were used in the British television show *Black Adder*, Season 2, Episode 2, “Ink and Incapability”.

However, when I asked ChatGPT 3.5 (August 3, 2023 version) where those words came from, it initially claimed they are from Monty Python, and later claimed they are from the Tom Stoppard play *Rosencrantz and Guildenstern Are Dead*.

If you ask now, you might get different results. But this example is a reminder that virtual assistants are not always accurate, so you should check whether the results are correct. As you gain experience, you will get a sense of which questions virtual assistants can answer reliably. In this example, a conventional web search can identify the source of these words quickly.

If you get stuck on any of the exercises in this chapter, consider asking a virtual assistant for help. If you get a result that uses features we haven’t learned yet, you can assign the VA a “role”.

For example, before you ask a question try typing “Role: Basic Python Programming Instructor”. After that, the responses you get should use only basic features. If you still see features we you haven’t learned, you can follow up with “Can you write that using only basic Python features?”

Exercise

Two words are anagrams if you can rearrange the letters from one to spell the other. For example, `tops` is an anagram of `stop`.

One way to check whether two words are anagrams is to sort the letters in both words. If the lists of sorted letters are the same, the words are anagrams.

Write a function called `is_anagram` that takes two strings and returns `True` if they are anagrams.

Using your function and the word list, find all the anagrams of `takes`.

Exercise

Python provides a built-in function called `reversed` that takes as an argument a sequence of elements – like a list or string – and returns a `reversed` object that contains the elements in reverse order.

```
In [57]: reversed('parrot')
```

```
Out[57]: <reversed at 0x7fe3de636b60>
```

If you want the reversed elements in a list, you can use the `list` function.

```
In [58]: list(reversed('parrot'))
```

```
Out[58]: ['t', 'o', 'r', 'r', 'a', 'p']
```

Of if you want them in a string, you can use the `join` method.

```
In [59]: ''.join(reversed('parrot'))
```

```
Out[59]: 'torrap'
```

So we can write a function that reverses a word like this.

```
In [60]: def reverse_word(word):
             return ''.join(reversed(word))
```

A palindrome is a word that is spelled the same backward and forward, like "noon" and "rotator". Write a function called `is_palindrome` that takes a string argument and returns `True` if it is a palindrome and `False` otherwise.

You can use the following loop to find all of the palindromes in the word list with at least 7 letters.

```
In [61]: for word in word_list:
             if len(word) >= 7 and is_palindrome(word):
                 print(word)
```

Exercise

Write a function called `reverse_sentence` that takes as an argument a string that contains any number of words separated by spaces. It should return a new string that contains the same words in reverse order. For example, if the argument is "Reverse this sentence", the result should be "Sentence this reverse".

Hint: You can use the `capitalize` methods to capitalize the first word and convert the other words to lowercase.

Exercise

Write a function called `total_length` that takes a list of strings and returns the total length of the strings. The total length of the words in `word_list` should be 902,728.

Chapter 2. Dictionaries

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 11th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jbleiel@oreilly.com*.

This chapter presents a built-in type called a dictionary. It is one of Python's best features – and the building block of many efficient and elegant algorithms.

We'll use dictionaries to compute the number of unique words in a book and the number of times each one appears. And in the exercises, we'll use dictionaries to solve word puzzles.

A dictionary is a mapping

A **dictionary** is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type. For example, suppose we make a list of number words, like this.

```
In [1]: lst = ['zero', 'one', 'two']
```

We can use an integer as an index to get the corresponding word.

```
In [2]: lst[1]
```

```
Out[2]: 'one'
```

But suppose we want to go in the other direction, and look up a word to get the corresponding integer. We can't do that with a list, but we can with a dictionary. We'll start by creating an empty dictionary and assigning it to `numbers`.

```
In [3]: numbers = {}
        numbers
```

```
Out[3]: {}
```

The curly braces, `{}`, represent an empty dictionary. To add items to the dictionary, we'll use square brackets.

```
In [4]: numbers['zero'] = 0
```

This assignment adds to the dictionary an **item**, which represents the association of a **key** and a **value**. In this example, the key is the string `'zero'` and the value is the integer `0`. If we display the dictionary, we see that it contains one item, which contains a key and a value separated by a colon, `:`.

```
In [5]: numbers
```

```
Out[5]: {'zero': 0}
```

We can add more items like this.

```
In [6]: numbers['one'] = 1
        numbers['two'] = 2
        numbers


Out[6]: {'zero': 0, 'one': 1, 'two': 2}
```

Now the dictionary contains three items.

To look up a key and get the corresponding value, we use the bracket operator.

```
In [7]: numbers['two']


Out[7]: 2
```

If the key isn't in the dictionary, we get a `KeyError`.

```
In [8]: numbers['three']


Out[8]: KeyError: 'three'
```

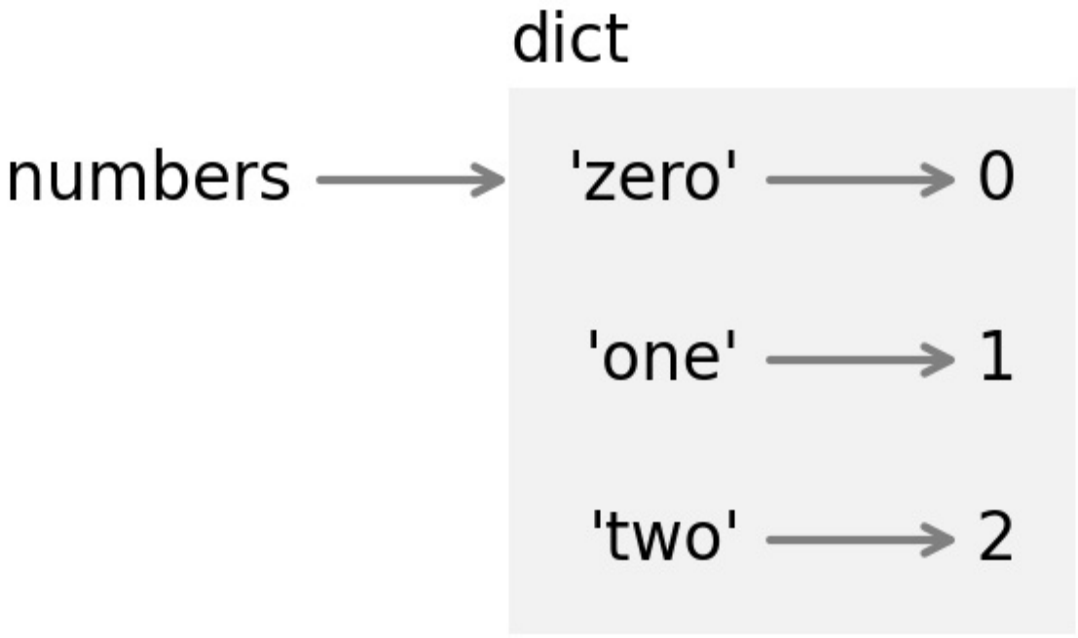
The `len` function works on dictionaries; it returns the number of items.

```
In [9]: len(numbers)


Out[9]: 3
```

In mathematical language, a dictionary represents a **mapping** from keys to values, so you can also say that each key "maps to" a value. In this example, each number word maps to the corresponding integer.

The following figure shows the state diagram for `numbers`.



A dictionary is represented by a box with the word “dict” outside and the items inside. Each item is represented by a key and an arrow pointing to a value. The quotation marks indicate that the keys here are strings, not variable names.

Creating dictionaries

In the previous section we created an empty dictionary and added items one at a time using the bracket operator. Instead, we could have created the dictionary all at once like this.

```
In [10]: numbers = {'zero': 0, 'one': 1, 'two': 2}
```

Each item consists of a key and a value separated by a colon. The items are separated by commas and enclosed in curly braces.

Another way to create a dictionary is to use the `dict` function. We can make an empty dictionary like this.

```
In [11]: empty = dict()
         empty


Out[11]: {}
```

And we can make a copy of a dictionary like this.

```
In [12]: numbers_copy = dict(numbers)
         numbers_copy


Out[12]: {'zero': 0, 'one': 1, 'two': 2}
```

It is often useful to make a copy before performing operations that modify dictionaries.

The in operator

The `in` operator works on dictionaries, too; it tells you whether something appears as a *key* in the dictionary.

```
In [13]: 'one' in numbers


Out[13]: True
```

The `in` operator does *not* check whether something appears as a value.

```
In [14]: 1 in numbers
```

```
Out[14]: False
```

To see whether something appears as a value in a dictionary, you can use the method `values`, which returns a sequence of values, and then use the `in` operator.

```
In [15]: 1 in numbers.values()
```

```
Out[15]: True
```

Python dictionaries use a data structure called a **hash table** that has a remarkable property: the `in` operator takes about the same amount of time no matter how many items are in the dictionary. That makes it possible to write some remarkably efficient algorithms.

To demonstrate, we'll compare two algorithms for finding pairs of words where one is the reverse of another – like `stressed` and `desserts`. We'll start by reading the word list.

```
In [16]: word_list = open('words.txt').read().split()
         len(word_list)
```

```
Out[16]: 113783
```

And here's `reverse_word` from the previous chapter.

```
In [17]: def reverse_word(word):
             return ''.join(reversed(word))
```

The following function loops through the words in the list. For each one, it reverses the letters and then checks whether the reversed word in the word list.

```
In [18]: def too_slow():
             count = 0
             for word in word_list:
                 reversed_word = reverse_word(word)
                 if reversed_word in word_list:
                     count += 1
             return count
```

This function takes more than a minute to run. The problem is that the `in` operator checks the words in the list one at a time, starting at the beginning. If it doesn't find what it's looking for – which happens most of the time – it has to search all the way to the end.

And the `in` operator is inside the loop, so it runs once for each word. Since there are more than 100,000 words in the list, and for each one we check more than 100,000 words, the total number of comparisons is the number of words squared – roughly – which is almost 13 billion.

```
In [19]: len(word_list)**2
```

```
Out[19]: 12946571089
```

We can make this function much faster with a dictionary. The following loop creates a dictionary that contains the words as keys.

```
In [20]: word_dict = {}
         for word in word_list:
             word_dict[word] = 1
```

The values in `word_dict` are all `1`, but they could be anything, because we won't ever look them up – we will only use this dictionary to check whether a key exists.

Now here's a version of the previous function that replaces `word_list` with `word_dict`.

```
In [21]: def much_faster():
             count = 0
             for word in word_dict:
                 reversed_word = reverse_word(word)
                 if reversed_word in word_dict:
                     count += 1
             return count
```

This function takes less than one hundredth of a second, so it's about 10,000 times faster than the previous version.

In general, the time it takes to find an element in a list is proportional to the length of the list. The time it takes to find a key in a dictionary is almost constant – regardless of the number of items.

A collection of counters

Suppose you are given a string and you want to count how many times each letter appears. A dictionary is a good tool for this job. We'll start with an empty dictionary.

```
In [22]: counter = {}
```

As we loop through the letters in the string, suppose we see the letter `a` for the first time. We can add it to the dictionary like this.

```
In [23]: counter['a'] = 1
```

The value `1` indicates that we have seen the letter once. Later, if we see the same letter again, we can increment the counter like this.

```
In [24]: counter['a'] += 1
```

Now the value associated with `a` is `2`, because we've seen the letter twice.

```
In [25]: counter
```

```
Out[25]: {'a': 2}
```

The following function uses these features to count the number of times each letter appears in a string.

```
In [26]: def value_counts(string):
             counter = {}
             for letter in string:
                 if letter not in counter:
                     counter[letter] = 1
                 else:
                     counter[letter] += 1
             return counter
```

Each time through the loop, if `letter` is not in the dictionary, we create a new item with key `letter` and value `1`. If `letter` is already in the dictionary we increment the value associated with `letter`.

Here's an example.

```
In [27]: counter = value_counts('brontosaurus')
         counter
```

```
Out[27]: {'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's': 2, 'a': 1,
'u': 2}
```

The items in `counter` show that the letter `b` appears once, `r` appears twice, and so on.

Looping and dictionaries

If you use a dictionary in a `for` statement, it traverses the keys of the

dictionary. To demonstrate, let's make a dictionary that counts the letters in `banana`.

```
In [28]: counter = value_counts('banana')
         counter
```

```
Out[28]: {'b': 1, 'a': 3, 'n': 2}
```

The following loop prints the keys, which are the letters.

```
In [29]: for key in counter:
             print(key)
```

```
Out[29]: b
         a
         n
```

To print the values, we can use the `values` method.

```
In [30]: for value in counter.values():
             print(value)
```

```
Out[30]: 1
         3
         2
```

To print the keys and values, we can loop through the keys and look up the corresponding values.

```
In [31]: for key in counter:
             value = counter[key]
             print(key, value)
```

```
Out[31]: b 1
```

```
a 3
n 2
```

In the next chapter, we'll see a more concise way to do the same thing.

Lists and dictionaries

You can put a list in a dictionary as a value. For example, here's a dictionary that maps from the number 4 to a list of four letters.

```
In [32]: d = {4: ['r', 'o', 'u', 's']}
         d


Out[32]: {4: ['r', 'o', 'u', 's']}
```

But you can't put a list in a dictionary as a key. Here's what happens if we try.

```
In [33]: d[letters] = 4


Out[33]: TypeError: unhashable type: 'list'
```

I mentioned earlier that dictionaries use hash tables, and that means that the keys have to be **hashable**.

A **hash** is a function that takes a value (of any kind) and returns an integer. Dictionaries use these integers, called hash values, to store and look up keys.

This system only works if a key is immutable, so its hash value is always the same. But if a key is mutable, its hash value could change, and the dictionary would not work. That's why keys have to be hashable, and why mutable types like lists aren't.

Since dictionaries are mutable, they can't be used as keys, either. But they

can be used as values.

Memos

If you ran the `fibonacci` function from Section xxx, you might have noticed that the bigger the argument you provide, the longer the function takes to run.

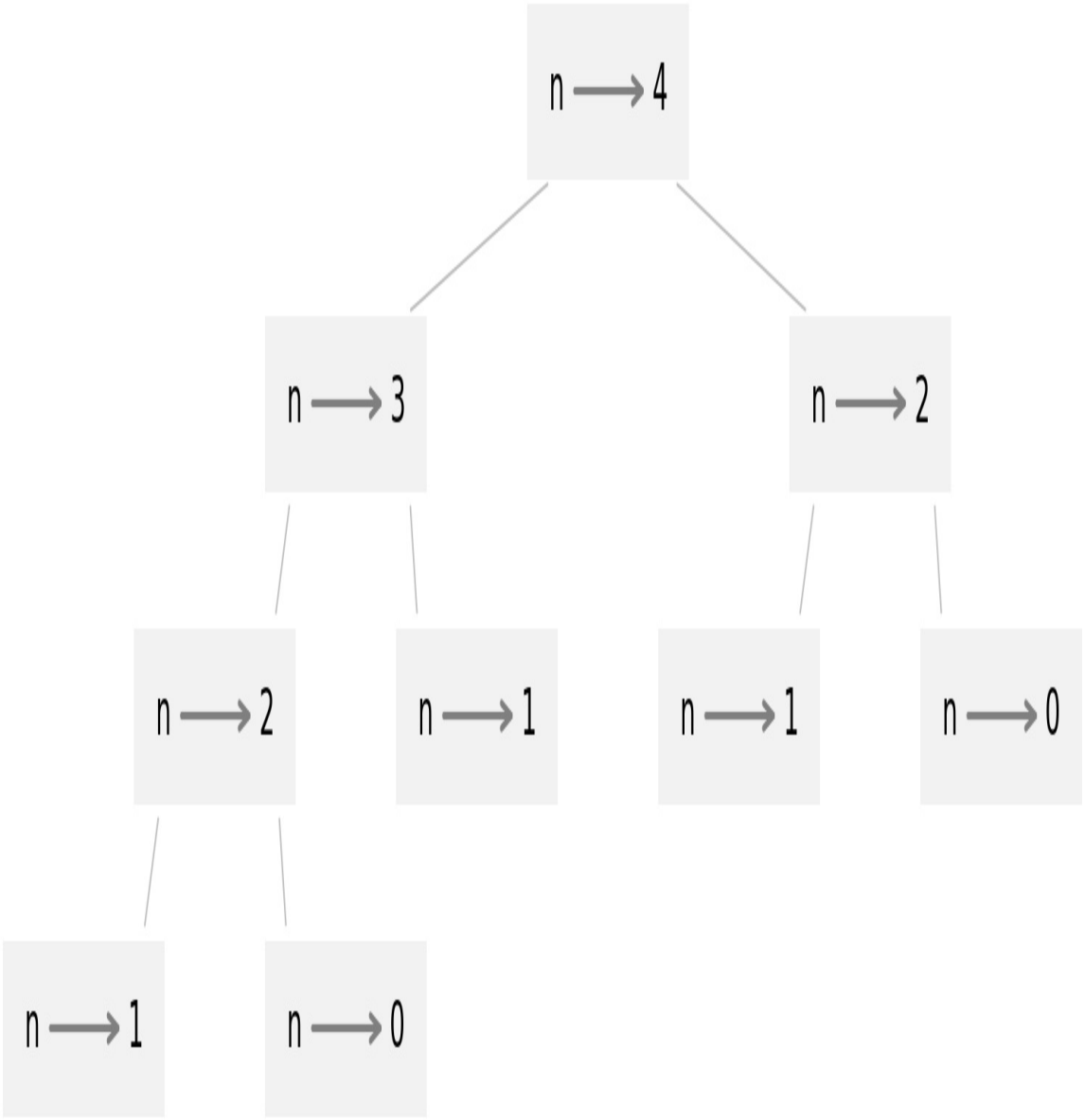
```
In [34]: def fibonacci(n):
             if n == 0:
                 return 0

             if n == 1:
                 return 1

             return fibonacci(n-1) + fibonacci(n-2)
```

Furthermore, the run time increases quickly.

To understand why, consider the following figure, which shows the **call graph** for `fibonacci` with `n=3`:



A call graph shows a set of function frames, with lines connecting each frame to the frames of the functions it calls. At the top of the graph, `fibonacci` with `n=4` calls `fibonacci` with `n=3` and `n=2`. In turn, `fibonacci` with `n=3` calls `fibonacci` with `n=2` and `n=1`. And so on.

Count how many times `fibonacci(0)` and `fibonacci(1)` are called.

This is an inefficient solution to the problem, and it gets worse as the argument gets bigger.

One solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a **memo**. Here is a "memoized" version of `fibonacci`:

```
In [35]: known = {0:0, 1:1}

         def fibonacci_memo(n):
             if n in known:
                 return known[n]

             res = fibonacci_memo(n-1) + fibonacci_memo(n-2)
             known[n] = res
             return res
```

`known` is a dictionary that keeps track of the Fibonacci numbers we already know It starts with two items: `0` maps to `0` and `1` maps to `1`.

Whenever `fibonacci_memo` is called, it checks `known`. If the result is already there, it can return immediately. Otherwise it has to compute the new value, add it to the dictionary, and return it.

Comparing the two functions, `fibonacci(40)` takes about 30 seconds to run. `fibonacci_memo(40)` takes about 30 microseconds, so it's a million times faster.

Debugging

As you work with bigger datasets it can become unwieldy to debug by printing and checking the output by hand. Here are some suggestions for debugging large datasets:

1. Scale down the input: If possible, reduce the size of the dataset. For example if the program reads a text file, start with just the first 10 lines, or with the smallest example you can find. You can either edit the files

themselves, or (better) modify the program so it reads only the first `n` lines.

If there is an error, you can reduce `n` to the smallest value where the error occurs. As you find and correct errors, you can increase `n` gradually.

2. Check summaries and types: Instead of printing and checking the entire dataset, consider printing summaries of the data – for example, the number of items in a dictionary or the total of a list of numbers.

   A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.

3. Write self-checks: Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a “sanity check” because it detects results that are “insane”.

   Another kind of check compares the results of two different computations to see if they are consistent. This is called a “consistency check”.

4. Format the output: Formatting debugging output can make it easier to spot an error. We saw an example in Section xxx. Another tool you might find useful is the `pprint` module, which provides a `pprint` function that displays built-in types in a more human-readable format (`pprint` stands for “pretty print”).

   Again, time you spend building scaffolding can reduce the time you spend debugging.

Exercises

Ask an assistant

In this chapter, I said the keys in a dictionary have to be hashable and I gave a short explanation. If you would like more details, ask a virtual assistant, "Why do keys in Python dictionaries have to be hashable?"

In Section xxx, we stored a list of words as keys in a dictionary so that we could use an efficient version of the `in` operator. We could have done the same thing using a `set`, which is another built-in data type. Ask a virtual assistant, "How do I make a Python set from a list of strings and check whether a string is an element of the set?"

Exercise

Dictionaries have a method called `get` that takes a key and a default value. If the key appears in the dictionary, `get` returns the corresponding value; otherwise it returns the default value. For example, here's a dictionary that maps from the letters in a string to the number of times they appear.

```
In [36]: counter = value_counts('brontosaurus')
```

If we look up a letter that appears in the word, `get` returns the number of times it appears.

```
In [37]: counter.get('b', 0)
```

```
Out[37]: 1
```

If we look up a letter that doesn't appear, we get the default value, `0`.

```
In [38]: counter.get('c', 0)
```

```
Out[38]: 0
```

Use `get` to write a more concise version of `value_counts`. You should be able to eliminate the `if` statement.

Exercise

Suppose you run `value_counts` twice and save the results in two dictionaries.

```
In [39]: counter1 = value_counts('brontosaurus')
         counter2 = value_counts('apatosaurus')
```

Each dictionary maps from a set of letters to the number of times they appear. Write a function called `add_counters` that takes two dictionaries like this and returns a new dictionary that contains all of the letters and the total number of times they appear in either word.

There are many ways to solve this problem. Once you have a working solution, consider asking a virtual assistant for different solutions.

Exercise

What is the longest word you can think of where each letter appears only once? Let's see if we can find one longer than `unpredictably`.

Write a function named `has_duplicates` that takes a sequence – like a list or string – as a parameter and returns `True` if there is any element that appears in the sequence more than once.

Exercise

A word is "interlocking" if we can split it into two words by taking alternating letters. For example, "schooled" is an interlocking word because it can be split into "shoe" and "cold".

To select alternating letters from a string, you can use a slice operator with three components that indicate where to start, where to stop, and the "step

size” between the letters.

In the following slice, the first component is `0`, so we start with the first letter. The second component is `None`, which means we should go all the way to the end of the string. And the third component is `2`, so there are two steps between the letters we select.

```
In [40]: word = 'schooled'
         first = word[0:None:2]
         first


Out[40]: 'shoe'
```

Instead of providing `None` as the second component, we can get the same effect by leaving it out altogether. For example, the following slice selects alternating letters, starting with the second letter.

```
In [41]: second = word[1::2]
         second


Out[41]: 'cold'
```

Write a function called `is_interlocking` that takes a word as an argument and returns `True` if it can be split into two interlocking words.

Chapter 3. Tuples

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 12th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jbleiel@oreilly.com*.

This chapter introduces one more built-in type, the tuple, and then shows how lists, dictionaries, and tuples work together. It also presents tuple assignment and a useful feature for functions with variable-length argument lists: the packing and unpacking operators.

In the exercises, we'll use tuples, along with lists and dictionaries, to solve more word puzzles and implement efficient algorithms.

One note: There are two ways to pronounce "tuple". Some people say "tuh-ple", which rhymes with "supple". But in the context of programming, most people say "too-ple", which rhymes with "quadruple".

Tuples are like lists

A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so tuples are a lot like lists. The important difference is that tuples are immutable.

To create a tuple, you can write a comma-separated list of values.

```
In [1]: t = 'l', 'u', 'p', 'i', 'n'
        type(t)
```

```
Out[1]: tuple
```

Although it is not necessary, it is common to enclose tuples in parentheses.

```
In [2]: t = ('l', 'u', 'p', 'i', 'n')
        type(t)
```

```
Out[2]: tuple
```

To create a tuple with a single element, you have to include a final comma.

```
In [3]: t1 = 'p',
        type(t1)
```

```
Out[3]: tuple
```

A single value in parentheses is not a tuple.

```
In [4]: t2 = ('p')
        type(t2)
```

```
Out[4]: str
```

Another way to create a tuple is the built-in function `tuple`. With no argument, it creates an empty tuple.

```
In [5]: t = tuple()
        t
```

```
Out[5]: ()
```

If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence.

```
In [6]: t = tuple('lupin')
        t
```

```
Out[6]: ('l', 'u', 'p', 'i', 'n')
```

You can also use `tuple` to make a copy of a tuple, but since tuples are immutable, it is almost never necessary to copy one.

Because `tuple` is the name of a built-in function, you should avoid using it as a variable name.

Most list operators also work with tuples. For example, the bracket operator indexes an element.

```
In [7]: t[0]
```

```
Out[7]: 'l'
```

And the slice operator selects a range of elements.

```
In [8]: t[1:3]
```

```
Out[8]: ('u', 'p')
```

The + operator concatenates tuples.

```
In [9]: tuple('lup') + ('i', 'n')
```

```
Out[9]: ('l', 'u', 'p', 'i', 'n')
```

And the `*` operator duplicates a tuple a given number of times.

```
In [10]: tuple('spam') * 2
```

```
Out[10]: ('s', 'p', 'a', 'm', 's', 'p', 'a', 'm')
```

The `sorted` function works with tuples – but the result is a list, not a tuple.

```
In [11]: sorted(t)
```

```
Out[11]: ['i', 'l', 'n', 'p', 'u']
```

The `reversed` function also works with tuples.

```
In [12]: reversed(t)
```

```
Out[12]: <reversed at 0x7f56c0072110>
```

The result is a `reversed` object, which we can convert to a list or tuple.

```
In [13]: tuple(reversed(t))
```

```
Out[13]: ('n', 'i', 'p', 'u', 'l')
```

Based on the examples so far, it might seem like tuples are the same as lists.

But tuples are immutable

If you try to modify a tuple with the bracket operator, you get a `TypeError`.

```
In [14]: t[0] = 'L'
```

```
Out[14]: TypeError: 'tuple' object does not support item assignment
```

And tuples don't have any of the methods that modify lists, like `append` and `remove`.

```
In [15]: t.remove('l')
```

```
Out[15]: AttributeError: 'tuple' object has no attribute 'remove'
```

Recall that an "attribute" is a variable or method associated with an object – this error message means that tuples don't have a method named `remove`.

Because tuples are immutable, they are hashable, which means they can be used as keys in a dictionary. For example, the following dictionary contains two tuples as keys that map to integers.

```
In [16]: d = {}
         d[1, 2] = 3
         d[3, 4] = 7
```

We can look up a tuple in a dictionary like this:

```
In [17]: d[1, 2]
```

```
Out[17]: 3
```

Or if we have a variable that refers to a tuple, we can use it as a key.

```
In [18]: t = (3, 4)
         d[t]
```

```
Out[18]: 7
```

Tuples can also appear as values in a dictionary.

Tuple assignment

You can put a tuple of variables on the left side of an assignment, and a tuple of values on the right.

```
In [19]: a, b = 1, 2
```

The values are assigned to the variables from left to right – in this example, `a` gets the value `1` and `b` gets the value `2`. We can display the results like this:

```
In [20]: a, b
```

```
Out[20]: (1, 2)
```

More generally, if the left side of an assignment is a tuple, the right side can be any kind of sequence – string, list or tuple. For example, to split an email address into a user name and a domain, you could write:

```
In [21]: email = 'monty@python.org'
         username, domain = email.split('@')
```

The return value from `split` is a list with two elements – the first element is assigned to `username`, the second to `domain`.

```
In [22]: username, domain
```

```
Out[22]: ('monty', 'python.org')
```

The number of variables on the left and the number of values on the right have to be the same – otherwise you get a `ValueError`.

```
In [23]: a, b = 1, 2, 3
```

```
Out[23]: ValueError: too many values to unpack (expected 2)
```

Tuple assignment is useful if you want to swap the values of two variables. With conventional assignments, you have to use a temporary variable, like this:

```
In [24]: temp = a
         a = b
         b = temp
```

That works, but with tuple assignment we can do the same thing without a temporary variable.

```
In [25]: a, b = b, a
```

This works because all of the expressions on the right side are evaluated before any of the assignments.

We can also use tuple assignment in a `for` statement. For example, to loop through the items in a dictionary, we can use the `items` method.

```
In [26]: d = {'one': 1, 'two': 2}

         for item in d.items():
             key, value = item
             print(key, '->', value)
```

```
Out[26]: one -> 1
         two -> 2
```

Each time through the loop, `item` is assigned a tuple that contains a key and the corresponding value.

We can write this loop more concisely, like this:

```
In [27]: for key, value in d.items():
             print(key, '->', value)
```

```
Out[27]: one -> 1
         two -> 2
```

Each time through the loop, a key and the corresponding value are assigned directly to `key` and `value`.

Tuples as return values

Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute `x//y` and then `x%y`. It is better to compute them both at the same time.

The built-in function `divmod` takes two arguments and returns a tuple of two values, the quotient and remainder.

```
In [28]: divmod(7, 3)
```

```
Out[28]: (2, 1)
```

We can use tuple assignment to store the elements of the tuple in two variables.

```
In [29]: quotient, remainder = divmod(7, 3)
         quotient
```

```
Out[29]: 2
```

```
In [30]: remainder
```

```
Out[30]: 1
```

Here is an example of a function that returns a tuple.

```
In [31]: def min_max(t):
             return min(t), max(t)
```

`max` and `min` are built-in functions that find the largest and smallest elements of a sequence. `min_max` computes both and returns a tuple of two values.

```
In [32]: min_max([2, 4, 1, 3])
```

```
Out[32]: (1, 4)
```

We can assign the results to variables like this:

```
In [33]: low, high = min_max([2, 4, 1, 3])
         low, high
```

```
Out[33]: (1, 4)
```

Argument packing

Functions can take a variable number of arguments. A parameter name that begins with the `*` operator **packs** arguments into a tuple. For example, the following function takes any number of arguments and computes their arithmetic mean – that is, their sum divided by the number of arguments.

```
In [34]: def mean(*args):
             return sum(args) / len(args)
```

The parameter can have any name you like, but `args` is conventional. We can call the function like this:

```
In [35]: mean(1, 2, 3)
```

```
Out[35]: 2.0
```

If you have a sequence of values and you want to pass them to a function as multiple arguments, you can use the `*` operator to **unpack** the tuple. For example, `divmod` takes exactly two arguments – if you pass a tuple as a parameter, you get an error.

```
In [36]: t = (7, 3)
         divmod(t)
```

```
Out[36]: TypeError: divmod expected 2 arguments, got 1
```

Even though the tuple contains two elements, it counts as a single argument. But if you unpack the tuple, it is treated as two arguments.

```
In [37]: divmod(*t)
```

```
Out[37]: (2, 1)
```

Packing and unpacking can be useful if you want to adapt the behavior of an existing function. For example, this function takes any number of arguments, removes the lowest and highest, and computes the mean of the rest.

```
In [38]: def trimmed_mean(*args):
             low, high = min_max(args)
             trimmed = list(args)
             trimmed.remove(low)
             trimmed.remove(high)
             return mean(*trimmed)
```

First it uses `min_max` to find the lowest and highest elements. Then it converts `args` to a list so it can use the `remove` method. Finally it unpacks the list so the elements are passed to `mean` as separate arguments, rather than as a single list.

Here's an example that shows the effect.

```
In [39]: mean(1, 2, 3, 10)
```

```
Out[39]: 4.0
```

```
In [40]: trimmed_mean(1, 2, 3, 10)
```

```
Out[40]: 2.5
```

This kind of "trimmed" mean is used in some sports with subjective judging – like diving and gymnastics – to reduce the effect of a judge whose score deviates from the others.

Zip

Tuples are useful for looping through the elements of two sequences and performing operations on corresponding elements. For example, suppose two teams play a series of seven games, and we record their scores in two lists, one for each team.

```
In [41]: scores1 = [1, 2, 4, 5, 1, 5, 2]
         scores2 = [5, 5, 2, 2, 5, 2, 3]
```

Let's see how many games each team won. We'll use `zip`, which is a built-in function that takes two or more sequences and returns a **zip object**.

```
In [42]: zip(scores1, scores2)
```

```
Out[42]: <zip at 0x7f56c01286c0>
```

We can use the zip object to loop through the values in the sequences pairwise.

```
In [43]: for pair in zip(scores1, scores2):
             print(pair)
```

```
Out[43]: (1, 5)
         (2, 5)
         (4, 2)
         (5, 2)
         (1, 5)
         (5, 2)
         (2, 3)
```

Each time through the loop, `pair` gets assigned a tuple of scores. So we can assign the scores to variables, and count the victories for the first team, like this:

```
In [44]: wins = 0
         for team1, team2 in zip(scores1, scores2):
             if team1 > team2:
                 wins += 1

         wins


Out[44]: 3
```

Sadly, the first team won only three games and lost the series.

If you have two lists and you want a list of pairs, you can use `zip` and `list`.

```
In [45]: t = list(zip(scores1, scores2))
```

The result is a list of tuples, so we can get the result of the last game like this:

```
In [46]: t[-1]


Out[46]: (2, 3)
```

If you have a list of keys and a list of values, you can use `zip` and `dict` to make a dictionary. For example, here's how we can make a dictionary that maps from each letter to its position in the alphabet.

```
In [47]: letters = 'abcdefghijklmnopqrstuvwxyz'
         numbers = range(len(letters))
         letter_map = dict(zip(letters, numbers))
```

Now we can look up a letter and get its index in the alphabet.

```
In [48]: letter_map['a'], letter_map['z']


Out[48]: (0, 25)
```

In this mapping, the index of `a` is `0` and the index of `z` is `25`.

If you need to loop through the elements of a sequence and their indices, you can use the built-in function `enumerate`.

```
In [49]: enumerate('abc')
```

```
Out[49]: <enumerate at 0x7f56c012a1c0>
```

The result is an **enumerate object** that loops through a sequence of pairs, where each pair contains an index (starting from 0) and an element from the given sequence.

```
In [50]: for index, element in enumerate('abc'):
             print(index, element)
```

```
Out[50]: 0 a
         1 b
         2 c
```

Comparing and Sorting

The relational operators work with tuples and other sequences. For example, if you use the `<` operator with tuples, it starts by comparing the first element from each sequence. If they are equal, it goes on to the next pair of elements, and so on, until it finds a pair that differ.

```
In [51]: (0, 1, 2) < (0, 3, 4)
```

```
Out[51]: True
```

Subsequent elements are not considered – even if they are really big.

```
In [52]: (0, 1, 2000000) < (0, 3, 4)
```

```
Out[52]: True
```

This way of comparing tuples is useful for sorting a list of tuples, or finding the minimum or maximum. As an example, let's find the most common letter in a word. In the previous chapter, we wrote `value_counts`, which takes a string and returns a dictionary that maps from each letter to the number of times it appears.

```
In [53]: def value_counts(string):
             counter = {}
             for letter in string:
                 if letter not in counter:
                     counter[letter] = 1
                 else:
                     counter[letter] += 1
             return counter
```

Here is the result for the string `banana`.

```
In [54]: counter = value_counts('banana')
         counter
```

```
Out[54]: {'b': 1, 'a': 3, 'n': 2}
```

With only three times, we can easily see that the most frequent letter is `a`, which appears three times. But if there were more items, it would be useful to sort them automatically.

We can get the items from `counter` like this.

```
In [55]: items = counter.items()
```

```
In [55]: items

Out[55]: dict_items([('b', 1), ('a', 3), ('n', 2)])
```

The result is a `dict_items` object that behaves like a list of tuples, so we can sort it like this.

```
In [56]: sorted(items)

Out[56]: [('a', 3), ('b', 1), ('n', 2)]
```

The default behavior is to use the first element from each tuple to sort the list, and use the second element to break ties.

However, to find the items with the highest counts, we want to use the second element to sort the list. We can do that by writing a function that takes a tuple and returns the second element.

```
In [57]: def second_element(t):
             return t[1]
```

Then we can pass that function to `sorted` as an optional argument called `key`, which indicates that this function should be used to compute the **sort key** for each item.

```
In [58]: sorted_items = sorted(items, key=second_element)
         sorted_items

Out[58]: [('b', 1), ('n', 2), ('a', 3)]
```

The sort key determines the order of the items in the list. The letter with the lowest count appears first, and the letter with the highest count appears last.

So we can find the most common letter like this.

```
In [59]: sorted_items[-1]
```

```
Out[59]: ('a', 3)
```

If we only want the maximum, we don't have to sort the list. We can use `max`, which also takes `key` as an optional argument.

```
In [60]: max(items, key=second_element)
```

```
Out[60]: ('a', 3)
```

To find the letter with the lowest count, we could use `min` the same way.

Inverting a dictionary

Suppose you want to invert a dictionary so you can look up a value and get the corresponding key. For example, if you have a word counter that maps from each word to the number of times it appears, you could make a dictionary that maps from integers to the words that appear that number of times.

But there's a problem – the keys in a dictionary have to be unique, but the values don't. For example, in a word counter, there could be many words with the same count.

So one way to invert a dictionary is to create a new dictionary where the values are lists of keys from the original. As an example, let's count the letters in `parrot`.

```
In [61]: d =  value_counts('parrot')
         d
```

```
Out[61]: {'p': 1, 'a': 1, 'r': 2, 'o': 1, 't': 1}
```

If we invert this dictionary, the result should be `{1: ['p', 'a', 'o', 't'], 2: ['r']}`, which indicates that the letters that appear once are `p`, `a`, `o`, and `t`, and the letter than appears twice is `2`.

The following function takes a dictionary and returns its inverse as a new dictionary.

```
In [62]: def invert_dict(d):
             new = {}
             for key, value in d.items():
                 if value not in new:
                     new[value] = [key]
                 else:
                     new[value].append(key)
             return new
```

The `for` statement loops through the keys and values in `d`. If the value is not already in the new dictionary, it is added and associated with a list that contains a single element. Otherwise it is appended to the existing list.

We can test it like this:

```
In [63]: invert_dict(d)
```

```
Out[63]: {1: ['p', 'a', 'o', 't'], 2: ['r']}
```

And we get the result we expected.

This is the first example we've seen where the values in the dictionary are lists. We will see more!

Debugging

Lists, dictionaries and tuples are **data structures**. In this chapter we are

starting to see compound data structures, like lists of tuples, or dictionaries that contain tuples as keys and lists as values. Compound data structures are useful, but they are prone to errors caused when a data structure has the wrong type, size, or structure. For example, if a function expects a list if integers and you give it a plain old integer (not in a list), it probably won't work.

To help debug these kinds of errors, I wrote a module called `structshape` that provides a function, also called `structshape`, that takes any kind of data structure as an argument and returns a string that summarizes its structure. You can download it from https://raw.githubusercontent.com/AllenDowney/ThinkPython/v3/structshape.

We can import it like this.

```
In [64]: from structshape import structshape
```

Here's an example with a simple list.

```
In [65]: t = [1, 2, 3]
         structshape(t)
```

```
Out[65]: 'list of 3 int'
```

Here's a list of lists.

```
In [66]: t2 = [[1,2], [3,4], [5,6]]
         structshape(t2)
```

```
Out[66]: 'list of 3 list of 2 int'
```

If the elements of the list are not the same type, `structshape` groups them by type.

```
In [67]: t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
         structshape(t3)
```

```
Out[67]: 'list of (3 int, float, 2 str, 2 list of int, int)'
```

Here's a list of tuples.

```
In [68]: s = 'abc'
         lt = list(zip(t, s))
         structshape(lt)
```

```
Out[68]: 'list of 3 tuple of (int, str)'
```

And here's a dictionary with three items that map integers to strings.

```
In [69]: d = dict(lt)
         structshape(d)
```

```
Out[69]: 'dict of 3 int->str'
```

If you are having trouble keeping track of your data structures, `structshape` can help.

Exercises

Ask a virtual assistant

The exercises in this chapter might be more difficult than exercises in previous chapters, so I encourage you to get help from a virtual assistant. When you pose more difficult questions, you might find that the answers are not correct on the first attempt, so this is a chance to practice crafting good prompts and following up with good refinements.

One strategy you might consider is to break a big problems into pieces that can be solved with simple functions. Ask the virtual assistant to write the functions and test them. Then, once they are working, ask for a solution to the original problem.

For some of the exercises below, I make suggestions about which data structures and algorithms to use. You might find these suggestions useful when you work on the problems, but they are also good prompts to pass along to a virtual assistant.

Exercise

In this chapter we made a dictionary that maps from each letter to its index in the alphabet.

```
In [70]: letters = 'abcdefghijklmnopqrstuvwxyz'
         numbers = range(len(letters))
         letter_map = dict(zip(letters, numbers))
```

For example, the index of `a` is `0`.

```
In [71]: letter_map['a']
```

```
Out[71]: 0
```

To go in the other direction, we can use list indexing. For example, the letter at index `1` is `b`.

```
In [72]: letters[1]
```

```
Out[72]: 'b'
```

We can use `letter_map` and `letters` to encode and decode words using

a Caesar cipher.

A Caesar cipher is a weak form of encryption that involves shifting each letter by a fixed number of places in the alphabet, wrapping around to the beginning if necessary. For example, `a` shifted by 2 is `c` and `z` shifted by 1 is `a`.

Write a function called `shift_word` that takes as parameters a string and an integer, and returns a new string that contains the letters from the string shifted by the given number of places.

To test your function, confirm that “cheer” shifted by 7 is “jolly” and “melon” shifted by 16 is “cubed”.

Hints: Use the modulus operator to wrap around from `z` back to `a`. Loop through the letters of the word, shift each one, and append the result to a list of letters. Then use `join` to concatenate the letters into a string.

Exercise

Write a function called `most_frequent_letters` that takes a string and prints the letters in decreasing order of frequency.

To get the items in decreasing order, you can use `reversed` along with `sorted` or you can pass `reverse=True` as a keyword parameter to `sorted`.

Exercise

In a previous exercise, we tested whether two strings are anagrams by sorting the letters in both words and checking whether the sorted letters are the same. Now let’s make the problem a little more challenging.

We’ll write a program that takes a list of words and prints all the sets of words that are anagrams. Here is an example of what the output might look like:

```
['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
['retainers', 'ternaries']
```

```
['generating', 'greatening']
['resmelts', 'smelters', 'termless']
```

Hint: For each word in the word list, sort the letters and join them back into a string. Make a dictionary that maps from this sorted string to a list of words that are anagrams of it.

Exercise

Write a function called `word_distance` that takes two words with the same length and returns the number of places where the two words differ.

Hint: Use `zip` to loop through the corresponding letters of the words.

Exercise

"Metathesis" is the transposition of letters in a word. Two words form a "metathesis pair" if you can transform one into the other by swapping two letters, like `converse` and `conserve`. Write a program that finds all of the metathesis pairs in the word list.

Hint: The words in a metathesis pair must be anagrams of each other.

Credit: This exercise is inspired by an example at http://puzzlers.org.

Exercise

Here's another Car Talk Puzzler (http://www.cartalk.com/content/puzzlers):

> *What is the longest English word, that remains a valid English word, as you remove its letters one at a time?*
>
> *Now, letters can be removed from either end, or the middle, but you can't rearrange any of the letters. Every time you drop a letter, you wind up with another English word. If you do that, you're eventually going to wind up with one letter and that too is going to be an English word—one that's found in the dictionary. I want to know what's the longest word and how many letters does it have?*

> *I'm going to give you a little modest example: Sprite. Ok? You start off with sprite, you take a letter off, one from the interior of the word, take the r away, and we're left with the word spite, then we take the e off the end, we're left with spit, we take the s off, we're left with pit, it, and I.*

Write a program to find all words that can be reduced in this way, and then find the longest one.

This exercise is a little more challenging than most, so here are some suggestions:

1. You might want to write a function that takes a word and computes a list of all the words that can be formed by removing one letter. These are the "children" of the word.
2. Recursively, a word is reducible if any of its children are reducible. As a base case, you can consider the empty string reducible.
3. The word list we've been using doesn't contain single letter words. So you might have to add "I" and "a".
4. To improve the performance of your program, you might want to memoize the words that are known to be reducible.

Chapter 4. Text Analysis and Generation

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 13th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jbleiel@oreilly.com*.

At this point we have covered Python's core data structures – lists, dictionaries, and tuples – and and some algorithms that use them. In this chapter, we'll use them to explore text analysis and Markov generation:

- Text analysis is a way to describe the statistical relationships between the words in a document, like the probability that one word is followed by another, and
- Markov generation is a way to generate new text with words and phrases similar to the original text.

These algorithms are similar to parts of a Large Language Model (LLM), which is the key component of a chatbot.

We'll by counting the number of times each word appears in a book. Then we'll look at pairs of words, and make a list of the words that can follow each word. We'll make a simple version of a Markov generator, and as an exercise, you'll have a chance to make a more general version.

Read a book!

As a first step toward text analysis, let's read a book – *The Strange Case Of Dr. Jekyll And Mr. Hyde* by Robert Louis Stevenson – and count the number of unique words.

```
In [1]: filename = 'dr_jekyll.txt'
```

We'll use a `for` loop to read lines from the file and `split` to divide the lines into words. Then, to keep track of unique words, we'll store each word as a key in a dictionary.

```
In [2]: unique_words = {}
        for line in open(filename):
            seq = line.split()
            for word in seq:
                unique_words[word] = 1

        len(unique_words)
```

```
Out[2]: 6040
```

The length of the dictionary is the number of unique words – about `6000` by this way of counting. But if we inspect them, we'll see that some are not valid words.

For example, let's look at the longest words in `unique_words`. We can use `sorted` to sort the words, passing the `len` function as a keyword argument so the words are sorted by length.

```
In [3]: sorted(unique_words, key=len)[-5:]
```

```
Out[3]: ['chocolate-coloured',
         'superiors—behold!”',
         'coolness—frightened',
         'gentleman—something',
```

```
 'pocket-handkerchief.']
```

The slice index, `[-5:]`, selects the last `5` elements of the sorted list, which are the longest words.

The list includes some legitimately long words, like “circumscription”, and some hyphenated words, like “chocolate-coloured”. But some of the longest “words” are actually two words separated by a dash. And other words include punctuation like periods, exclamation points, and quotation marks.

So, before we move on, let’s deal with dashes and other punctuation.

Punctuation

To identify the words in the text, we need to deal with two issues:

- When a dash appears in a line, we should replace it with a space – then when we use `split`, the words will be separated.
- After splitting the words, we can use `strip` to remove punctuation.

To handle the first issue, we can use the following function, which takes a string, replaces dashes with spaces, splits the string, and returns the resulting list.

```
In [4]: def split_line(line):
            return line.replace('—', ' ').split()
```

Here’s an example.

```
In [5]: split_line('coolness—frightened')
```

```
Out[5]: ['coolness', 'frightened']
```

Now, to remove punctuation from the beginning and end of each word, we

can use `strip`, but we need a list of characters that are considered punctuation.

Characters in Python strings are in Unicode, which is an international standard used to represent letters in nearly every alphabet, numbers, symbols, punctuation marks, and more. The `unicodedata` module provides a `category` function we can use to tell which characters are punctuation. Given a letter, it returns a string with information about what category the letter is in.

```
In [6]: import unicodedata

        unicodedata.category('A')


Out[6]: 'Lu'
```

The category string of `A` is `Lu` – the `L` means it is a letter and the `u` means it is uppercase.

The category string of `.` is `Po` – the `P` means it is punctuation, the `o` means its subcategory is “other”.

```
In [7]: unicodedata.category('.')


Out[7]: 'Po'
```

We can find the punctuation marks in the book by checking for characters with categories that begin with `P`. The following loop stores the unique punctuation marks in a dictionary.

```
In [8]: punc_marks = {}
        for line in open(filename):
            for char in line:
                category = unicodedata.category(char)
                if category.startswith('P'):
                    punc_marks[char] = 1
```

To make a list of punctuation marks, we can join the keys of the dictionary into a string.

```
In [9]: punctuation = ''.join(punc_marks)
        print(punctuation)
```

```
Out[9]: .';,-“”:?—‘!()_
```

Now that we know which characters in the book are punctuation, we can write a function that takes a word, strips punctuation from the beginning and end, and converts it to lower case.

```
In [10]: def clean_word(word):
             return word.strip(punctuation).lower()
```

Here's an example.

```
In [11]: clean_word('“Behold!”')
```

```
Out[11]: 'behold'
```

Because `strip` removes characters from the beginning and end, it leaves hyphenated words alone.

```
In [12]: clean_word('pocket-handkerchief')
```

```
Out[12]: 'pocket-handkerchief'
```

Now here's a loop that uses `split_line` and `clean_word` to identify the unique words in the book.

```
In [13]: unique_words2 = {}
         for line in open(filename):
             for word in split_line(line):
                 word = clean_word(word)
                 unique_words2[word] = 1

         len(unique_words2)


Out[13]: 4005
```

With this stricter definition of what a word is, there are about 4000 unique words. And we can confirm that the list of longest words has been cleaned up.

```
In [14]: sorted(unique_words2, key=len)[-5:]


Out[14]: ['circumscription',
          'unimpressionable',
          'fellow-creatures',
          'chocolate-coloured',
          'pocket-handkerchief']
```

Now let’s see how many times each word is used.

Word frequencies

The following loop computes the frequency of each unique word.

```
In [15]: word_counter = {}
         for line in open(filename):
             for word in split_line(line):
                 word = clean_word(word)
                 if word not in word_counter:
                     word_counter[word] = 1
                 else:
                     word_counter[word] += 1
```

The first time we see a word, we initialize its frequency to `1`. If we see the same word again later, we increment its frequency.

To see which words appear most often, we can use `items` to get the key-value pairs from `word_counter`, and sort them by the second element of the pair, which is the frequency. First we'll define a function that selects the second element.

```
In [16]: def second_element(t):
             return t[1]
```

Now we can use `sorted` with two keyword arguments:

- `key=second_element` means the items will be sorted according to the frequencies of the words.
- `reverse=True` means they items will be sorted in reverse order, with the most frequent words first.

```
In [17]: items = sorted(word_counter.items(), key=second_element,
reverse=True)
```

Here are the five most frequent words.

```
In [18]: for word, freq in items[:5]:
             print(freq, word, sep='\t')
```

```
Out[18]: 1614    the
         972     and
         941     of
         640     to
         640     i
```

In the next section, we'll encapsulate this loop in a function. And we'll use it to demonstrate a new feature – optional parameters.

Optional parameters

We've used built-in functions that take optional parameters. For example, `round` takes an optional parameters called `ndigits` that indicates how many decimal places to keep.

```
In [19]: round(3.141592653589793, ndigits=3)
```

```
Out[19]: 3.142
```

But it's not just built-in functions – we can write functions with optional parameters, too. For example, the following function takes two parameters, `word_counter` and `num`.

```
In [20]: def print_most_common(word_counter, num=5):
             items = sorted(word_counter.items(),
key=second_element, reverse=True)

             for word, freq in items[:num]:
                 print(freq, word, sep='\t')
```

The second parameter looks like an assignment statement, but it's not – it's an optional parameter.

If you call this function with one argument, `num` gets the **default value**, which is `5`.

```
In [21]: print_most_common(word_counter)
```

```
Out[21]: 1614    the
         972     and
         941     of
         640     to
         640     i
```

If you call this function with two arguments, the second argument gets assigned to `num` instead of the default value.

```
In [22]: print_most_common(word_counter, 3)
```

```
Out[22]: 1614   the
         972    and
         941    of
```

In that case, we would say the optional argument **overrides** the default value.

If a function has both required and optional parameters, all of the required parameters have to come first, followed by the optional ones.

Dictionary subtraction

Suppose we want to spell-check a book – that is, find a list of words that might be misspelled. One way to do that is to find words in the book that don't appear in a list of valid words. In previous chapters, we've used a list of words that are considered valid in word games like Scrabble. Now we'll use this list to spell-check Robert Louis Stevenson.

We can think of this problem as set subtraction – that is, we want to find all the words from one set (the words in the book) that are not in the other (the words in the list).

As we've done before, we can read the contents of `words.txt` and split it into a list of strings.

```
In [23]: word_list = open('words.txt').read().split()
```

The we'll store the words as keys in a dictionary so we can use the `in` operator to check quickly whether a word is valid.

```
In [24]: valid_words = {}
```

```
            for word in word_list:
                valid_words[word] = 1
```

Now, to identify words that appear in the book but not in the word list, we'll use `subtract`, which takes two dictionaries as parameters and returns a new dictionary that contains all the keys from one that are not in the other.

```
In [25]: def subtract(d1, d2):
             res = {}
             for key in d1:
                 if key not in d2:
                     res[key] = d1[key]
             return res
```

Here's how we use it.

```
In [26]: diff = subtract(word_counter, valid_words)
```

To get a sample of words that might be misspelled, we can print the most common words in `diff`.

```
In [27]: print_most_common(diff)
```

```
Out[27]: 640    i
         628    a
         128    utterson
         124    mr
         98     hyde
```

The most common "misspelled" words are mostly names and a few single-letter words (Mr. Utterson is Dr. Jekyll's friend and lawyer).

If we select words that only appear once, they are more likely to be actual misspellings. We can do that by looping through the items and making a list of words with frequency `1`.

```
In [28]: singletons = []
         for word, freq in diff.items():
             if freq == 1:
                 singletons.append(word)
```

Here are the last few elements of the list.

```
In [29]: singletons[-5:]
```

```
Out[29]: ['gesticulated', 'abjection', 'circumscription',
'reindue', 'fearstruck']
```

Most of them are valid words that are not in the word list. But `reindue` appears to be a misspelling of `reinduce`, so at least we found one legitimate error.

Random numbers

As a step toward Markov text generation, next we'll choose a random sequence of words from `word_counter`. But first let's talk about randomness.

Given the same inputs, most computer programs are **deterministic**, which means they generate the same outputs every time. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are one example, but there are more.

Making a program truly nondeterministic turns out to be difficult, but there are ways to fake it. One is to use algorithms that generate **pseudorandom** numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

The `random` module provides functions that generate pseudorandom

numbers – which I will simply call “random” from here on. We can import it like this.

```
In [30]: import random
```

The `random` module provides a function called `choice` that chooses an element from a list at random, with every element having the same probability of being chosen.

```
In [31]: t = [1, 2, 3]
         random.choice(t)
```

```
Out[31]: 1
```

If you call the function again, you might get the same element again, or a different one.

```
In [32]: random.choice(t)
```

```
Out[32]: 2
```

In the long run, we expect to get every element about the same number of times.

If you use `choice` with a dictionary, you get a `KeyError`.

```
In [33]: random.choice(word_counter)
```

```
Out[33]: KeyError: 422
```

To choose a random key, you have to put the keys in a list and then call `choice`.

```
In [34]: words = list(word_counter)
         random.choice(words)
```

```
Out[34]: 'posture'
```

If we generate a random sequence of words, it doesn't make much sense.

```
In [35]: for i in range(6):
             word = random.choice(words)
             print(word, end=' ')
```

```
Out[35]: ill-contained written apocryphal nor busy spoke
```

Part of the problem is that we are not taking into account that some words are more common than others. The results will be better if we choose words with different **weights**, so that some are chosen more often than others.

If we use the values from `word_counter` as weights, each word is chosen with a probability that depends on its frequency.

```
In [36]: weights = word_counter.values()
```

The `random` module provides another function called `choices` that takes weights as an optional argument.

```
In [37]: random.choices(words, weights=weights)
```

```
Out[37]: ['than']
```

And it takes another optional argument, `k`, that specifies the number of words to select.

```
In [38]: random_words = random.choices(words, weights=weights, k=6)
```

```
random_words
```

```
Out[38]: ['reach', 'streets', 'edward', 'a', 'said', 'to']
```

The result is a list of strings that we can join into something that's looks more like a sentence.

```
In [39]: ' '.join(random_words)
```

```
Out[39]: 'reach streets edward a said to'
```

If you choose words from the book at random, you get a sense of the vocabulary, but a series of random words seldom makes sense because there is no relationship between successive words. For example, in a real sentence you expect an article like "the" to be followed by an adjective or a noun, and probably not a verb or adverb. So the next step is to look at these relationships between words.

Bigrams

Instead of looking at one word at a time, now we'll look at sequences of two words, which are called **bigrams**. A sequence of three words is called a **trigram**, and a sequence with some unspecified number of words is called an **n-gram**.

Let's write a program that finds all of the bigrams in the book and the number of times each one appears. To store the results, we'll use a dictionary where

- The keys are tuples of strings that represent bigrams, and
- The values are integers that represent frequencies.

Let's call it `bigram_counter`.

```
In [40]: bigram_counter = {}
```

The following function takes a list of two strings as a parameter. First it makes a tuple of the two strings, which can be used as a key in a dictionary. Then it adds the key to `bigram_counter`, if it doesn't exist, or increments the frequency if it does.

```
In [41]: def count_bigram(bigram):
             key = tuple(bigram)
             if key not in bigram_counter:
                 bigram_counter[key] = 1
             else:
                 bigram_counter[key] += 1
```

As we go through the book, we have to keep track of each pair of consecutive words. So if we see the sequence "man is not truly one", we would add the bigrams `man is`, `is not`, `not truly`, and so on.

To keep track of these bigrams, we'll use a list called `window`, because it is like a window that slides over the pages of the book, showing only two words at a time. Initially, `window` is empty.

```
In [42]: window = []
```

We'll use the following function to process the words one at a time.

```
In [43]: def process_word(word):
             window.append(word)

             if len(window) == 2:
                 count_bigram(window)
                 window.pop(0)
```

The first time this function is called, it appends the given word to `window`. Since there is only one word in the window, we don't have a bigram yet, so the function ends.

The second time it's called – and every time thereafter – it appends a second word to `window`. Since there are two words in the window, it calls `count_bigram` to keep track of how many times each bigram appears. Then it uses `pop` to remove the first word from the window.

The following program loops through the words in the book and processes them one at a time.

```
In [44]: for line in open(filename):
             for word in split_line(line):
                 word = clean_word(word)
                 process_word(word)
```

The result is a dictionary that maps from each bigram to the number of times it appears. We can use `print_most_common` to see the most common bigrams.

```
In [45]: print_most_common(bigram_counter)
```

```
Out[45]: 178    ('of', 'the')
         139    ('in', 'the')
         94     ('it', 'was')
         80     ('and', 'the')
         73     ('to', 'the')
```

Looking at these results, we can get a sense of which pairs of words are most likely to appear together. We can also use the results to generate random text, like this.

```
In [46]: bigrams = list(bigram_counter)
         weights = bigram_counter.values()
         random_bigrams = random.choices(bigrams, weights=weights,
k=6)
```

`bigrams` is a list of the bigrams that appear in the books. `weights` is a list

of their frequencies, so `random_bigrams` is a sample where the probability a bigram is selected is proportional to its frequency.

Here are the results.

```
In [47]: for pair in random_bigrams:
             print(' '.join(pair), end=' ')
```

```
Out[47]: to suggest this preface to detain fact is above all the
laboratory
```

This way of generating text is better than choosing random words, but still doesn't make a lot of sense.

Markov analysis

We can do better with Markov chain text analysis, which computes, for each word in a text, the list of words that come next. As an example, we'll analyze these lyrics from the Monty Python song *Eric, the Half a Bee*:

```
In [48]: song = """
         Half a bee, philosophically,
         Must, ipso facto, half not be.
         But half the bee has got to be
         Vis a vis, its entity. D'you see?
         """
```

To store the results, we'll use a dictionary that maps from each word to the list of words that follow it.

```
In [49]: successor_map = {}
```

As an example, let's start with the first two words of the song.

```
In [50]: first = 'half'
```

```
second = 'a'
```

If the first word is not in `successor_map`, we have to add a new item that maps from the first word to a list containing the second word.

```
In [51]: successor_map[first] = [second]
         successor_map
```

```
Out[51]: {'half': ['a']}
```

If the first word is already in the dictionary, we can look it up to get the list of successors we've seen so far, and append the new one.

```
In [52]: first = 'half'
         second = 'not'

         successor_map[first].append(second)
         successor_map
```

```
Out[52]: {'half': ['a', 'not']}
```

The following function encapsulates these steps.

```
In [53]: def add_bigram(bigram):
             first, second = bigram

             if first not in successor_map:
                 successor_map[first] = [second]
             else:
                 successor_map[first].append(second)
```

If the same bigram appears more that once, the second word is added to the list more than once. In this way, `successor_map` keeps track of how many times each successor appears.

As we did in the previous section, we'll use a list called `window` to store pairs of consecutive words. And we'll use the following function to process the words one at a time.

```
In [54]: def process_word_bigram(word):
             window.append(word)

             if len(window) == 2:
                 add_bigram(window)
                 window.pop(0)
```

Here's how we use it to process the words in the song.

```
In [55]: successor_map = {}
         window = []

         for word in song.split():
             word = clean_word(word)
             process_word_bigram(word)
```

And here are the results.

```
In [56]: successor_map
```

```
Out[56]: {'half': ['a', 'not', 'the'],
          'a': ['bee', 'vis'],
          'bee': ['philosophically', 'has'],
          'philosophically': ['must'],
          'must': ['ipso'],
          'ipso': ['facto'],
          'facto': ['half'],
          'not': ['be'],
          'be': ['but', 'vis'],
          'but': ['half'],
          'the': ['bee'],
          'has': ['got'],
          'got': ['to'],
          'to': ['be'],
          'vis': ['a', 'its'],
          'its': ['entity'],
```

```
 'entity': ["d'you"],
 "d'you": ['see']}
```

The word `half` can be followed by `a`, `not`, or `the`. The word `a` can be followed by `bee` or `vis`. Most of the other words appear only once, so they are followed by only a single word.

Now let's analyze the book.

```
In [57]: successor_map = {}
         window = []

         for line in open(filename):
             for word in split_line(line):
                 word = clean_word(word)
                 process_word_bigram(word)
```

We can look up any word and find the words that can follow it.

```
In [58]: successor_map['going']
```

```
Out[58]: ['east', 'in', 'to', 'to', 'up', 'to', 'of']
```

In this list of successors, notice that the word `to` appears three times – the other successors only appear once.

Generating text

We can use the results from the previous section to generate new text with the same relationships between consecutive words as in the original. Here's how it works:

- Starting with any word that appears in the text, we look up its possible successors and choose one at random.

- Then, using the chosen word, we look up its possible successors, and choose one at random.

We can repeat this process to generate as many words as we want. As an example, let's start with the word `although`. Here are the words that can follow it.

```
In [59]: word = 'although'
         successors = successor_map[word]
         successors
```

```
Out[59]: ['i', 'a', 'it', 'the', 'we', 'they', 'i']
```

Because repeated successors appear more than once, we can use `choice` to choose from the list with equal probability.

```
In [60]: word = random.choice(successors)
         word
```

```
Out[60]: 'i'
```

Repeating these steps, we can use the following loop to generate a longer series.

```
In [61]: for i in range(10):
             successors = successor_map[word]
             word = random.choice(successors)
             print(word, end=' ')
```

```
Out[61]: continue to hesitate and swallowed the smile withered from
that
```

The result sounds more like a real sentence, but it still doesn't make much sense.

We can do better using more than one word as a key in `successor_map`. For example, we can make a dictionary that maps from each bigram, or trigram, to the list of words that can come next. As an exercise, you'll have a chance to implement this analysis and see what the results look like.

Debugging

At this point we are writing more substantial programs, and you might find that you are spending more time debugging. If you are stuck on a difficult bug, there are five things to try:

- Reading: Examine your code, read it back to yourself, and check that it says what you meant to say.
- Running: Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to build scaffolding.
- Ruminating: Take some time to think! What kind of error is it: syntax, runtime, or semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?
- Rubberducking: If you explain the problem to someone else, you sometimes find the answer before you finish asking the question. Often you don't need the other person; you could just talk to a rubber duck. And that's the origin of the well-known strategy called **rubber duck debugging**. I am not making this up – see https://en.wikipedia.org/wiki/Rubber_duck_debugging.
- Retreating: At some point, the best thing to do is back up – undoing recent changes – until you get to a program that works. Then you can start rebuilding.

Beginning programmers sometimes get stuck on one of these activities and

forget the others. Each activity comes with its own failure mode.

For example, reading your code works if the problem is a typographical error, but not if the problem is a conceptual misunderstanding. If you don't understand what your program does, you can read it 100 times and never see the error, because the error is in your head.

Running experiments can work, especially if you run small, simple tests. But if you run experiments without thinking or reading your code, it can take a long time to figure out what's happening.

You have to take time to think. Debugging is like an experimental science. You should have at least one hypothesis about what the problem is. If there are two or more possibilities, try to think of a test that would eliminate one of them.

But even the best debugging techniques will fail if there are too many errors, or if the code you are trying to fix is too big and complicated. Sometimes the best option is to retreat, simplifying the program until you get back to something that works.

Beginning programmers are often reluctant to retreat because they can't stand to delete a line of code (even if it's wrong). If it makes you feel better, copy your program into another file before you start stripping it down. Then you can copy the pieces back one at a time.

Finding a hard bug requires reading, running, ruminating, and sometimes retreating. If you get stuck on one of these activities, try the others.

Exercises

Ask a virtual assistant

In `add_bigram`, the `if` statement creates a new list or appends an element to an existing list, depending on whether the key is already in the dictionary.

```
In [62]: def add_bigram(bigram):
             first, second = bigram
```

```
    if first not in successor_map:
        successor_map[first] = [second]
    else:
        successor_map[first].append(second)
```

Dictionaries provide a method called `setdefault` that we can use to do the same thing more concisely. Ask a virtual assistant how it works, or copy `add_word` into a virtual assistant and ask "Can you rewrite this using `setdefault`?"

In this chapter we implemented Markov chain text analysis and generation. If you are curious, you can ask a virtual assistant for more information on the topic. One of the things you might learn is that virtual assistants use algorithms that are similar in many ways – but also different in important ways. Ask a VA, "What are the differences between large language models like GPT and Markov chain text analysis?"

Exercise

Write a function that counts the number of times each trigram (sequence of three words) appears. If you test your function with the text of *Dr. Jekyll and Mr. Hyde*, you should find that the most common trigram is "said the lawyer".

Hint: Write a function called `count_trigram` that is similar to `count_bigram`. Then write a function called `process_word_trigram` that is similar to `process_word_bigram`.

Exercise

Now let's implement Markov chain text analysis with a mapping from each bigram to a list of possible successors.

Starting with `add_bigram`, write a function called `add_trigram` that takes a list of three words and either adds or updates an item in `successor_map`, using the first two words as the key and the third word as

a possible successor.

Here's a version of `process_word_trigram` that calls `add_trigram`.

```
In [63]: def process_word_trigram(word):
             window.append(word)

             if len(window) == 3:
                 add_trigram(window)
                 window.pop(0)
```

You can use the following loop to test your function with the words from the book.

```
In [64]: successor_map = {}
         window = []

         for line in open(filename):
             for word in split_line(line):
                 word = clean_word(word)
                 process_word_trigram(word)
```

In the next exercise, you'll use the results to generate new random text.

Exercise

For this exercise, we'll assume that `successor_map` is a dictionary that maps from each bigram to the list of words that follow it.

To generate random text, we'll start by choosing a random key from `successor_map`.

```
In [65]: successors = list(successor_map)
         bigram = random.choice(successors)
         bigram
```

```
Out[65]: ('doubted', 'if')
```

Now write a loop that generates `50` more words following these steps:

1. In `successor_map`, look up the list of words that can follow `bigram`.
2. Choose one of them at random and print it.
3. For the next iteration, make a new bigram that contains the second word from `bigram` and the chosen successor.

For example, if we start with the bigram `('doubted', 'if')` and choose `'from'` as its successor, the next bigram is `('if', 'from')`.

If everything is working, you should find that the generated text is recognizably similar in style to the original, and some phrases make sense, but the text might wander from one topic to another.

As a bonus exercise, modify your solution to the last two exercises to use trigrams as keys in `successor_map`, and see what effect it has on the results.

About the Author

Allen Downey is a Staff Producer at Brilliant and Professor Emeritus at Olin College of Engineering. He has taught computer science at Wellesley College, Colby College, and U.C. Berkeley. He has a Ph.D. in Computer Science from U.C. Berkeley and a Master’s Degree from MIT.