UNIT-V

Classes and Functions

Time

As another example of a programmer-defined type, we'll define a class called Time
that records the time of day. The class definition looks like this:

```
class Time:
    """Represents the time of day.

    attributes: hour, minute,
    second """
```

We can create a new Time object and assign attributes for hours, minutes, and seconds:

```
time =
Time()
time.hour =
11
time.minute = 59
time.second = 30
```

The state diagram for the Timeobject looks like Figure 16-1.

As an exercise, write a function called print_timethat takes a Time object and prints it in the form hour:minute:second. Hint: the format sequence '%.2d'prints an integer using at least two digits, including a leading zero if necessary.

Write a boolean function called is_afterthat takes two Time objects, t1and t2, and returns True if t1 follows t2 chronologically and False otherwise. Challenge: don't use an ifstatement.

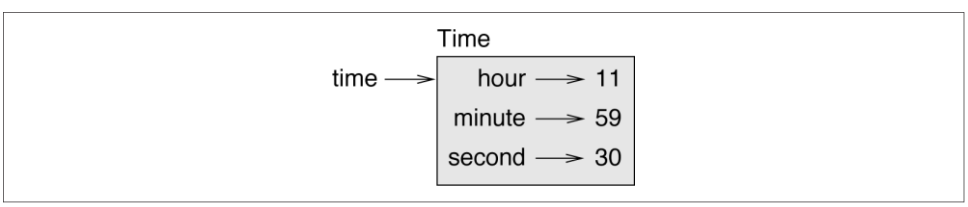

Figure 16-1. Object diagram.

Pure Functions

In the next few sections, we'll write two functions that add time values. They

demon- strate two kinds of functions: pure functions and modifiers. They also demonstrate a development plan I'll call **prototype and patch**, which is a way of tackling a complex problem by starting with a simple prototype and incrementally dealing with the com- plications.

Here is a simple prototype of add_time:

```
def add_time(t1,
    t2): sum =
    Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute +
    t2.minute sum.second =
    t1.second + t2.second return
    sum
```

The function creates a new Time object, initializes its attributes, and returns a reference to the new object. This is called a **pure function** because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value.

To test this function, I'll create two Time objects: start contains the start time of a movie, like *Monty Python and the Holy Grail*, and duration contains the runtime of the movie, which is 1 hour 35 minutes.

add_time figures out when the movie will be done:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second =  0
>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>>
print_time(done)
10:80:00
```

The result, 10:80:00, might not be what you were hoping for. The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty. When that happens, we have to "carry" the extra seconds into the minute column or the extra minutes into the hour column.

Here's an improved version:

```
def add_time(t1,
    t2): sum =
    Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute +
    t2.minute sum.second =
    t1.second + t2.second

    if sum.second >=
        60: sum.second
        -= 60
        sum.minute += 1

    if sum.minute >=
        60: sum.minute
        -= 60
        sum.hour += 1

    return sum
```

Although this function is correct, it is starting to get big. We will see a shorter alterna- tive later.

Modifiers

Sometimes it is useful for a function to modify the objects it gets as parameters. In that case, the changes are visible to the caller. Functions that work this way are called **modifiers**.

increment, which adds a given number of seconds to a Time object, can be written naturally as a modifier. Here is a rough draft:

```
def increment(time,
    seconds): time.second
    += seconds

    if time.second >=
        60: time.second
        -= 60
        time.minute += 1
```

```
if time.minute >=
    60:
    time.minute -=
    60
    time.hour += 1
```

The first line performs the basic operation; the remainder deals with the special cases we saw before.

Is this function correct? What happens if seconds is much greater than 60?

In that case, it is not enough to carry once; we have to keep doing it until time.second is less than 60. One solution is to replace the if statements with while statements. That would make the function correct, but not very efficient. As an exercise, write a correct version of increment that doesn't contain any loops.

Anything that can be done with modifiers can also be done with pure functions. In fact, some programming languages only allow pure functions. There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. But modifiers are convenient at times, and functional programs tend to be less efficient.

In general, I recommend that you write pure functions whenever it is reasonable and resort to modifiers only if there is a compelling advantage. This approach might be called a **functional programming style**.

As an exercise, write a "pure" version of increment that creates and returns a new Time object rather than modifying the parameter.

Prototyping versus Planning

The development plan I am demonstrating is called "prototype and patch". For each function, I wrote a prototype that performed the basic calculation and then tested it, patching errors along the way.

This approach can be effective, especially if you don't yet have a deep understanding of the problem. But incremental corrections can generate code that is unnecessarily complicated (since it deals with many special cases) and unreliable (since it is hard to know if you have found all the errors).

An alternative is **designed development**, in which high-level insight into the problem can make the programming much easier. In this case, the insight is that a Time object is really a three-digit number in base 60 (see *http://en.wikipedia.org/wiki/Sexagesimal*.)! The second attribute is the "ones column", the minute attribute is the "sixties column", and the hour attribute is the

"thirty-six hundreds column".

When we wrote add_time and increment, we were effectively doing addition in base 60, which is why we had to carry from one column to the next.

This observation suggests another approach to the whole problem—we can convert Time objects to integers and take advantage of the fact that the computer knows how to do integer arithmetic.

Here is a function that converts Times to integers:

```
def time_to_int(time):
    minutes = time.hour * 60 +
    time.minute seconds = minutes *
    60 + time.second return seconds
```

And here is a function that converts an integer to a Time (recall that divmod divides the first argument by the second and returns the quotient and remainder as a tuple):

```
def
    int_to_time(secon
    ds): time = Time()
    minutes, time.second = divmod(seconds,
    60) time.hour, time.minute =
    divmod(minutes, 60) return time
```

You might have to think a bit, and run some tests, to convince yourself that these functions are correct. One way to test them is to check that time_to_int(int_to_time(x)) == x for many values of x. This is an example of a consistency check.

Once you are convinced they are correct, you can use them to rewrite add_time:

```
def add_time(t1, t2):
    seconds = time_to_int(t1) +
    time_to_int(t2) return
    int_to_time(seconds)
```

This version is shorter than the original, and easier to verify. As an exercise, rewrite increment using time_to_int and int_to_time.

In some ways, converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with time val- ues is better.

But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversion functions (time_to_int and int_to_time), we get a

pro- gram that is shorter, easier to read and debug, and more reliable.

It is also easier to add features later. For example, imagine subtracting two Times to find the duration between them. The naive approach would be to implement subtrac- tion with borrowing. Using the conversion functions would be easier and more likely to be correct.

Ironically, sometimes making a problem harder (or more general) makes it easier (because there are fewer special cases and fewer opportunities for error).

Debugging

A Time object is well-formed if the values of minute and second are between 0 and 60 (including 0 but not 60) and if hour is positive. hour and minute should be integral values, but we might allow second to have a fraction part.

Requirements like these are called **invariants** because they should always be true. To put it a different way, if they are not true, something has gone wrong.

Writing code to check invariants can help detect errors and find their causes. For example, you might have a function like valid_time that takes a Time object and returns False if it violates an invariant:

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second <
        0: return False
    if time.minute >= 60 or time.second
        >= 60: return False
    return True
```

At the beginning of each function you could check the arguments to make sure they are valid:

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid Time object in
    add_time') seconds = time_to_int(t1) +
    time_to_int(t2)
    return int_to_time(seconds)
```

Or you could use an **assert statement**, which checks a given invariant and raises an exception if it fails:

```
def add_time(t1, t2):
    assert valid_time(t1) and
    valid_time(t2) seconds =
    time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

assertstatements are useful because they distinguish code that deals with normal conditions from code that checks for errors.

Object-Oriented Features

Python is an **object-oriented programming language**, which means that it provides features that support object-oriented programming, which has these defining charac- teristics:

- Programs include class and method definitions.
- Most of the computation is expressed in terms of operations on objects.
- Objects often represent things in the real world, and methods often correspond to the ways things in the real world interact.

So far, we have not taken advantage of the features Python provides to support object-oriented programming. These features are not strictly necessary; most of them

provide alternative syntax for things we have already done. But in many cases, the alternative is more concise and more accurately conveys the structure of the program.

For example, in Time1.py there is no obvious connection between the class definition and the function definitions that follow. With some examination, it is apparent that every function takes at least one Timeobject as an argument.

This observation is the motivation for **methods**; a method is a function that is associ- ated with a particular class. We have seen methods for strings, lists, dictionaries and tuples. In this chapter, we will define methods for programmer-definedtypes.

Methods are semantically the same as functions, but there are two syntactic differ- ences:

- Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.
- The syntax for invoking a method is different from the syntax for calling a func- tion.

In the next few sections, we will take the functions from the previous two chapters and transform them into methods. This transformation is purely

mechanical; you can do it by following a sequence of steps. If you are comfortable converting from one form to another, you will be able to choose the best form for whatever you are doing.

Printing Objects

write a function named print_time:

```
class Time:
    """Represents the time of day."""

def print_time(time):
    print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

To call this function, you have to pass a Timeobject as an argument:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>>
print_time(start)
09:45:00
```

To make print_time a method, all we have to do is move the function definition inside the class definition. Notice the change in indentation.

```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

Now there are two ways to call print_time. The first (and less common) way is to use function syntax:

```
>>>
Time.print_time(start)
09:45:00
```

In this use of dot notation, Time is the name of the class, and print_time is the name of the method. startis passed as a parameter.

The second (and more concise) way is to use method syntax:

```
>>>
start.print_time()
09:45:00
```

In this use of dot notation, print_timeis the name of the method (again), and start

is the object the method is invoked on, which is called the **subject**. Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about.

Inside the method, the subject is assigned to the first parameter, so in this case start is assigned to time.

By convention, the first parameter of a method is called self, so it would be more common to write print_time like this:

```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

The reason for this convention is an implicit metaphor:

- The syntax for a function call, print_time(start), suggests that the function is the active agent. It says something like, "Hey print_time! Here's an object for you to print."
- In object-oriented programming, the objects are the active agents. A method invocation like start.print_time() says "Hey start! Please print yourself."

This change in perspective might be more polite, but it is not obvious that it is useful. In the examples we have seen so far, it may not be. But sometimes shifting responsi- bility from the functions onto the objects makes it possible to write more versatile functions (or methods), and makes it easier to maintain and reuse code.

As an exercise, rewrite time_to_int (from "Prototyping versus Planning" on page 190) as a method. You might be tempted to rewrite int_to_time as a method, too, but that doesn't really make sense because there would be no object to invoke it on.

Another Example

Here's a version of increment (from "Modifiers" on page 189) rewritten as a method:

```
# inside class Time:

    def increment(self, seconds):
        seconds +=
        self.time_to_int() return
        int_to_time(seconds)
```

This version assumes that time_to_intis written as a method. Also, note that it is a pure function, not a modifier.

Here's how you would invoke increment:

```
>>>
start.print_time()
09:45:00
>>> end = start.increment(1337)
>>>
end.print_time()
10:07:17
```

The subject, start, gets assigned to the first parameter, self. The argument, 1337, gets assigned to the second parameter, seconds.

This mechanism can be confusing, especially if you make an error. For example, if you invoke incrementwith two arguments, you get:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

The error message is initially confusing, because there are only two arguments in parentheses. But the subject is also considered an argument, so all together that's three.

By the way, a **positional argument** is an argument that doesn't have a parameter name; that is, it is not a keyword argument. In this function call:

```
sketch(parrot, cage, dead=True)
```

parrotand cageare positional, and deadis a keyword argument.

A More Complicated Example

Rewriting is_after (from "Time" on page 187) is slightly more complicated because it takes two Time objects as parameters. In this case it is conventional to name the first parameter selfand the second parameter other:

```
# inside class Time:
    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

To use this method, you have to invoke it on one object and pass the other as an argument:

```
>>>
end.is_after(start)
True
```

One nice thing about this syntax is that it almost reads like English: "end is after start?"

The initMethod

The init method (short for "initialization") is a special method that gets invoked when an object is instantiated. Its full name is__init__(two underscore characters, fol- lowed by init, and then two more underscores). An init method for the Time class might look like this:

```
# inside class Time:

    def init (self, hour=0, minute=0,
        second=0): self.hour = hour
        self.minute =
        minute
        self.second =
        second
```

It is common for the parameters of init to have the same names as the attributes. The statement

```
self.hour = hour
```

stores the value of the parameter hour as an attribute of self.

The parameters are optional, so if you call Time with no arguments, you get the default values:

```
>>> time = Time()
>>>
time.print_time()
00:00:00
```

If you provide one argument, it overrides hour:

```
>>> time = Time (9)
>>>
time.print_time()
09:00:00
```

If you provide two arguments, they override hour and minute:

```
>>> time = Time(9, 45)
>>>
time.print_time()
09:45:00
```

And if you provide three arguments, they override all three default values.

The_____str_____Method

_str_is a special method, like_init_, that is supposed to return a string representation of an object.

For example, here is a strmethod for Time objects:

```
# inside class Time:

    def str (self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

When you printan object, Python invokes the strmethod:

```
>>> time = Time(9, 45)
>>>
print(time)
09:45:00
```

When I write a new class, I almost always start by writing_init_, which makes it easier to instantiate objects, and_str_, which is useful for debugging.

As an exercise, write a strmethod for the Pointclass. Create a Point object and print it.

Operator Overloading

By defining other special methods, you can specify the behavior of operators on programmer-defined types. For example, if you define a method named add for the Timeclass, you can use the +operator on Time objects.

Here is what the definition might look like:

```
# inside class Time:

    def add (self, other):
        seconds = self.time_to_int() +
        other.time_to_int() return
        int_to_time(seconds)
```

And here is how you could use it:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start +
duration) 11:20:00
```

When you apply the + operator to Time objects, Python invokes _add_. When you print the result, Python invokes _str_. So there is a lot happening behind the scenes!

Changing the behavior of an operator so that it works with programmer-defined types is called **operator overloading**. For every operator in Python there is a corre-

As an exercise, write an addmethod for the Point class.

Type-Based Dispatch

In the previous section we added two Time objects, but you also might want to add an integer to a Time object. The following is a version of _add_ that checks the type of otherand invokes either add_timeor increment:

```
# inside class Time:

    def _add_(self, other):
        if isinstance(other, Time):
            return
            self.add_time(other)
        else:
            return self.increment(other)

    def add_time(self, other):
        seconds = self.time_to_int() +
        other.time_to_int() return
        int_to_time(seconds)

    def increment(self, seconds):
        seconds +=
        self.time_to_int() return
        int_to_time(seconds)
```

The built-in function isinstancetakes a value and a class object, and returns Trueif the value is an instance of the class.

If otheris a Time object,_add_invokes add_time. Otherwise it assumes that the parameter is a number and invokes increment. This operation is called a **type-based dispatch** because it dispatches the computation to different methods based on the type of the arguments.

Here are examples that use the +operator with different types:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start +
duration) 11:20:00
>>> print(start +
1337) 10:07:17
```

Unfortunately, this implementation of addition is not commutative. If the integer is the first operand, you get

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add a Time object, and it doesn't know how. But there is a clever solu- tion for this problem: the special method_radd_, which stands for "right-side add". This method is invoked when a Time object appears on the right side of the +opera- tor. Here's thedefinition:

```
# inside class Time:

    def radd_(self, other):
        return self._add_(other)
```

And here's how it's used:

```
>>> print(1337 +
start) 10:07:17
```

As an exercise, write an add method for Points that works with either a Point object or a tuple:

- If the second operand is a Point, the method should return a new Point whose *x* coordinate is the sum of the *x* coordinates of the operands, and likewise for the *y* coordinates.
- If the second operand is a tuple, the method should add the first element of the tuple to the *x* coordinate and the second element to the *y* coordinate, and return a new Point with the result.

Polymorphism

Type-based dispatch is useful when it is necessary, but (fortunately) it is not always necessary. Often you can avoid it by writing functions that work correctly for argu- ments with different types.

Many of the functions we wrote for strings also work for other sequence types. For example, in “Dictionary as a Collection of Counters” on page 127 we used histogram to count the number of times each letter appears in a word:

```
def
    histogram(
    s): d =
    dict() for c
    in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

This function also works for lists, tuples, and even dictionaries, as long as the ele- ments of sare hashable, so they can be used as keys in d:

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

Functions that work with several types are called **polymorphic**. Polymorphism can facilitate code reuse. For example, the built-in function sum, which adds the elements of a sequence, works as long as the elements of the sequence support addition.

Since Time objects provide an addmethod, they work with sum:

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>>
print(total)
23:01:00
```

In general, if all of the operations inside a function work with a given type, the func- tion works with that type.

The best kind of polymorphism is the unintentional kind, where you discover that a function you already wrote can be applied to a type you never planned for.

Interface and Implementation

One of the goals of object-oriented design is to make software more maintainable, which means that you can keep the program working when other parts of the system change, and modify the program to meet new requirements.

A design principle that helps achieve that goal is to keep interfaces separate from implementations. For objects, that means that the methods a class provides should not depend on how the attributes are represented.

For example, in this chapter we developed a class that represents a time of day. Meth- ods provided by this class include time_to_int, is_after, and add_time.

We could implement those methods in several ways. The details of the implementa- tion depend on how we represent time. In this chapter, the attributes of a Time object are hour, minute, and second.

As an alternative, we could replace these attributes with a single integer representing the number of seconds since midnight. This implementation would make some methods, like is_after, easier to write, but it makes other methods harder.

After you deploy a new class, you might discover a better implementation. If other parts of the program are using your class, it might be time-consuming and error- prone to change the interface.

But if you designed the interface carefully, you can change the implementation without changing the interface, which means that other parts of the program don't have to change.

Debugging

It is legal to add attributes to objects at any point in the execution of a program, but if you have objects with the same type that don't have the same attributes, it is easy to make mistakes. It is considered a good idea to initialize all of an object's attributes in the init method.

If you are not sure whether an object has a particular attribute, you can use the built-in function hasattr(see "Debugging" on page 183).

Another way to access attributes is the built-in function vars, which takes an object and returns a dictionary that maps from attribute names (as strings) to their values:

```
>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```

For purposes of debugging, you might find it useful to keep this function handy:

```
def
    print_attributes(obj
    ): for attr in
    vars(obj):
        print(attr, getattr(obj, attr))
```

print_attributes traverses the dictionary and prints each attribute name and its corresponding value.

The built-in function getattr takes an object and an attribute name (as a string) and returns the attribute's value.

Inheritance

Card Objects

There are 52 cards in a deck, each of which belongs to 1 of 4 suits and 1 of 13 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 $\mapsto$ 3 Hearts $\mapsto$ 2 Diamonds $\mapsto$ 1 Clubs $\mapsto$ 0

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

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

Jack $\mapsto$ 11

Queen $\mapsto$ 12

King $\mapsto$ 13

I am using the ↦ symbol to make it clear that these mappings are not part of the Python program. They are part of the program design, but they don't appear explic- itly in the code.

The class definition for Cardlooks like this:

```
class Card:
    """Represents a standard playing card."""

    def init (self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

As usual, the init method takes an optional parameter for each attribute. The default card is the 2 of Clubs.

To create a Card, you call Cardwith the suit and rank of the card you want:

```
queen_of_diamonds = Card(1, 12)
```

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**:

```
# inside class Card:

    suit_names = ['Clubs', 'Diamonds', 'Hearts',
    'Spades'] rank_names = [None, 'Ace', '2', '3', '4',
    '5', '6', '7',
                '8', '9', '10', 'Jack', 'Queen', 'King']

    def str (self):
return "%s of %s' % (Card.rank_names[self.rank],
                        Card.suit_names[self.suit])
```

Variables like suit_namesand rank_names, which are defined inside a class but out- side of any method, are called class attributes because they are associated with the class object Card.

This term distinguishes them from variables like suitand 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_, selfis a Card object, and self.rankis its rank. Similarly, Cardis a class object, and Card.rank_namesis 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.

Putting it all together, the expression Card.rank_names[self.rank] means "use the attribute rank from the object self as an index into the list rank_names from the class Card, and select the appropriate string."

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.

With the methods we have so far, we can create and print cards:

```
>>> card1 = Card(2, 11)
>>>
print(card1)
Jack of Hearts
```

Figure 18-1 is a diagram of the Card class object and one Card instance. Card is a class object; its type is type. card1 is an instance of Card, so its type is Card. To save space, I didn't draw the contents of suit_names and rank_names.

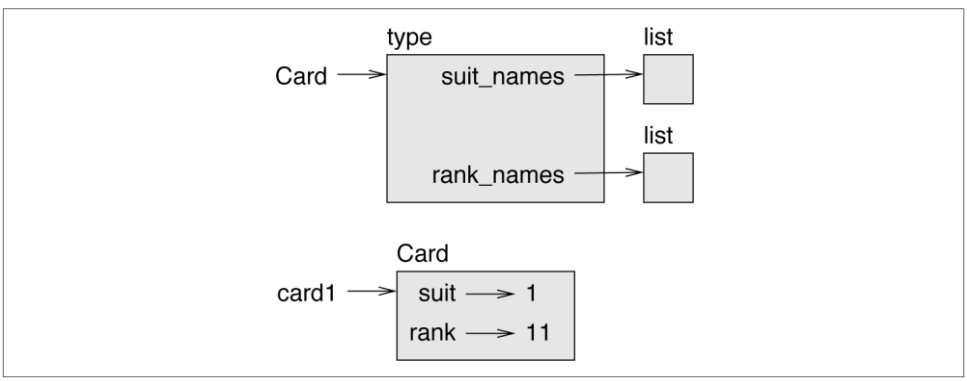

Figure 18-1. Object diagram.

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 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 impor- tant.

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 out- rank all of the Diamonds, and so on.

With that decided, we can write_lt_:

```
# inside class Card:

    def_lt_(self, other):
        # check the suits
        if self.suit < other.suit: return True
        if self.suit > other.suit: return
        False

        # suits are the same... check
        ranks return self.rank <
        other.rank
```

You can write this more concisely using tuple comparison:

```
# inside class Card:

    def_lt_(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit,
        other.rank return t1 <
        t2
```

As an exercise, write an__lt_method for Time objects. You can use tuple compari- son, but you also might consider comparing integers.

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
cardsand generates the standard set of 52 cards:

```
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(ca
            rd)
```

The easiest way to populate the deck is with a nested loop. The outer loop enumerates the suits from 0 to 3. The inner loop enumerates the ranks from 1 to 13. Each itera- tion creates a new Card with the current suit and rank, and appends it to self.cards.

Printing the Deck

Here is a str method for Deck:

```
#inside class Deck:

    def str (self):
        res = []
        for card in self.cards:
            res.append(str(car
            d))
        return '\n'.join(res)
```

This method demonstrates an efficient way to accumulate a large string: building a list of strings and then using the string method join. The built-in function str invokes the str method on each card and returns the string representation.

Since we invoke join on a newline character, the cards are separated by newlines. Here's what the result looks like:

```
\>>> deck = Deck()
    >>>
    print(deck)
    Ace of Clubs
    2 of Clubs
    3 of Clubs
    ...
    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 new- lines.

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 popprovides a convenient way to do that:

```
#inside class Deck:

    def pop_card(self):
        return self.cards.pop()
```

Since popremoves the *last* card in the list, we are dealing from the bottom of the deck.

To add a card, we can use the list method append:

```
#inside class Deck:

    def add_card(self,
        card):
        self.cards.append(ca
        rd)
```

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 implementa- tion.

As another example, we can write a Deck method named shuffleusing the function shufflefrom the randommodule:

```
# inside class Deck:

    def shuffle(self):
        random.shuffle(self.ca
        rds)
```

Don’t forget to import random.

As an exercise, write a Deck method named sortthat uses the list method sortto sort the cards in a Deck. sortuses the_lt_method we defined to determine the order.

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. Todefine a new class that inherits from an existing class, you put the name of the existing class in parentheses:

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

This definition indicates that Handinherits from Deck; that means we can use meth- ods like pop_cardand add_cardfor 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, Handinherits_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 cardswith an empty list.

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

```
# inside class Hand:

    def init (self, label=''):
        self.cards = []
        self.label = label
```

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

```
>>> hand = Hand('new hand')
>>>
hand.cards
[]
>>>
hand.label
'new hand'
```

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

```
>>> 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:

```
#inside class Deck:

    def move_cards(self, hand,
        num): for i in
        range(num):
            hand.add_card(self.pop_card())
```

move_cardstakes two arguments, a Hand object and the number of cards to deal. It modifies both selfand 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 mod- ify 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.

Class Diagrams

So far we have seen stack diagrams, which show the state of a program, and object diagrams, which show the attributes of an object and their values. These diagrams represent a snapshot in the execution of a program, so they change as the program runs.

They are also highly detailed; for some purposes, too detailed. A class diagram is a more abstract representation of the structure of a program. Instead of showing indi- vidual 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, Deckand Hand.

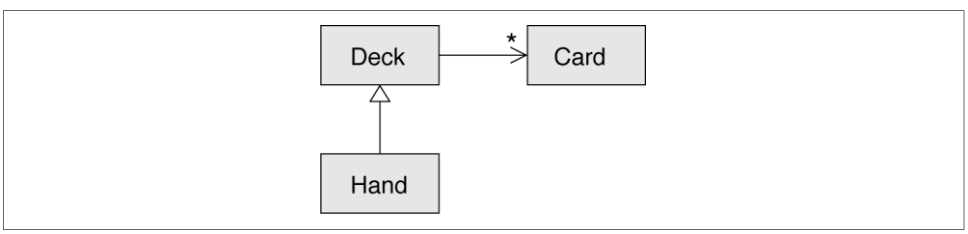

Figure 18-2. Class diagram.

The arrow with a hollow triangle head represents an IS-A relationship; in this case it indicates that Hand inherits from Deck.

The standard arrowhead represents a HAS-A relationship; in this case a Deck has ref- erences to Card objects.

The star (*) near the arrowhead 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.

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 correspond-

ence 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 discov- ered function interfaces by encapsulation and generalization, we can discover class interfaces by **data encapsulation**.

Markov analysis, from "Markov Analysis" on page 158, provides a good example. If you download my code from *http://thinkpython2.com/code/markov.py*, 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:

```
class Markov:

    def init (self):
        self.suffix_map =
        {} self.prefix = ()
```

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

```
    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)
```

Transforming a program like this—changing the design without changing the behav- ior—is another example of refactoring (see "Refactoring" on page 41).

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 vari- ables 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 *http://thinkpython2.com/code/ markov.py*, and follow the steps described above to encapsulate the global variables as attributes of a new class called Markov.

Solution: *http://thinkpython2.com/code/Markov.py* (note the capital M).

Debugging

Inheritance can make debugging difficult because when you invoke a method on an object, it might be hard to figure out which method will be invoked.

Suppose you are writing a function that works with Hand objects. You would like it to work with all kinds of Hands, like PokerHands, BridgeHands, etc. If you invoke a method like shuffle, you might get the one defined in Deck, but if any of the sub- classes override this method, you'll get that version instead. This behavior is usually a good thing, but it can be confusing.

Any time you are unsure about the flow of execution through your program, the sim- plest solution is to add print statements at the beginning of the relevant methods. If Deck.shuffle prints a message that says something like Running

Deck.shuffle, then as the program runs it traces the flow of execution.

As an alternative, you could use this function, which takes an object and a method name (as a string) and returns the class that provides the definition of the method:

```
def find_defining_class(obj,
    meth_name): for ty in
    type(obj).mro():
        if meth_name in ty._dict_:
            return ty
```

Here's an example:

```
>>> hand = Hand()
>>> find_defining_class(hand, 'shuffle')
<class 'Card.Deck'>
```

So the shufflemethod for this Hand is the one in Deck.

find_defining_classuses the mromethod to get the list of class objects (types) that will be searched for methods. "MRO" stands for "method resolution order", which is the sequence of classes Python searches to "resolve" a method name.

Here's a design suggestion: when you override a method, the interface of the new method should be the same as the old. It should take the same parameters, returnthe same type, and obey the same preconditions and postconditions. If you follow this rule, you will find that any function designed to work with an instance of a parent class, like a Deck, will also work with instances of child classes like a Hand and Poker- Hand.

If you violate this rule, which is called the "Liskov substitution principle", your code will collapse like (sorry) a house of cards.

The Goodies

One of my goals for this book has been to teach you as little Python as possible. When there were two ways to do something, I picked one and avoided mentioning the other. Or sometimes I put the second one into an exercise.

Now I want to go back for some of the good bits that got left behind. Python provides a number of features that are not really necessary—you can write good code without them—but with them you can sometimes write code that's more concise, readable or efficient, and sometimes all three.

Conditional Expressions

We saw conditional statements in "Conditional Execution" on page 49. Conditional statements are often used to choose one of two values; for example:

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

This statement checks whether x is positive. If so, it computes math.log. If not, math.log would raise a ValueError. To avoid stopping the program, we generate a "NaN", which is a special floating-point value that represents "Not a Number".

We can write this statement more concisely using a **conditional expression**:

```
y = math.log(x) if x > 0 else float('nan')
```

You can almost read this line like English: "y gets log-x if x is greater than 0; other- wise it gets NaN".

Recursive functions can sometimes be rewritten using conditional expressions. For example, here is a recursive version of factorial:

```
def
    factorial(n)
    : if n == 0:
        return
    1 else:
        return n * factorial(n-1)
```

We can rewrite it like this:

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)
```

Another use of conditional expressions is handling optional arguments. For example, here is the init method from GoodKangaroo(see Exercise 17-2):

```
    def init (self, name, contents=None):
        self.name = name
        if contents == None:
            contents = []
        self.pouch_contents = contents
```

We can rewrite this one like this:

```
def __init__(self, name, contents=None):
    self.name = name
    self.pouch_contents = [] if contents == None else contents
```

In general, you can replace a conditional statement with a conditional expression if both branches contain simple expressions that are either returned or assigned to the same variable.

List Comprehensions

In "Map, Filter and Reduce" on page 111 we saw the map and filter patterns. For example, this function takes a list of strings, maps the string method capitalizeto the elements, and returns a new list of strings:

```
def
    capitalize_all(t)
    : res = []
    for s in t:
        res.append(s.capitalize
    ()) return res
```

We can write this more concisely using a **list comprehension**:

```
def capitalize_all(t):
    return [s.capitalize() for s in t]
```

The bracket operators indicate that we are constructing a new list. The expression inside the brackets specifies the elements of the list, and the forclause indicates what sequence we aretraversing.

The syntax of a list comprehension is a little awkward because the loop variable, s in this example, appears in the expression before we get to the definition.

List comprehensions can also be used for filtering. For example, this function selects only the elements of tthat are uppercase, and returns a new list:

```
def
    only_upper(
    t): res = []
    for s in t:
        if s.isupper():
            res.append(
            s)
    return res
```

We can rewrite it using a list comprehension:

```
def only_upper(t):
    return [s for s in t if s.isupper()]
```

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

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

```
>>> g = (x**2 for x in range(5))
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

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:

```
>>>
next(g) 0
>>>
next(g) 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:

```
>>> 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 StopException:

```
>>>
next(g)
StopIterati
on
```

Generator expressions are often used with functions like sum, max, and min:

```
>>> sum(x**2 for x in
range(5)) 30
```

any andall

Python provides a built-in function, any, that takes a sequence of boolean values and returns Trueif any of the values are True. It works on lists:

```
>>> any([False, False,
True]) True
```

But it is often used with generator expressions:

```
>>> any(letter == 't' for letter in
'monty') True
```

That example isn't very useful because it does the same thing as the inoperator. But we could use anyto rewrite some of the search functions we wrote in "Search" on page 101. For example, we could write avoidslike this:

```
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 anywith a generator expression is efficient because it stops immediately if it finds a Truevalue, so it doesn't have to evaluate the whole sequence.

Python provides another built-in function, all, that returns Trueif every element of the sequence is True. As an exercise, use allto rewrite uses_allfrom "Search" on page 101.

Sets

In "Dictionary Subtraction" on page 156 I use dictionaries to find the words that appear in a document but not in a word list. The function I wrote takes d1, which contains the words from the document as keys, and d2, which contains the list of words. It returns a dictionary that contains the keys from d1 that are not in d2:

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] =
            None
    return res
```

In all of these dictionaries, the values are None because we never use them. As a result, we waste some storage space.

Python provides another built-in type, called a set, that behaves like a collection of dictionary keys with no values. Adding elements to a set is fast; so is checking mem- bership. And sets provide methods and operators to compute common set opera- tions.

For example, set subtraction is available as a method called differenceor as an operator, -. So we can rewrite subtractlike this:

```
def subtract(d1, d2):
    return set(d1) - set(d2)
```

The result is a set instead of a dictionary, but for operations like iteration, the behavior is the same.

Some of the exercises in this book can be done concisely and efficiently with sets. For example, here is a solution to has_duplicates, from Exercise 10-7, that uses a dictio- nary:

```
def
    has_duplicates(
    t): d = {}
    for x in t:
        if x in d:
            return True
        d[x] = True
    return False
```

When an element appears for the first time, it is added to the dictionary. If the same element appears again, the function returns True.

Using sets, we can write the same function like this:

```
def has_duplicates(t):
    return len(set(t)) < len(t)
```

An element can only appear in a set once, so if an element in tappears more than once, the set will be smaller than t. If there are no duplicates, the set will be the same size as t.

We can also use sets to do some of the exercises in Chapter 9. For example, here's a version of uses_onlywith a loop:

```
def uses_only(word,
    available): for letter in
    word:
        if letter not in available:
```

```
            return False
    return True
```

uses_onlychecks whether all letters in wordare in available. We can rewrite it like this:

```
def uses_only(word, available):
    return set(word) <= set(available)
```

The <= operator checks whether one set is a subset or another, including the possibil- ity that they are equal, which is true if all the letters in word appear in available.

As an exercise, rewrite avoidsusing sets.

Counters

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 itera- tion:

```
>>> from collections import Counter
>>> count = Counter('parrot')
>>> count
Counter({'r': 2, 't': 1, 'o': 1, 'p': 1, 'a': 1})
```

Counters behave like dictionaries in many ways; they map from each key to the num- ber 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:

```
>>>
count['d'] 0
```

We can use Counters to rewrite is_anagramfrom Exercise 10-6:

```
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:

```
>>> count = Counter('parrot')
>>> for val, freq in count.most_common(3):
...         print(val,
freq) r 2
p 1
a 1
```

defaultdict

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:

```
>>> from collections import defaultdict
>>> d = 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:

```
>>> t = d['new key']
>>>
t []
```

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

```
>>> t.append('new value')
>>> d
defaultdict(<class 'list'>, {'new key': ['new value']})
```

If you are making a dictionary of lists, you can often write simpler code using defaultdict. In my solution to Exercise 12-2, which you can get from *http://thinkpy thon2.com/code/anagram_sets.py*, I make a dictionary that maps from a sorted string of letters to the list of words that can be spelled with those letters. For example, 'opst' maps to the list ['opts', 'post', 'pots', 'spot', 'stop', 'tops'].

Here's the original code:

```
def
    all_anagrams(filena
```

```
me): d = {}
for line in open(filename):
    word =
    line.strip().lower() t =
    signature(word)
    if t not in d:
        d[t] =
    [word] else:
        d[t].append(word)
return d
```

This can be simplified using setdefault, which you might have used in Exercise 11-2:

```
def
    all_anagrams(filena
    me): 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 compli- cated, it might be.

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

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

My solution to Exercise 18-3, which you can download from *http://thinkpython2.com/ code/PokerHandSoln.py*, uses setdefaultin the function has_straightflush. This solution has the drawback of creating a Hand object every time through the loop, whether it is needed or not. As an exercise, rewrite it using a defaultdict.

Named Tuples

Many simple objects are basically collections of related values. For example, the Point object defined in Chapter 15 contains two numbers, x and y. When you define a class like this, you usually start with an init method and a str method:

```
class Point:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

This is a lot of code to convey a small amount of information. Python provides a more concise way to say the same thing:

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

The first argument is the name of the class you want to create. The second is a list of the attributes Point objects should have, as strings. The return value from namedtuple is a class object:

```
>>> Point
<class '__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:

```
>>> p = Point(1, 2)
>>> p
Point(x=1, y=2)
```

The init method assigns the arguments to attributes using the names you provided. The str method prints a representation of the Point object and its attributes.

You can access the elements of the named tuple by name:

```
>>> p.x,
p.y (1, 2)
```

But you can also treat a named tuple as a tuple:

```
>>> p[0],
p[1] (1, 2)
```

```
>>> x, y = p
>>> x,
y (1, 2)
```

Named tuples provide a quick way to define simple classes. The drawback is that sim- ple 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:

```
class Pointier(Point):
    # add more methods here
```

Or you could switch to a conventional class definition.

Gathering KeywordArgs

In "Variable-Length Argument Tuples" on page 142, we saw how to write a function that gathers its arguments into a tuple:

```
def
    printall(*args)
    : print(args)
```

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

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

But the *operator doesn't gather keyword arguments:

```
>>> printall(1, 2.0, third='3')
TypeError: printall() got an unexpected keyword argument 'third'
```

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

```
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 keywords to values:

```
>>> 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:

```
>>> d = dict(x=1, y=2)
```

```
>>>
Point(**d)
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:

```
>>> d = dict(x=1, y=2)
>>> Point(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
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.