The language feature most often associated with object-oriented programming is inheritance. Inheritance is the ability to define a new class that is a modified version of an existing class. In this chapter I demonstrate inheritance using classes that represent playing cards, decks of cards, and poker hands.

There are fifty-two cards in a deck, each of which belongs to one of four suits and one of thirteen ranks. The suits are Spades, Hearts, Diamonds, and Clubs (in descending order in bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. Depending on the game that you are playing, an Ace may be higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is obvious what the attributes should be: rank and suit. It is not as obvious what type the attributes should be. One possibility is to use strings containing words like 'Spade' for suits and 'Queen' for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had a higher rank or suit.

An alternative is to use integers to encode the ranks and suits. In this context, “encode” means that we are going to define a mapping between numbers and suits, or between numbers and ranks. This kind of encoding is not meant to be a secret (that would be “encryption”).

For example, this table shows the suits and the corresponding integer codes:
Spades --> 3

Hearts --> 2

Diamonds --> 1

Clubs --> 0

This code makes it easy to compare cards; because higher suits map to higher numbers, we can compare suits by comparing their codes.

The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

Jack --> 11

Queen --> 12

King --> 13

In [1]:
```
class Card:
    """Represents a standard playing card."""
    def __init__(self, suit=0, rank=2):
```

```
        self.suit = suit
        self.rank = rank
```

In [2]:
```
queen_of_diamonds = Card(1, 12)
```

18.2 Class attributes

In order to print Card objects in a way that people can easily read, we need a mapping from the integer codes to the corresponding ranks and suits. A natural way to do that is with lists of strings. We assign these lists to class attributes:

In [3]:
```
class Card:
    """Represents a standard playing card."""
    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7','8', '9', '10', 'Jack',
    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank],Card.suit_names[self.suit])
```

Variables like suit_names and rank_names, which are defined inside a class but outside of any method, are called class attributes because they are associated with the class object Card.

This term distinguishes them from variables like suit and rank, which are called instance attributes because they are associated with a particular instance.

Both kinds of attribute are accessed using dot notation. For example, in __str__, self is a Card object, and self.rank is its rank. Similarly, Card is a class object, and Card.rank_names is a list of strings associated with the class.

Every card has its own suit and rank, but there is only one copy of suit_names and rank_names.

The first element of rank_names is None because there is no card with rank zero. By including None as a place-keeper, we get a mapping with the nice property that the index 2 maps to the string '2', and so on. To avoid this tweak, we could have used a dictionary instead of a list.

In [4]:
```
card1 = Card(2, 11)
print(card1)
```

```
Jack of Hearts
```

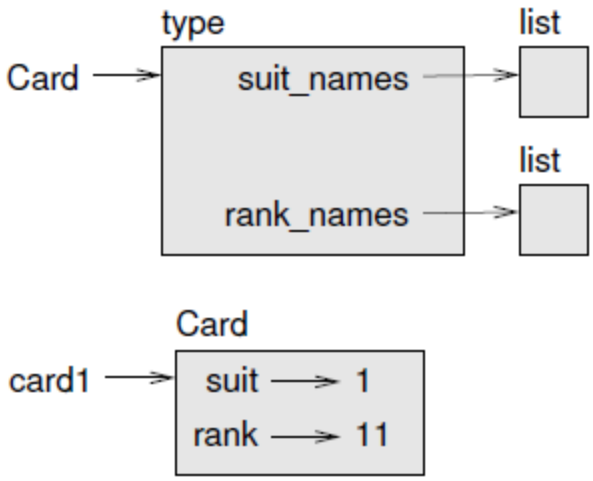

Figure 18.1: Object diagram.

18.3 Comparing cards

For built-in types, there are relational operators (<, >, ==, etc.) that compare values and determine when one is greater than, less than, or equal to another. For programmer-defined types, we can override the behavior of the built-in operators by providing a method named __lt__, which stands for "less than".
__lt__ takes two parameters, self and other, and returns True if self is strictly less than other.

The correct ordering for cards is not obvious. For example, which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit. In order to compare cards, you have to decide whether rank or suit is more important.

The answer might depend on what game you are playing, but to keep things simple, we'll make the arbitrary choice that suit is more important, so all of the Spades outrank all of the Diamonds, and so on.

In [5]:
```
class Card:
    """Represents a standard playing card."""
    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7','8', '9', '10', 'Jack',
    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank],Card.suit_names[self.suit])
    def __lt__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 < t2
```

18.4 Decks

Now that we have Cards, the next step is to define Decks. Since a deck is made up of cards, it is natural for each Deck to contain a list of cards as an attribute.

The following is a class definition for Deck. The init method creates the attribute cards and generates the standard set of fifty-two cards:

In [6]:
```
class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

18.5 Printing the deck

In [7]:
```
class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

In [8]:
```
deck = Deck()
print(deck)
```

```
Ace of Clubs
2 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
6 of Clubs
7 of Clubs
8 of Clubs
9 of Clubs
10 of Clubs
Jack of Clubs
Queen of Clubs
King of Clubs
Ace of Diamonds
2 of Diamonds
3 of Diamonds
4 of Diamonds
5 of Diamonds
6 of Diamonds
7 of Diamonds
8 of Diamonds
9 of Diamonds
10 of Diamonds
Jack of Diamonds
Queen of Diamonds
King of Diamonds
Ace of Hearts
2 of Hearts
3 of Hearts
4 of Hearts
5 of Hearts
6 of Hearts
7 of Hearts
8 of Hearts
9 of Hearts
10 of Hearts
Jack of Hearts
Queen of Hearts
King of Hearts
Ace of Spades
2 of Spades
3 of Spades
4 of Spades
5 of Spades
6 of Spades
7 of Spades
8 of Spades
9 of Spades
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

Even though the result appears on 52 lines, it is one long string that contains newlines.

18.6 Add, remove, shuffle and sort

To deal cards, we would like a method that removes a card from the deck and returns it. The list method pop provides a convenient way to do that:

Since pop removes the last card in the list, we are dealing from the bottom of the deck.

A method like this that uses another method without doing much work is sometimes called a veneer. The metaphor comes from woodworking, where a veneer is a thin layer of good quality wood glued to the surface of a cheaper piece of wood to improve the appearance.

In this case add_card is a "thin" method that expresses a list operation in terms appropriate for decks. It improves the appearance, or interface, of the implementation.

As another example, we can write a Deck method named shuffle using the function shuffle from the random module:

In [10]:
```
import random
class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
    def pop_card(self):
        return self.cards.pop()
    def add_card(self, card):
        self.cards.append(card)
    def shuffle(self):
        random.shuffle(self.cards)
```

18.7 Inheritance

Inheritance is the ability to define a new class that is a modified version of an existing class. As an example, let's say we want a class to represent a "hand", that is, the cards held by one player. A hand is similar to a deck: both are made up of a collection of cards, and both require operations like adding and removing cards.

A hand is also different from a deck; there are operations we want for hands that don't make sense for a deck. For example, in poker we might compare two hands to see which one wins. In bridge, we might compute a score for a hand in order to make a bid.

This relationship between classes—similar, but different—lends itself to inheritance. To define a new class that inherits from an existing class, you put the name of the existing class in parentheses:

In [11]:
```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

This definition indicates that Hand inherits from Deck; that means we can use methods like pop_card and add_card for Hands as well as Decks.

When a new class inherits from an existing one, the existing one is called the parent and the new class is called the child.

In this example, Hand inherits __init__ from Deck, but it doesn't really do what we want: instead of populating the hand with 52 new cards, the init method for Hands should initialize cards with an empty list.

If we provide an init method in the Hand class, it overrides the one in the Deck class:

In [12]:
```
class Hand(Deck):
    """Represents a hand of playing cards."""
    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

When you create a Hand, Python invokes this init method, not the one in Deck.

In [13]:
```
hand = Hand('new hand')
hand.cards
```

Out[13]:
```
[]
```

In [14]:
```
hand.label
```

Out[14]:
```
'new hand'
```

The other methods are inherited from Deck, so we can use pop_card and add_card to deal a card:

In [15]:
```
deck = Deck()
card = deck.pop_card()
hand.add_card(card)
print(hand)
```

```
King of Spades
```

A natural next step is to encapsulate this code in a method called move_cards:

In [16]:
```
import random
class Deck:
```

```
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
    def pop_card(self):
        return self.cards.pop()
    def add_card(self, card):
        self.cards.append(card)
    def shuffle(self):
        random.shuffle(self.cards)
    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

move_cards takes two arguments, a Hand object and the number of cards to deal. It modifies both self and hand, and returns None.

In some games, cards are moved from one hand to another, or from a hand back to the deck. You can use move_cards for any of these operations: self can be either a Deck or a Hand, and hand, despite the name, can also be a Deck

Inheritance is a useful feature. Some programs that would be repetitive without inheritance can be written more elegantly with it. Inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them. In some cases, the inheritance structure reflects the natural structure of the problem, which makes the design easier to understand.

On the other hand, inheritance can make programs difficult to read. When a method is invoked, it is sometimes not clear where to find its definition. The relevant code may be spread across several modules. Also, many of the things that can be done using inheritance can be done as well or better without it.

18.8 Class diagrams

A class diagram is a more abstract representation of the structure of a program. Instead of showing individual objects, it shows classes and the relationships between them.

There are several kinds of relationship between classes:
• Objects in one class might contain references to objects in another class. For example, each Rectangle contains a reference to a Point, and each Deck contains references to many Cards. This kind of relationship is called HAS-A, as in, “a Rectangle has a Point.”

• One class might inherit from another. This relationship is called IS-A, as in, “a Hand is a kind of a Deck.”

• One class might depend on another in the sense that objects in one class take objects in the second class as parameters, or use objects in the second class as part of a computation. This kind of relationship is called a dependency.

A class diagram is a graphical representation of these relationships. For example, Figure 18.2 shows the relationships between Card, Deck and Hand.

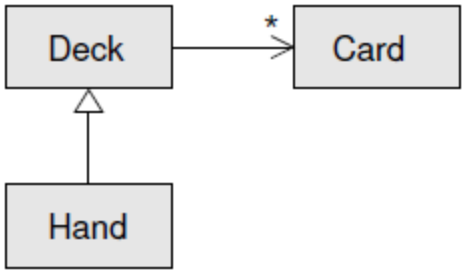

Figure 18.2: Class diagram.

The arrow with a hollow triangle head represents an IS-A relationship; in this case it indicates that Hand inherits from Deck.
The standard arrow head represents a HAS-A relationship; in this case a Deck has references to Card objects.

The star (*) near the arrow head is a multiplicity; it indicates how many Cards a Deck has. A multiplicity can be a simple number, like 52, a range, like 5..7 or a star, which indicates that a Deck can have any number of Cards.

There are no dependencies in this diagram. They would normally be shown with a dashed arrow. Or if there are a lot of dependencies, they are sometimes omitted.
A more detailed diagram might show that a Deck actually contains a list of Cards, but built-in types like list and dict are usually not included in class diagrams.

18.10 Data encapsulation

The previous chapters demonstrate a development plan we might call “object-oriented design”. We identified objects we needed—like Point, Rectangle and Time—and defined classes to represent them. In each case there is an obvious correspondence between the object and some entity in the real world (or at least a mathematical world).
But sometimes it is less obvious what objects you need and how they should interact. In that case you need a different development plan. In the same way that we discovered function

interfaces by encapsulation and generalization, we can discover class interfaces by data encapsulation.

In [3]:
```
"""This module contains a code example related to

Think Python, 2nd Edition
by Allen Downey
http://thinkpython2.com

Copyright 2015 Allen Downey

License: http://creativecommons.org/licenses/by/4.0/
"""

from __future__ import print_function, division

import sys
import string
import random

# global variables
suffix_map = {}        # map from prefixes to a list of suffixes
prefix = ()            # current tuple of words


def process_file(filename, order=2):
    """Reads a file and performs Markov analysis.

    filename: string
    order: integer number of words in the prefix

    returns: map from prefix to list of possible suffixes.
    """
    fp = open(filename)
    skip_gutenberg_header(fp)

    for line in fp:
        if line.startswith('*** END OF THIS'):
            break

        for word in line.rstrip().split():
            process_word(word, order)


def skip_gutenberg_header(fp):
    """Reads from fp until it finds the line that ends the header.

    fp: open file object
    """
    for line in fp:
        if line.startswith('*** START OF THIS'):
            break


def process_word(word, order=2):
```

```
    """Processes each word.

    word: string
    order: integer

    During the first few iterations, all we do is store up the words;
    after that we start adding entries to the dictionary.
    """
    global prefix
    if len(prefix) < order:
        prefix += (word,)
        return

    try:
        suffix_map[prefix].append(word)
    except KeyError:
        # if there is no entry for this prefix, make one
        suffix_map[prefix] = [word]

    prefix = shift(prefix, word)


def random_text(n=100):
    """Generates random wordsfrom the analyzed text.

    Starts with a random prefix from the dictionary.

    n: number of words to generate
    """
    # choose a random prefix (not weighted by frequency)
    start = random.choice(list(suffix_map.keys()))

    for i in range(n):
        suffixes = suffix_map.get(start, None)
        if suffixes == None:
            # if the start isn't in map, we got to the end of the
            # original text, so we have to start again.
            random_text(n-i)
            return

        # choose a random suffix
        word = random.choice(suffixes)
        print(word, end=' ')
        start = shift(start, word)


def shift(t, word):
    """Forms a new tuple by removing the head and adding word to the tail.

    t: tuple of strings
    word: string

    Returns: tuple of strings
    """
    return t[1:] + (word,)
```

```
def main(script, filename='158-0.txt', n=100, order=2):
    try:
        #n = int(n)
        order = int(order)
    except ValueError:
        print('Usage: %d filename [# of words] [prefix length]' % script)
    else:
        process_file(filename, order)
        random_text(n)
        print()

if __name__ == '__main__':
    main(*sys.argv)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-3-d3f32f8b891a> in <module>
    120
    121 if __name__ == '__main__':
--> 122     main(*sys.argv)

<ipython-input-3-d3f32f8b891a> in main(script, filename, n, order)
    115         print('Usage: %d filename [# of words] [prefix length]' % script)
    116     else:
--> 117         process_file(filename, order)
    118         random_text(n)
    119         print()

<ipython-input-3-d3f32f8b891a> in process_file(filename, order)
     29     returns: map from prefix to list of possible suffixes.
     30     """
---> 31     fp = open(filename)
     32     skip_gutenberg_header(fp)
     33

FileNotFoundError: [Errno 2] No such file or directory: '-f'
```

you'll see that it uses two global variables—suffix_map and prefix—that are read and written from several functions.

suffix_map = {}

prefix = ()

Because these variables are global, we can only run one analysis at a time. If we read two texts, their prefixes and suffixes would be added to the same data structures (which makes for some interesting generated text).

To run multiple analyses, and keep them separate, we can encapsulate the state of each analysis in an object. Here's what that looks like:

In [8]:
```
class Markov:
    def __init__(self):
```

```
        self.suffix_map = {}
        self.prefix = ()
    def process_word(self, word, order=2):
        if len(self.prefix) < order:
            self.prefix += (word,)
            return
        try:
            self.suffix_map[self.prefix].append(word)
        except KeyError:
            # if there is no entry for this prefix, make one
            self.suffix_map[self.prefix] = [word]
        self.prefix = shift(self.prefix, word)
```

Next, we transform the functions into methods. For example, here's process_word:

Transforming a program like this—changing the design without changing the behavior—is another example of refactoring

This example suggests a development plan for designing objects and methods:

1. Start by writing functions that read and write global variables (when necessary).
2. Once you get the program working, look for associations between global variables and the functions that use them.
3. Encapsulate related variables as attributes of an object.
4. Transform the associated functions into methods of the new class.

As an exercise, download my Markov code from https://thinkpython.com/code/ markov.py, and follow the steps described above to encapsulate the global variables as attributes of a new class called Markov. Solution: https://thinkpython.com/code/ markov2.py.

In [9]:
```
"""This module contains a code example related to

Think Python, 2nd Edition
by Allen Downey
http://thinkpython2.com

Copyright 2015 Allen Downey

License: http://creativecommons.org/licenses/by/4.0/
"""

from __future__ import print_function, division


import sys
import random

from markov import skip_gutenberg_header, shift
```

```
class Markov:
    """Encapsulates the statistical summary of a text."""

    def __init__(self):
        self.suffix_map = {}        # map from prefixes to a list of suffixes
        self.prefix = ()            # current tuple of words

    def process_file(self, filename, order=2):
        """Reads a file and performs Markov analysis.

        filename: string
        order: integer number of words in the prefix

        Returns: map from prefix to list of possible suffixes.
        """
        fp = open(filename)
        skip_gutenberg_header(fp)

        for line in fp:
            if line.startswith('*** END OF THIS'):
                break

            for word in line.rstrip().split():
                self.process_word(word, order)

    def process_word(self, word, order=2):
        """Processes each word.

        word: string
        order: integer

        During the first few iterations, all we do is store up the words;
        after that we start adding entries to the dictionary.
        """
        if len(self.prefix) < order:
            self.prefix += (word,)
            return

        try:
            self.suffix_map[self.prefix].append(word)
        except KeyError:
            # if there is no entry for this prefix, make one
            self.suffix_map[self.prefix] = [word]

        self.prefix = shift(self.prefix, word)

    def random_text(self, n=100):
        """Generates random wordsfrom the analyzed text.

        Starts with a random prefix from the dictionary.

        n: number of words to generate
        """
        # choose a random prefix (not weighted by frequency)
        start = random.choice(list(self.suffix_map.keys()))
```

```
        for i in range(n):
            suffixes = self.suffix_map.get(start, None)
            if suffixes == None:
                # if the prefix isn't in map, we got to the end of the
                # original text, so we have to start again.
                self.random_text(n-i)
                return

            # choose a random suffix
            word = random.choice(suffixes)
            print(word, end=' ')
            start = shift(start, word)


def main(script, filename='158-0.txt', n=100, order=2):
    try:
        n = int(n)
        order = int(order)
    except ValueError:
        print('Usage: %d filename [# of words] [prefix length]' % script)
    else:
        markov = Markov()
        markov.process_file(filename, order)
        markov.random_text(n)


if __name__ == '__main__':
    main(*sys.argv)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-9-4eee6bbf3b82> in <module>
     16 import random
     17
---> 18 from markov import skip_gutenberg_header, shift
     19
     20

ModuleNotFoundError: No module named 'markov'
```

In [11]:
```
x=-3
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

In [12]:
```
y
```

Out[12]:
```
nan
```

In [13]:
```
y = math.log(x) if x > 0 else float('nan')
```

List comprehensions are concise and easy to read, at least for simple expressions. And they are usually faster than the equivalent for loops, sometimes much faster. So if you are mad at

me for not mentioning them earlier, I understand.

But, in my defense, list comprehensions are harder to debug because you can't put a print statement inside the loop. I suggest that you use them only if the computation is simple enough that you are likely to get it right the first time. And for beginners that means never.

Generator expressions are similar to list comprehensions, but with parentheses instead of square brackets:

In [14]:
```
g = (x**2 for x in range(5))
g
```

Out[14]:
```
<generator object <genexpr> at 0x0000013E6D533AF0>
```

The result is a generator object that knows how to iterate through a sequence of values. But unlike a list comprehension, it does not compute the values all at once; it waits to be asked. The built-in function next gets the next value from the generator:

In [15]:
```
next(g)
```

Out[15]:
```
0
```

In [16]:
```
next(g)
```

Out[16]:
```
1
```

When you get to the end of the sequence, next raises a StopIteration exception. You can also use a for loop to iterate through the values:

In [17]:
```
for val in g:
    print(val)
```

```
4
9
16
```

The generator object keeps track of where it is in the sequence, so the for loop picks up where next left off. Once the generator is exhausted, it continues to raise StopIteration:

In [18]:
```
next(g)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-18-e734f8aca5ac> in <module>
----> 1 next(g)

StopIteration:
```

In [19]:
```
#Generator expressions are often used with functions like sum, max, and min:
sum(x**2 for x in range(5))
```

Out[19]:
```
30
```

In [20]:
```
any([False, False, True])
```

Out[20]:
```
True
```

But it is often used with generator expressions:

In [21]:
```
any(letter == 't' for letter in 'monty')
```

Out[21]:
```
True
```

In [22]:
```
def avoids(word, forbidden):
    return not any(letter in forbidden for letter in word)
```

The function almost reads like English, “word avoids forbidden if there are not any forbidden letters in word.”

Using any with a generator expression is efficient because it stops immediately if it finds a True value, so it doesn’t have to evaluate the whole sequence.

Adding elements to a set is fast; so is checking membership.

A Counter is like a set, except that if an element appears more than once, the Counter keeps track of how many times it appears. If you are familiar with the mathematical idea of a multiset, a Counter is a natural way to represent a multiset.

Counter is defined in a standard module called collections, so you have to import it. You can initialize a Counter with a string, list, or anything else that supports iteration:

In [23]:
```
from collections import Counter
count = Counter('parrot')
count
```

Out[23]:
```
Counter({'p': 1, 'a': 1, 'r': 2, 'o': 1, 't': 1})
```

Counters behave like dictionaries in many ways; they map from each key to the number of times it appears. As in dictionaries, the keys have to be hashable.

Unlike dictionaries, Counters don’t raise an exception if you access an element that doesn’t appear. Instead, they return 0:

In [24]:
```
count['d']
```

Out[24]:
```
0
```

In [25]:
```
def is_anagram(word1, word2):
    return Counter(word1) == Counter(word2)
```

If two words are anagrams, they contain the same letters with the same counts, so their Counters are equivalent.

Counters provide methods and operators to perform set-like operations, including addition, subtraction, union and intersection. And they provide an often-useful method, most_common, which returns a list of value-frequency pairs, sorted from most common to least:

In [26]:
```
count = Counter('parrot')
for val, freq in count.most_common(3):
    print(val, freq)
```

```
r 2
p 1
a 1
```

The collections module also provides defaultdict, which is like a dictionary except that if you access a key that doesn't exist, it can generate a new value on the fly.

When you create a defaultdict, you provide a function that's used to create new values. A function used to create objects is sometimes called a factory. The built-in functions that create lists, sets, and other types can be used as factories:

In [27]:
```
from collections import defaultdict
d = defaultdict(list)
```

In [28]:
```
d
```

Out[28]:
```
defaultdict(list, {})
```

Notice that the argument is list, which is a class object, not list(), which is a new list. The function you provide doesn't get called unless you access a key that doesn't exist.

In [29]:
```
t = d['new key']
t
```

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

The new list, which we're calling t, is also added to the dictionary. So if we modify t, the change appears in d:

In [30]:
```
t.append('new value')
d
```

Out[30]:
```
defaultdict(list, {'new key': ['new value']})
```

In [31]:
```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
```

```
        t = signature(word)
        d.setdefault(t, []).append(word)
    return d
```

This solution has the drawback that it makes a new list every time, regardless of whether it is needed. For lists, that's no big deal, but if the factory function is complicated, it might be.

We can avoid this problem and simplify the code using a defaultdict:

In [32]:
```
def all_anagrams(filename):
    d = defaultdict(list)
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d[t].append(word)
    return d
```

In [33]:
```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

In [34]:
```
Point
```

Out[34]:
```
__main__.Point
```

Point automatically provides methods like __init__ and __str__ so you don't have to write them.

To create a Point object, you use the Point class as a function:

In [35]:
```
p = Point(1, 2)
p
```

Out[35]:
```
Point(x=1, y=2)
```

You can access the elements of the named tuple by name

In [36]:
```
p.x, p.y
```

Out[36]:
```
(1, 2)
```

In [37]:
```
p[0], p[1]
```

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

In [38]:
```
x, y = p
x, y
```

Out[38]:
```
(1, 2)
```

Named tuples provide a quick way to define simple classes. The drawback is that simple classes don't always stay simple. You might decide later that you want to add methods to a

named tuple. In that case, you could define a new class that inherits from the named tuple:

In [40]:
```
class Pointier(Point):
    # add more methods here
    pass
```

In [41]:
```
def printall(*args):
    print(args)
```

You can call this function with any number of positional arguments (that is, arguments that don't have keywords):

In [42]:
```
printall(1, 2.0, '3')
```

```
(1, 2.0, '3')
```

But the * operator doesn't gather keyword arguments:

In [43]:
```
printall(1, 2.0, third='3')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-43-af1432a881b7> in <module>
----> 1 printall(1, 2.0, third='3')

TypeError: printall() got an unexpected keyword argument 'third'
```

To gather keyword arguments, you can use the ** operator:

In [44]:
```
def printall(*args, **kwargs):
    print(args, kwargs)
```

You can call the keyword gathering parameter anything you want, but kwargs is a common choice. The result is a dictionary that maps from keywords to values:

In [45]:
```
printall(1, 2.0, third='3')
```

```
(1, 2.0) {'third': '3'}
```

If you have a dictionary of keywords and values, you can use the scatter operator, ** to call a function:

In [46]:
```
d = dict(x=1, y=2)
Point(**d)
```

Out[46]:
```
Point(x=1, y=2)
```

Without the scatter operator, the function would treat d as a single positional argument, so it would assign d to x and complain because there's nothing to assign to y:

In [47]:
```
d = dict(x=1, y=2)
Point(d)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-47-7a515be11c9c> in <module>
      1 d = dict(x=1, y=2)
----> 2 Point(d)

TypeError: __new__() missing 1 required positional argument: 'y'
```

When you are working with functions that have a large number of parameters, it is often useful to create and pass around dictionaries that specify frequently used options.

In [ ]: