You can order print and ebook versions of *Think Python 3e* from Bookshop.org and Amazon.

Work done by Colin Howard starting in Exercises and indicated with boldface text

In [1]:
```
from os.path import basename, exists

def download(url):
    filename = basename(url)
    if not exists(filename):
        from urllib.request import urlretrieve

        local, _ = urlretrieve(url, filename)
        print("Downloaded " + str(local))
    return filename

download('https://github.com/AllenDowney/ThinkPython/raw/v3/thinkpython.py');
download('https://github.com/AllenDowney/ThinkPython/raw/v3/diagram.py');

import thinkpython
```

(chapter_tuples)=

Tuples

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

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

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

In [3]:
```
t = ('l', 'u', 'p', 'i', 'n')
type(t)
```

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

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

In [4]:
```
t1 = 'p',
type(t1)
```

Out[4]:
```
tuple
```

A single value in parentheses is not a tuple.

In [5]:
```
t2 = ('p')
type(t2)
```

Out[5]:
```
str
```

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

In [6]:
```
t = tuple()
t
```

Out[6]:
```
()
```

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

In [7]:
```
t = tuple('lupin')
t
```

Out[7]:
```
('l', 'u', 'p', 'i', 'n')
```

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 [8]:
```
t[0]
```

Out[8]:
```
'l'
```

And the slice operator selects a range of elements.

In [9]:
```
t[1:3]
```

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

The `+` operator concatenates tuples.

In [10]:
```
tuple('lup') + ('i', 'n')
```

Out[10]:
```
('l', 'u', 'p', 'i', 'n')
```

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

In [11]:
```
tuple('spam') * 2
```

Out[11]:
```
('s', 'p', 'a', 'm', 's', 'p', 'a', 'm')
```

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

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

Out[12]:
```
['i', 'l', 'n', 'p', 'u']
```

The `reversed` function also works with tuples.

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

Out[13]:
```
<reversed at 0x7f114c1217b0>
```

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

In [14]:
```
tuple(reversed(t))
```

Out[14]:
```
('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 [15]:
```
t[0] = 'L'
```

```
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 [16]:
```
t.remove('l')
```

```
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 [17]:
```
d = {}
d[1, 2] = 3
d[3, 4] = 7
```

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

In [18]:
```
d[1, 2]
```

Out[18]:
```
3
```

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

In [19]:
```
t = (3, 4)
d[t]
```

Out[19]:
```
7
```

Tuples can also appear as values in a dictionary.

In [2]:
```
t = tuple('abc')
d = {'key': t}
d
```

Out[2]:
```
{'key': ('a', 'b', 'c')}
```

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 [21]:
```
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 [22]:
```
a, b
```

Out[22]:
```
(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 [23]:
```
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 [24]:
```
username, domain
```

Out[24]:
```
('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 [25]:
```
a, b = 1, 2, 3
```

```
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 [26]:
```
temp = a
a = b
b = temp
```

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

In [27]:
```
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 [28]:
```
d = {'one': 1, 'two': 2}

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

```
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 [29]:
```
for key, value in d.items():
    print(key, '->', value)
```

```
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 [30]:
```
divmod(7, 3)
```

Out[30]:
```
(2, 1)
```

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

In [31]:
```
quotient, remainder = divmod(7, 3)
quotient
```

Out[31]:
```
2
```

In [32]:
```
remainder
```

Out[32]:
```
1
```

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

In [33]:
```
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 [34]:
```
min_max([2, 4, 1, 3])
```

Out[34]:
```
(1, 4)
```

We can assign the results to variables like this:

In [35]:
```
low, high = min_max([2, 4, 1, 3])
low, high
```

Out[35]:
```
(1, 4)
```

(section_argument_pack)=

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 [36]:
```
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 [37]:
```
mean(1, 2, 3)
```

Out[37]:
```
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 [38]:
```
t = (7, 3)
divmod(t)
```

```
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 [39]:
```
divmod(*t)
```

Out[39]:
```
(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 [40]:
```
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 [41]:
```
mean(1, 2, 3, 10)
```

Out[41]:
```
4.0
```

In [42]:
```
trimmed_mean(1, 2, 3, 10)
```

Out[42]:
```
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 [43]:
```
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**, so-called because it pairs up the elements of the sequences like the teeth of a zipper.

In [44]:
```
zip(scores1, scores2)
```

Out[44]:
```
<zip at 0x7f1136705b80>
```

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

In [45]:
```
for pair in zip(scores1, scores2):
    print(pair)
```

```
(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 [46]:
```
wins = 0
for team1, team2 in zip(scores1, scores2):
    if team1 > team2:
        wins += 1

wins
```

Out[46]:
```
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 [47]:
```
t = list(zip(scores1, scores2))
t
```

Out[47]:
```
[(1, 5), (2, 5), (4, 2), (5, 2), (1, 5), (5, 2), (2, 3)]
```

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

In [48]:
```
t[-1]
```

Out[48]:
```
(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 [49]:
```
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 [50]:
```
letter_map['a'], letter_map['z']
```

Out[50]:
```
(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 [51]:
```
enumerate('abc')
```

Out[51]:
```
<enumerate at 0x7f114c127640>
```

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 [52]:
```
for index, element in enumerate('abc'):
    print(index, element)
```

```
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 [53]:
```
(0, 1, 2) < (0, 3, 4)
```

Out[53]:
```
True
```

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

In [54]:
```
(0, 1, 2000000) < (0, 3, 4)
```

Out[54]:
```
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 [55]:
```
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 [56]:
```
counter = value_counts('banana')
counter
```

Out[56]:
```
{'b': 1, 'a': 3, 'n': 2}
```

With only three items, 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 [57]:
```
items = counter.items()
items
```

Out[57]:
```
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 [58]:
```
sorted(items)
```

Out[58]:
```
[('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 [59]:
```
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 [60]:
```
sorted_items = sorted(items, key=second_element)
sorted_items
```

Out[60]:
```
[('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 [61]:
```
sorted_items[-1]
```

Out[61]:
```
('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 [62]:
```
max(items, key=second_element)
```

Out[62]:
```
('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 [63]:
```
d =  value_counts('parrot')
d
```

Out[63]:
```
{'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 `'r'`.

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

In [64]:
```
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 [65]:
```
invert_dict(d)
```

Out[65]:
```
{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!

(section_debugging_11)=

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.py.

In [66]:
```
download('https://raw.githubusercontent.com/AllenDowney/ThinkPython/v3/structshape.
```

We can import it like this.

In [67]:
```
from structshape import structshape
```

Here's an example with a simple list.

In [68]:
```
t = [1, 2, 3]
structshape(t)
```

Out[68]:
```
'list of 3 int'
```

Here's a list of lists.

In [69]:
```
t2 = [[1,2], [3,4], [5,6]]
structshape(t2)
```

Out[69]:
```
'list of 3 list of 2 int'
```

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

In [70]:
```
t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
structshape(t3)
```

Out[70]:
```
'list of (3 int, float, 2 str, 2 list of int, int)'
```

Here's a list of tuples.

In [71]:
```
s = 'abc'
lt = list(zip(t, s))
```

```
structshape(lt)
```

```
Out[71]: 'list of 3 tuple of (int, str)'
```

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

```
In [72]: d = dict(lt)
         structshape(d)
```

```
Out[72]: 'dict of 3 int->str'
```

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

Glossary

pack: Collect multiple arguments into a tuple.

unpack: Treat a tuple (or other sequence) as multiple arguments.

zip object: The result of calling the built-in function `zip`, can be used to loop through a sequence of tuples.

enumerate object: The result of calling the built-in function `enumerate`, can be used to loop through a sequence of tuples.

sort key: A value, or function that computes a value, used to sort the elements of a collection.

data structure: A collection of values, organized to perform certain operations efficiently.

Exercises

```
In [73]: # This cell tells Jupyter to provide detailed debugging information
         # when a runtime error occurs. Run it before working on the exercises.

         %xmode Verbose
```

```
Exception reporting mode: Verbose
```

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.

Begin work done by Colin Howard

There doesn't seem like there's anything to actually *do* for this exercise. I will definitely work with a virtual assistant when I need to.

End work done by Colin Howard

Exercise

In this chapter I said that tuples can be used as keys in dictionaries because they are hashable, and they are hashable because they are immutable. But that is not always true.

If a tuple contains a mutable value, like a list or a dictionary, the tuple is no longer hashable because it contains elements that are not hashable. As an example, here's a tuple that contains two lists of integers.

In [74]:
```
list0 = [1, 2, 3]
list1 = [4, 5]

t = (list0, list1)
t
```

Out[74]:
```
([1, 2, 3], [4, 5])
```

Write a line of code that appends the value `6` to the end of the second list in `t`. If you display `t`, the result should be `([1, 2, 3], [4, 5, 6])`.

In [75]:
```

```

Out[75]:
```
([1, 2, 3], [4, 5, 6])
```

Try to create a dictionary that maps from `t` to a string, and confirm that you get a `TypeError`.

In [76]:
```
d = {t: 'this tuple contains two lists'}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[76], line 1
----> 1 d = {t: 'this tuple contains two lists'}
        d = {1: 'a', 2: 'b', 3: 'c'}
        t = ([1, 2, 3], [4, 5, 6])

TypeError: unhashable type: 'list'
```

For more on this topic, ask a virtual assistant, "Are Python tuples always hashable?"

(section_exercise_11)=

Exercise

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

In [2]:
```
letters = 'abcdefghijklmnopqrstuvwxyz'
numbers = range(len(letters))
letter_map = dict(zip(letters, numbers))
```

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

In [3]:
```
letter_map['a']
```

Out[3]:
```
0
```

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

In [4]:
```
letters[1]
```

Out[4]:
```
'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.

You can use this outline to get started.

Begin work done by Colin Howard

In [8]:
```
def shift_word(word, n):
    """Shift the letters of `word` by `n` places.

    >>> shift_word('cheer', 7)
    'jolly'
    >>> shift_word('melon', 16)
    'cubed'
    """
    shifted_list = []
    for letter in word:
        shifted_list.append(letters[(letter_map[letter]+n)%26])
    shifted = ''.join(shifted_list)
    return shifted
```

End work done by Colin Howard (*testing below*)

In [6]:
```
shift_word('cheer', 7)
```

Out[6]:
```
'jolly'
```

In [9]:
```
shift_word('melon', 16)
```

Out[9]:
```
'cubed'
```

You can use `doctest` to test your function.

In [10]:
```
from doctest import run_docstring_examples

def run_doctests(func):
    run_docstring_examples(func, globals(), name=func.__name__)

run_doctests(shift_word)
```

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`.

You can use this outline of the function to get started.

In [85]:
```
def most_frequent_letters(string):

```

```
    return None
```

In [86]:

```
```

And this example to test your function.

In [87]:

```
most_frequent_letters('brontosaurus')
```

```
r 2
o 2
s 2
u 2
b 1
n 1
t 1
a 1
```

Once your function is working, you can use the following code to print the most common letters in *Dracula*, which we can download from Project Gutenberg.

In [88]:

```
download('https://www.gutenberg.org/cache/epub/345/pg345.txt');
```

In [89]:

```
string = open('pg345.txt').read()
most_frequent_letters(string)
```

  156330
e 81324
t 58340
a 52582
o 51637
n 44576
h 42426
s 39637
i 38366
r 36158
d 28683
l 26097
u 18481
w 17436
m 17249

 15869
f 14212
c 13656
y 12826
g 12649
, 11397
p 9173
b 8834
. 8531
k 6310
I 5739
v 5633
- 3918
" 2953
T 1848
; 1683
H 1681
' 1620
A 1225
W 1047
M 1011
_ 995
S 984
x 814
! 752
L 749
* 671
: 650
C 586
D 582
P 547
G 546
q 546
B 494
j 493
? 492
E 454
J 433
V 431
O 425

```
R 388
N 388
Y 344
z 279
F 263
1 162
U 158
2 105
Q 95
K 74
3 67
0 61
X 59
5 49
( 48
) 48
4 45
7 40
9 38
8 36
6 33
’ 25
& 21
{ 12
} 12
è 10
æ 9
ö 9
= 9
> 9
/ 6
Z 5
[ 4
] 4
ë 3
£ 3
é 3
â 2
ï 2
$ 2
 1
# 1
— 1
“ 1
” 1
á 1
ô 1
à 1
% 1
```

According to Zim's "Codes and Secret Writing", the sequence of letters in decreasing order of frequency in English starts with "ETAONRISH". How does this sequence compare with the results from *Dracula*?

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.

The following cells download `words.txt` and read the words into a list.

In [90]:
```
download('https://raw.githubusercontent.com/AllenDowney/ThinkPython/v3/words.txt');
```

In [91]:
```
word_list = open('words.txt').read().split()
```

Here's the `sort_word` function we've used before.

In [92]:
```
def sort_word(word):
    return ''.join(sorted(word))
```

In [93]:
```

```

To find the longest list of anagrams, you can use the following function, which takes a key-value pair where the key is a string and the value is a list of words. It returns the length of the list.

In [94]:
```
def value_length(pair):
    key, value = pair
    return len(value)
```

We can use this function as a sort key to find the longest lists of anagrams.

In [95]:
```
anagram_items = sorted(anagram_dict.items(), key=value_length)
for key, value in anagram_items[-10:]:
    print(value)
```

```
['carets', 'cartes', 'caster', 'caters', 'crates', 'reacts', 'recast', 'traces']
['earings', 'erasing', 'gainers', 'reagins', 'regains', 'reginas', 'searing', 'serin
ga']
['lapse', 'leaps', 'pales', 'peals', 'pleas', 'salep', 'sepal', 'spale']
['palest', 'palets', 'pastel', 'petals', 'plates', 'pleats', 'septal', 'staple']
['peris', 'piers', 'pries', 'prise', 'ripes', 'speir', 'spier', 'spire']
['capers', 'crapes', 'escarp', 'pacers', 'parsec', 'recaps', 'scrape', 'secpar', 'sp
acer']
['estrin', 'inerts', 'insert', 'inters', 'niters', 'nitres', 'sinter', 'triens', 'tr
ines']
['least', 'setal', 'slate', 'stale', 'steal', 'stela', 'taels', 'tales', 'teals', 't
esla']
['alerts', 'alters', 'artels', 'estral', 'laster', 'ratels', 'salter', 'slater', 'st
aler', 'stelar', 'talers']
['apers', 'asper', 'pares', 'parse', 'pears', 'prase', 'presa', 'rapes', 'reaps', 's
pare', 'spear']
```

If you want to know the longest words that have anagrams, you can use the following loop to print some of them.

In [96]:
```
longest = 7

for key, value in anagram_items:
    if len(value) > 1:
        word_len = len(value[0])
        if word_len > longest:
            longest = word_len
            print(value)
```

```
['abasement', 'entamebas']
['abreacting', 'acerbating']
['altercations', 'intercoastal']
['certification', 'rectification']
['certifications', 'rectifications']
['impressivenesses', 'permissivenesses']
```

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.

Here's an outline of the function with doctests you can use to check your function.

In [97]:
```
def word_distance(word1, word2):
    """Computes the number of places where two word differ.

    >>> word_distance("hello", "hxllo")
    1
    >>> word_distance("ample", "apply")
    2
    >>> word_distance("kitten", "mutton")
    3
```

```
    """
    return None
```

In [98]:

In [99]:
```
from doctest import run_docstring_examples

def run_doctests(func):
    run_docstring_examples(func, globals(), name=func.__name__)

run_doctests(word_distance)
```

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.

In [100…

```
aerologies areologies
antimonies antinomies
bedrugging begrudging
certification rectification
certifications rectifications
certifiers rectifiers
certifying rectifying
certitudes rectitudes
concerting concreting
conservation conversation
conservations conversations
conserving conversing
deification edification
deifications edifications
denotation detonation
denotations detonations
discanting distancing
dissention distension
dissentions distensions
dormancies mordancies
livenesses vilenesses
frumenties furmenties
garmenting margenting
lamenesses malenesses
limestones milestones
misdealing misleading
molarities moralities
monarchies nomarchies
monologies nomologies
performing preforming
portending protending
presetting pretesting
questionnaire questionniare
questionnaires questionniares
regelating relegating
repertoires repertories
rowdinesses wordinesses
sepulchers sepulchres
solitaires solitaries
spotlights stoplights
```

Exercise

This is a bonus exercise that is not in the book. It is more challenging than the other exercises in this chapter, so you might want to ask a virtual assistant for help, or come back to it after you've read a few more chapters.

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.

In [101…

In [102…

In [103…

In [104…

In [105…

```
wranglers wanglers anglers angers agers ages age ae a
wrappings wrapping rapping raping aping ping pig pi i
carrousels carousels carouses arouses arouse arose arse are ae a
completing competing compting comping coping oping ping pig pi i
insolating isolating solating slating sating sting ting tin in i
restarting restating estating stating sating sting ting tin in i
staunchest stanchest stanches stances stanes sanes anes ane ae a
stranglers strangers stranger strange strang stang tang tag ta a
twitchiest witchiest withiest withies withes wites wits its is i
complecting completing competing compting comping coping oping ping pig pi i
```

In [ ]:

Think Python: 3rd Edition