CLASSES AND OBJECTS

1. Programmer-defined types

- We have used many of Python's built-in types; now we are going to define a new type. As an example, we will create a type called Point that represents a point in two-dimensional space.
- In mathematical notation, points are often written in parentheses with a comma separating the coordinates.
- For example, (0, 0) represents the origin, and (x, y) represents the point x units to theright and y units up from the origin.
- There are several ways we might represent points in Python:
  1. We could store the coordinates separately in two variables, x and y.
  2. We could store the coordinates as elements in a list or tuple.
  3. We could create a new type to represent points as objects.
- Creating a new type is more complicated than the other options, but it has advantages that will be apparent soon.
- A programmer-defined type is also called a class. A class definition looks like this:

```
class Point:
"""Represents a point in 2-D space."""
```

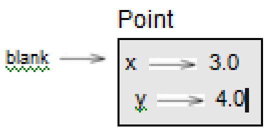


Figure 15.1: Object diagram.

- The header indicates that the new class is called Point. The body is a docstring that ex-plains what the class is for. You can define variables and methods inside a class definition, but we will get back to that later.
- Defining a class named Point creates a class object.

```
>>> Point
        <class '__main__.Point'>
```

- Because Point is defined at the top level, its "full name" is __main__.Point.
- The class object is like a factory for creating objects. To create a Point, you call Point as if it were a function.

```
blank = Point()
blank
<__main__.Point object at 0xb7e9d3ac>
```

- The return value is a reference to a Point object, which we assign to blank.
- Creating a new object is called instantiation, and the object is an instance of the class.
- When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix 0x means that the following number is in hexadecimal).

2. Attributes

- You can assign values to an instance using dot notation:

```
blank.x = 3.0
blank.y = 4.0
```

- This syntax is similar to the syntax for selecting a variable from a module, such as math.pi or string.whitespace .
- In this case, though, we are assigning values to named elements of an object. These elements are called attributes.
- A state diagram that shows an object and its attributes is called an object diagram; see Figure 15.1.
- The variable blank refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number.
- You can read the value of an attribute using the same syntax:

blank.y 4.0	x = blank.x x 3.0

- The expression blank.x means, "Go to the object blank refers to and get the value of x." In the example, we assign that value to a variable named x. There is no conflict between the variable x and the attribute x.
- You can use dot notation as part of any expression. For example

```
'(%g, %g)' % (blank.x, blank.y)
 '(3.0, 4.0)'
```

```
distance = math.sqrt(blank.x**2 + blank.y**2)
distance
5.0
```

You can pass an instance as an argument in the usual way. For example:

```
def print_point(p):
print('(%g, %g)' % (p.x, p.y))
```

- print_point takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass blank as an argument:

```
print_point(blank) (3.0, 4.0)
```

- Inside the function, p is an alias for blank, so if the function modifies p, blank changes.

3. Rectangles

- Sometimes it is obvious what the attributes of an object should be, but other times you have to make decisions.

- For example, imagine you are designing a class to represent rectangles. What attributes would you use to specify the location and size of a rectangle?
- You can ignore angle; to keep things simple, assume that the rectangle is either vertical or horizontal.
- There are at least two possibilities:
    1. You could specify one corner of the rectangle (or the center), the width, and the height.
    2. You could specify two opposing corners.
- At this point it is hard to say whether either is better than the other, so we'll implement the first one, just as an example.
- Here is the class definition:

```
class Rectangle:
"""Represents a rectangle.
attributes: width, height, corner."""
```

- The docstring lists the attributes: width and height are numbers; corner is a Point object that specifies the lower-left corner.
- To represent a rectangle, you have to instantiate a Rectangle object and assign values to the attributes:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

- The expression box.corner.x means, "Go to the object box refers to and select the attribute named corner; then go to that object and select the attribute named x."
- Figure 15.2 shows the state of this object. An object that is an attribute of another object is **embedded**.

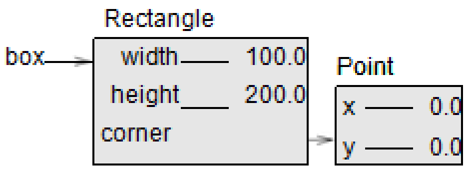


Figure 15.2: Object diagram.

4. Instances as return values

- Functions can return instances. For example, find_center takes a Rectangle as an argument and returns a Point that contains the coordinates of the center of the Rectangle:

```
def find_center(rect):
p = Point()
p.x = rect.corner.x + rect.width/2
p.y = rect.corner.y + rect.height/2
return p
```

- Here is an example that passes box as an argument and assigns the resulting Point to center:

```
center = find_center(box)
print_point(center)
(50, 100)
```

5. Objects are mutable

- You can change the state of an object by making an assignment to one of its attributes. For example, to change the size of a rectangle without changing its position, you can modify the values of width and height:

```
box.width = box.width + 50
box.height = box.height + 100
```

- You can also write functions that modify objects.
- For example, grow_rectangle takes a Rectangle object and two numbers, dwidth and dheight, and adds the numbers to the width and height of the rectangle:

```
def grow_rectangle(rect, dwidth, dheight):
rect.width += dwidth
rect.height += dheight
```

- Here is an example that demonstrates the effect:

```
box.width, box.height (150.0, 300.0)
grow_rectangle(box, 50, 100)
box.width, box.height
(200.0, 400.0)
```

- Inside the function, rect is an alias for box, so when the function modifies rect, box changes.

6. Copying

- Aliasing can make a program difficult to read because changes in one place might have unexpected effects in another place.
- It is hard to keep track of all the variables that might refer to a given object.
- Copying an object is often an alternative to aliasing. The copy module contains a function called copy that can duplicate any object:

```
p1 = Point()
p1.x = 3.0
p1.y = 4.0

import copy
p2 = copy.copy(p1)
```

- p1 and p2 contain the same data, but they are not the same Point.

print_point(p1) (3, 4)
print_point(p2) (3, 4)
p1 is p2 False
p1 == p2 False

- The is operator indicates that p1 and p2 are not the same object, which is what we expected. But you might have expected == to yield True because these points contain the same data.
- In that case, you will be disappointed to learn that for instances, the default behavior of the == operator is the same as the is operator; it checks object identity, not object equivalence.
- That's because for programmer-defined types, Python doesn't know what should be considered equivalent. At least, not yet.

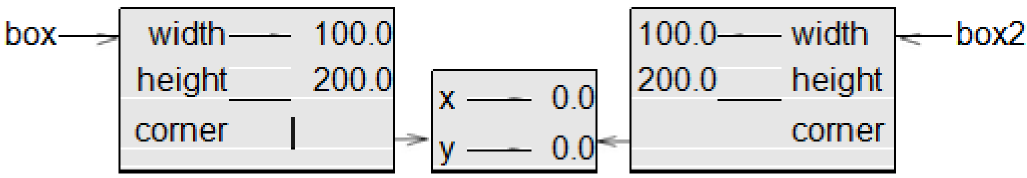


Figure 15.3: Object diagram.

- If you use copy.copy to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point.

box2 = copy.copy(box) box2 is box False
box2.corner is box.corner True

- Figure 15.3 shows what the object diagram looks like. This operation is called a **shallow copy** because it copies the object and any references it contains, but not the embedded objects.
- For most applications, this is not what you want.
- In this example, invoking grow_rectangle on one of the Rectangles would not affect the other, but invoking move_rectangle on either would affect both! This behavior is confusing and error-prone.
- Fortunately, the copy module provides a method named deepcopy that copies not only the object but also the objects it refers to, and the objects they refer to, and so on. You will not be surprised to learn that this operation is called a **deep copy.**

box3 = copy.deepcopy(box)
box3 is box False
box3.corner is box.corner False

- box3 and box are completely separate objects.

CLASSES AND FUNCTIONS

1. Time

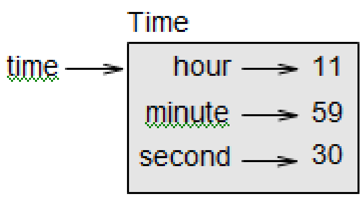
- As another example of a programmer-defined type, we'll define a class called Time that records the time of day. The class definition looks like this:

```
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 Time object looks like Figure below.



2. Pure functions

- In the next few sections, we'll write two functions that add time values.
- They demonstrate 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 complications.
- 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, let us create two Time objects: start contains the start time of a movie, like Monty Python and the Holy Grail, and duration contains the run time of the movie, which is one 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
```

3. 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 sixty?
- In that case, it is not enough to carry once; we have to keep doing it until time.second is less than sixty.
- One solution is to replace the if statements with while statements. That would make the function correct, but not very efficient.
- Anything that can be done with modifiers can also be done with pure functions.

4. Prototyping versus planning

- The development plan, i.e. demonstrating is called “prototype and patch”. For each function, we 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.
- 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(seconds):
 time = Time()
minutes, time.second = divmod(seconds, 60)
time.hour, time.minute = divmod(minutes, 60)
return time
```

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

```
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.

CLASSES AND METHODS

1. 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 characteristics:

- 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.

- A method is a function that is associated with a particular class.
- Methods are semantically the same as functions, but there are two syntactic differences:

- 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 function.

2. Printing Objects

- We already defined a class named and also wrote 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, we have to pass a Time object 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. start is 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_time is 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.”

3. Another Example

- Here’s a version of increment 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_int is 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 increment with 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)
```

- parrot and cage are positional, and dead is a keyword argument.

4. A More Complicated Example

- Rewriting is_after is slightly more complicated because it takes two Time objects as parameters.
- In this case it is conventional to name the first parameter self and 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
```

5. The init Method

- 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, followed 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 we provide one argument, it overrides hour:

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

- If we provide two arguments, they override hour and minute.

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

- And if we provide three arguments, they override all three default values

6. The __str__ Method

- __str__ is a special method, like init , that is supposed to return a string representa- tion of an object.

- For example, here is a str method for Time objects:

```
# inside class Time:
def __str__(self):
    return '%.2d:%.2d:%.2d' % (self.hour, self.minute,  self.second)
```

- When you print an object, Python invokes the str method:

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

7. Operator Overloading

- By defining other special methods, you can specify the behavior of operators on programmer-defined types.
- For example, if we define a method named__add__for the Time class, you can use the + operator on Time objects.
- Here is what the definition might look like:

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

- And here is how we 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 corresponding special method, like add .

8. Type-Based Dispatch

- The following is the version of _add_ that checks the type of other and invokes either add_time or increment:

```
def_add_(self,other):
    if isintance(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 isinstance takes a value and a class object, and returns True if the value is an instance of the class.
- If other is 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 solution 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 + operator. Here's the definition:

```
# inside class Time:
def __radd__(self, other):
        return self._add_(other)
```

- And here's how it's used:

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

9. 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 arguments with different types.

- Many of the functions we wrote for strings also work for other sequence types. For example, 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 elements of s are 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.

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 31)
>>> 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 function works with that type.
- The best kind of polymorphism is the unintentional kind, where you discover that a func- tion you already wrote can be applied to a type you never planned for.

INHERITANCE

1. Card Objects
2. Class attributes
3. Comparing Cards
4. Decks
5. Printing the Deck
6. Add, Remove, Shuffle and Sort
7. Inheritance
8. Class Diagrams
9. Data Encapsulation

1. 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.
- The ranks are Ace,2,3,4,5,6,7,8,9,10,Jack,Queen and King.
- If we want to define a new object to represent a playing card, it is obvious what the attributes should be: rank and suit.
- One possibility is to use strings containing words like 'Spade' for suits and 'Queen' for ranks.
- 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.
- Ex:

  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 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 $\mapsto$ 11
  Queen $\mapsto$ 12
  King $\mapsto$ 13
- 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 explicitly in the code.

- The class definition for Card looks like this:

```
class Card(object):
    """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, we call Card with the suit and rank of the card you want

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

2. Class Attributes

- In order to print Card, 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:

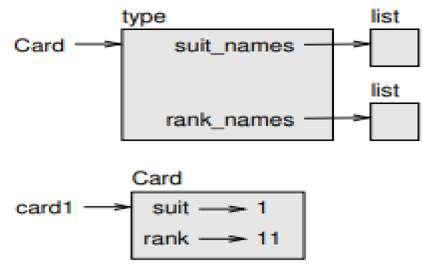
```
# inside class Card:

    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_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.
- The 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.
    - Ex, 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.
- 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
```



Object diagram

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 user-defined types, we can override the behavior of the built-in operators by providing a method named__ cmp___
- __cmp___takes two parameters, self and other, and returns a positive number if the first object is greater, a negative number if the second object is greater, and 0 if they are equal to each other.

```
# inside class Card:

    def __cmp__(self, other):
        # check the suits
        if self.suit > other.suit: return 1
        if self.suit < other.suit: return -1

        # suits are the same... check ranks
        if self.rank > other.rank: return 1
        if self.rank < other.rank: return -1

        # ranks are the same... it's a tie
        return 0
```

Class

```
# inside class Card:

    def __cmp__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return cmp(t1, t2)
```

cmp using tuples

- The built-in function cmp has the same interface as the method_ cmp : it takes two values and returns a positive number if the first is larger, a negative number if the second is larger, and 0 if they are equal.
- In Python 3, cmp no longer exists, and the_ cmp_ method is not supported. Instead you should provide__ lt__ , which returns True if self is less than other. You can implement __ lt using tuples and the < operator.

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 init method creates the attribute cards and generates the standard set of fifty-two cards:

```
class Deck(object):

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

- The outer loop enumerates the suits from 0 to 3.
- The inner loop enumerates the ranks from 1 to 13.
- Each iteration creates a new Card with the current suit and rank, and appends it to self.cards.

5. Printing the Deck

```
#inside class Deck:

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

Program

```
>>> 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
```

Output

- This method demonstrates an efficient way to accumulate a large string: building a list of strings and then using 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.
- Even though the result appears on 52 lines, it is one long string that contains newlines.

6. Add, Remove, Shuffle and Sort

<u>Remove</u>

- 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.

```
#inside class Deck:

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

<u>Add</u>

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

```
#inside class Deck:

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

- A method like this that uses another function without doing much real work is sometimes calleda **veneer-**(where it is common to glue a thin layer of good quality wood to the surface of a cheaper piece of wood).
- In this case we are defining a "thin" method that expresses a list operation in terms that are appropriate for decks.

Shuffle

- we can write a Deck method named shuffle using the function shuffle from the random module:
- Don't forget to import random.

```
# inside class Deck:

    def shuffle(self):
        random.shuffle(self.cards)
```

7. Inheritance

- ➢ 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.
- ➢ It is called "inheritance" because the new class inherits the methods of the existing class.
- ➢ Extending this metaphor, the existing class is called the parent and the new class is called the child.
- ➢ The definition of a child class is like other class definitions, but the name of the parent class appears in parentheses:

```
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.
- ➢ Inheritance is a useful feature
- ➢ 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 program easier to understand.

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.

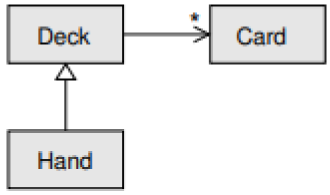


Figure: Class diagram

- 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 changes in one class would require changes in the other.
- A class diagram is a graphical representation of these relationships. For example, the above class diagram shows the relationships between Card, Deck and Hand.
- 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.