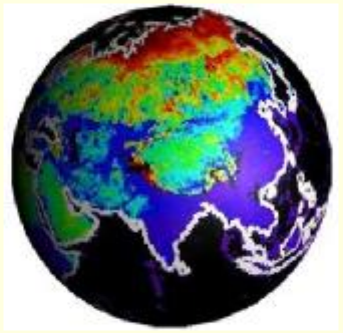


Chapter 18 Inheritance

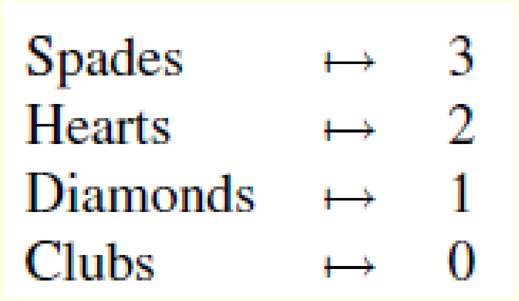
Jihn-Fa Jan, Ph.D.
Associate Professor
Department of Land Economics
National Chengchi University

Card objects

- There are fifty-two cards in a deck, each of which belongs to one of four suits and one of thirteen ranks. The suits are Spades, Hearts, Diamonds, and Clubs (in descending order in bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. Depending on the game that you are playing, an Ace may be higher than King or lower than 2.
- If we want to define a new object to represent a playing card, it is obvious what the attributes should be: rank and suit. It is not as obvious what type the attributes should be. One possibility is to use strings containing words like 'Spade' for suits and 'Queen' for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had a higher rank or suit.
- An alternative is to use integers to encode the ranks and suits. In this context, “encode” means that we are going to define a mapping between numbers and suits, or between numbers and ranks. This kind of encoding is not meant to be a secret (that would be “encryption”).

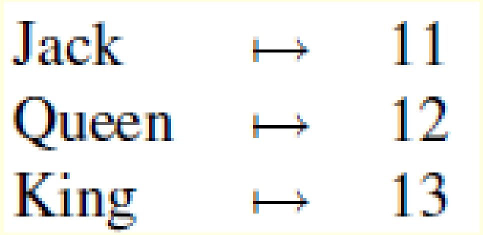
Card objects (cont'd)

- 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 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

Card objects (cont'd)

- The class definition for Card looks like this:

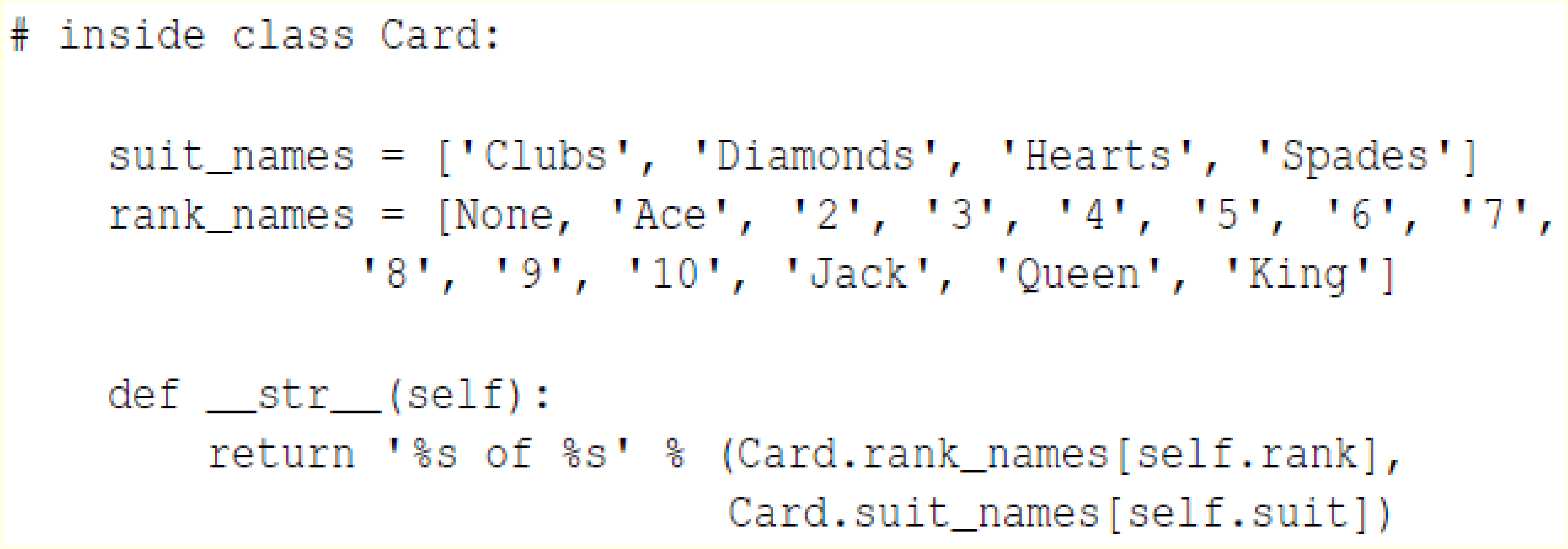
```
class Card(object):
    """represents a standard playing card."""

    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, you call Card with 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_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.

Class attributes (cont'd)

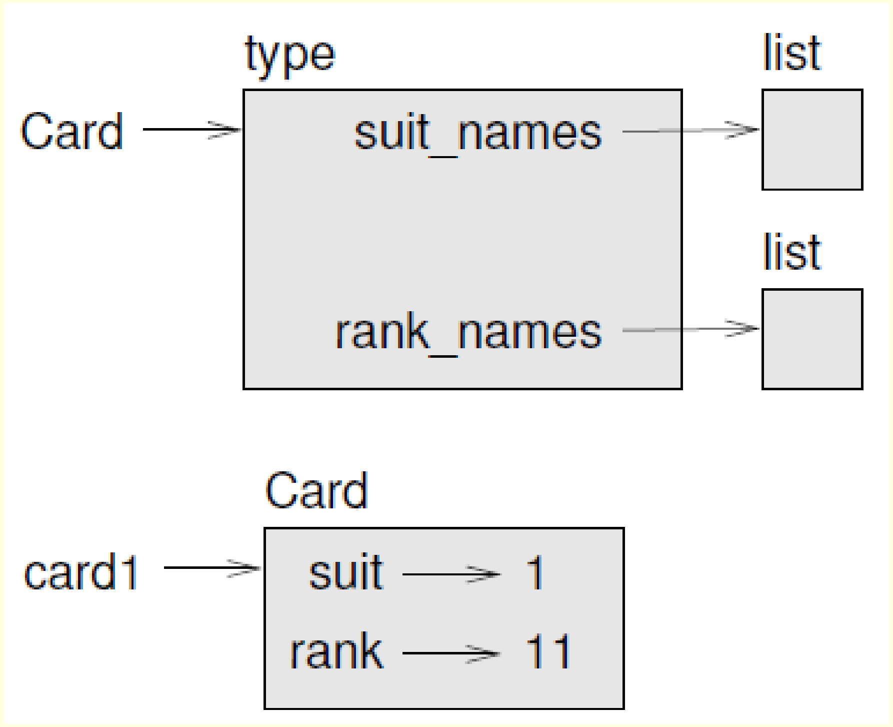
- This term distinguished them from variables like suit and rank, which are called instance attributes because they are associated with a particular instance.
- Both kinds of attribute are accessed using dot notation. For example, in __str__, self is a Card object, and self.rank is its rank. Similarly, Card is a class object, and Card.rank_names is a list of strings associated with the class.
- Every card has its own suit and rank, but there is only one copy of suit_names and rank_names.
- 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.

Class attributes (cont'd)

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

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

- Here is a diagram that shows the Card class object and one Card instance:



Comparing cards

- For built-in types, there are conditional 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.
- The correct ordering for cards is not obvious. For example, which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit. In order to compare cards, you have to decide whether rank or suit is more important.
- The answer might depend on what game you are playing, but to keep things simple, we’ll make the arbitrary choice that suit is more important, so all of the Spades outrank all of the Diamonds, and so on.

Comparing cards (cont'd)

- With that decided, we can write __cmp__:

```
# 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
```

- You can write this more concisely using tuple comparison:

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

Comparing cards (cont'd)

- 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 of the second is larger, and 0 if they are equal.

Decks

- **Now that we have Cards, the next step is to define Decks. Since a deck is made up of cards, it is natural for each Deck to contain a list of cards as an attribute.**
- **The following is a class definition for Deck. The init method creates the attribute cards and generates the standard set of fifty-two cards:**

```
class Deck(object):
    """represents a deck of cards"""

    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 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 iteration 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:

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

- 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. Here's what the result looks like:

```
>>> deck = Deck()
>>> print deck
Ace of Clubs
2 of Clubs
3 of Clubs
...
10 of Spades
```

Add, remove, shuffle and sort

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

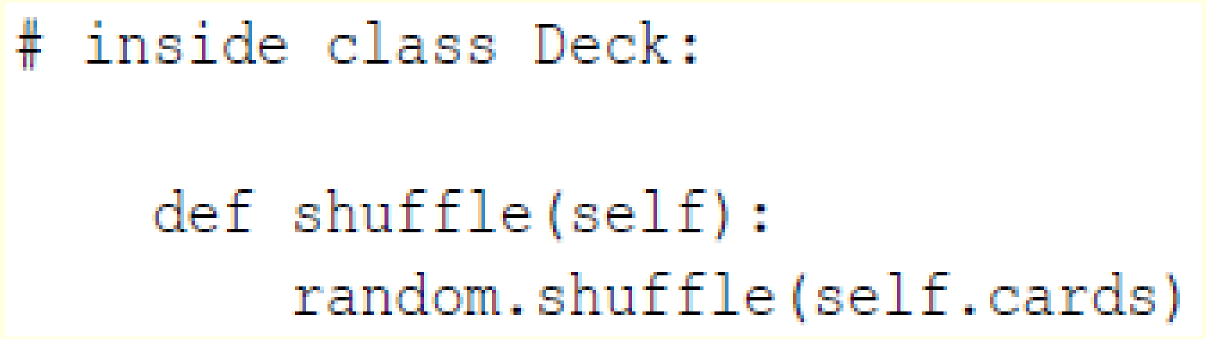
```
#inside class Deck:

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

- Since pop removes the *last card in the list, we are dealing from the bottom of the deck. In real life* bottom dealing is frowned upon, but in this context it’s ok.
- To add a card, we can use the list method append:

```
#inside class Deck:

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

Add, remove, shuffle and sort (cont'd)

- A method like this that uses another function without doing much real work is sometimes called a veneer. The metaphor comes from woodworking, 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.
- As another example, we can write a Deck method named shuffle using the function shuffle from the random module (Don't forget to import random.):



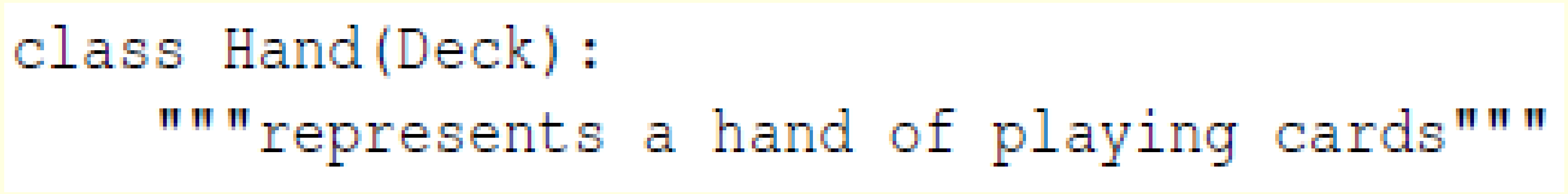
```
# inside class Deck:

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

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.
- As an example, let’s say we want a class to represent a “hand,” that is, the set of cards held by one player. A hand is similar to a deck: both are made up of a set 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.

Inheritance (cont'd)

- This relationship between classes—similar, but different—lends itself to inheritance.
- The definition of a child class is like other class definitions, but the name of the parent class appears in parentheses:

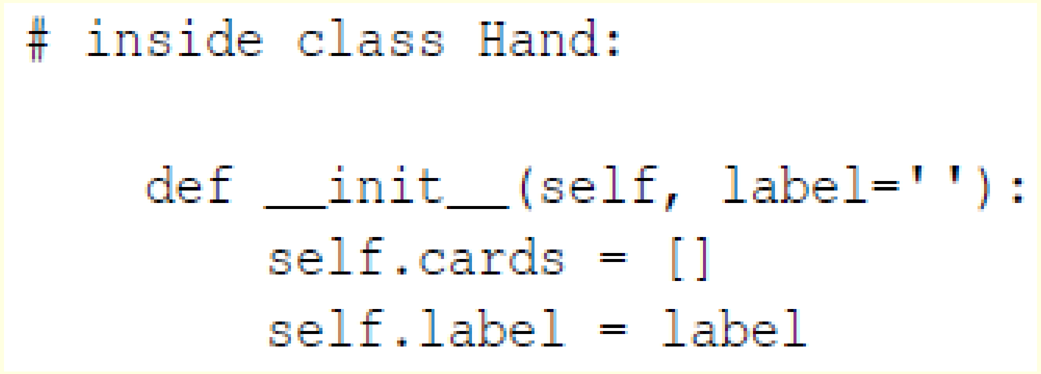


```
class Hand(Deck):
    """represents a hand of playing cards"""
```

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

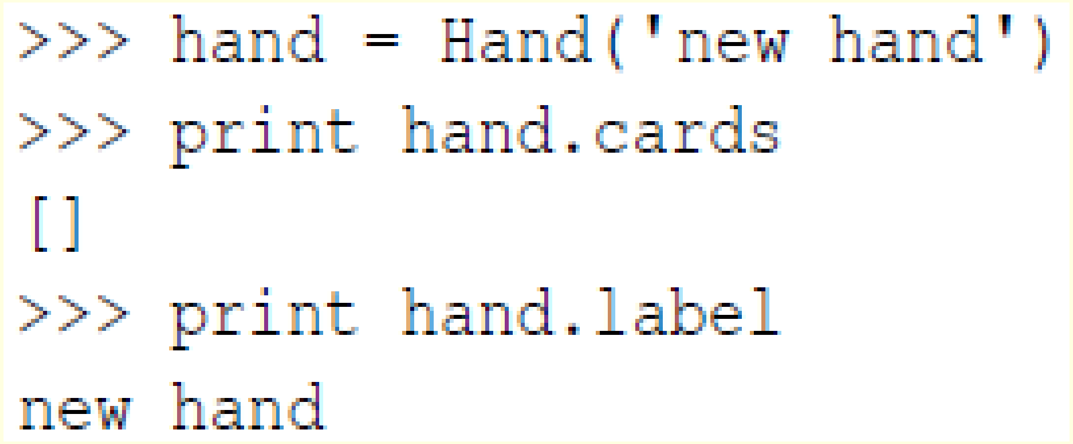
Inheritance (cont'd)

- 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
```

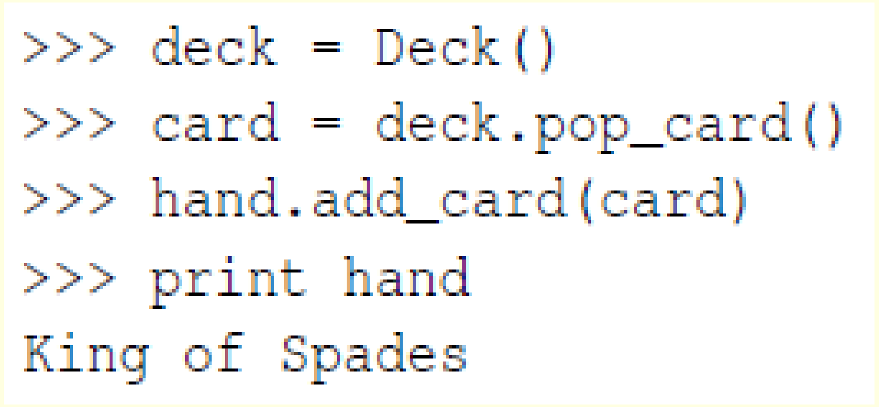
- So when you create a Hand, Python invokes this init method:



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

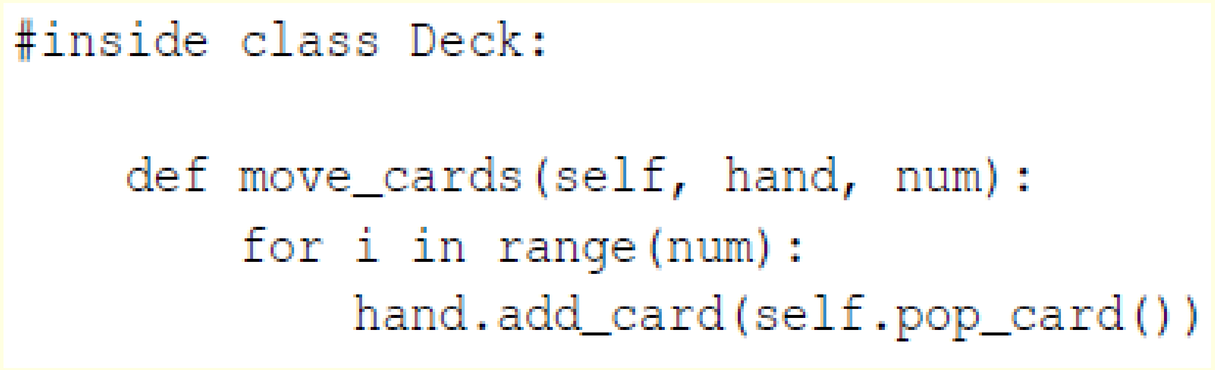
Inheritance (cont'd)

- But the other methods are inherited from Deck, so we can use pop_card and 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())
```

Inheritance (cont'd)

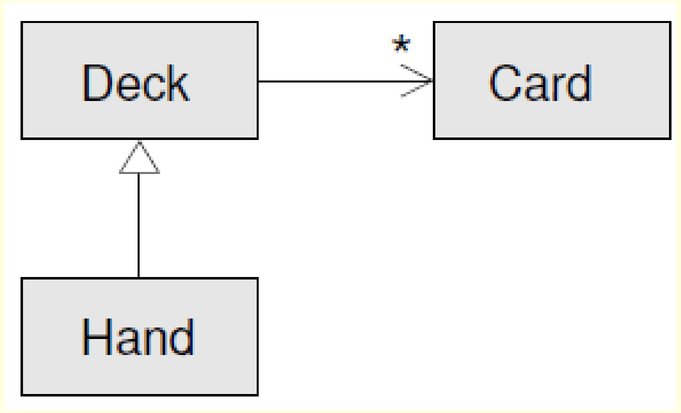
- move_cards takes two arguments, a Hand object and the number of cards to deal. It modifies both self and hand, and returns None.
- In some games, cards are moved from one hand to another, or from a hand back to the deck. You can use move_cards for any of these operations: self can be either a Deck or a Hand, and hand, despite the name, can also be a Deck.
- Inheritance is a useful feature. Some programs that would be repetitive without inheritance can be written more elegantly with it. Inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them. In some cases, the inheritance structure reflects the natural structure of the problem, which makes the program 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 scattered among 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 diagrams is a more abstract representation of the structure of a program. Instead of showing individual objects, it shows classes and the relationships between them.**
- **There are several kinds of relationship between classes:**
  - Objects in one class might contain references to objects in another class. For example, each Rectangle contains a reference to a Point, and each Deck contains references to many Cards. This kind of relationship is called HAS-A, as in, “a Rectangle has a Point.”
  - One class might inherit from another. This relationship is called IS-A, as in, “a Hand is a kind of a Deck.”
  - One class might depend on another in the sense that changes in one class would require changes in the other.

Class diagrams (cont'd)

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



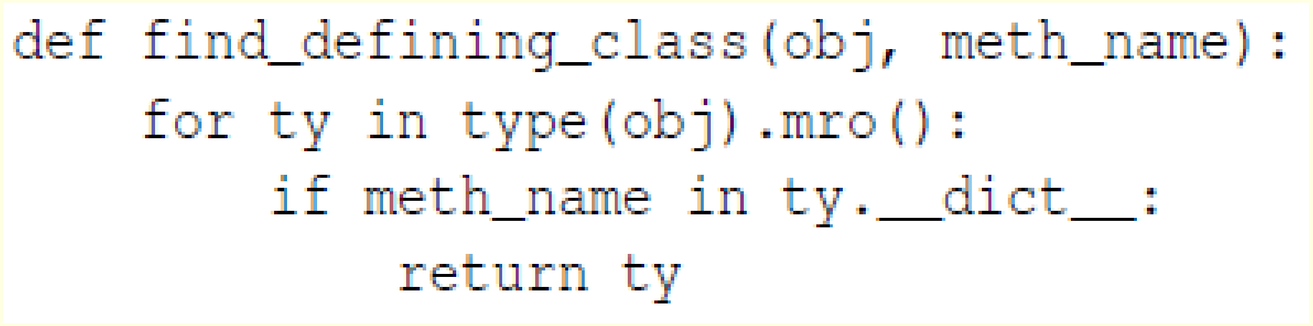
- The arrow with a hollow triangle head represents an IS-A relationship; in this case it indicates that Hand inherits from Deck.
- The standard arrow head represents a HAS-A relationship; in this case a Deck has references to Card objects.
- The star (*) near the arrow head is a multiplicity; it indicates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

Debugging

- Inheritance can make debugging a challenge because when you invoke a method on an object, you might not know 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 subclasses override this method, you'll get that version instead.
- Any time you are unsure about the flow of execution through your program, the simplest 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.

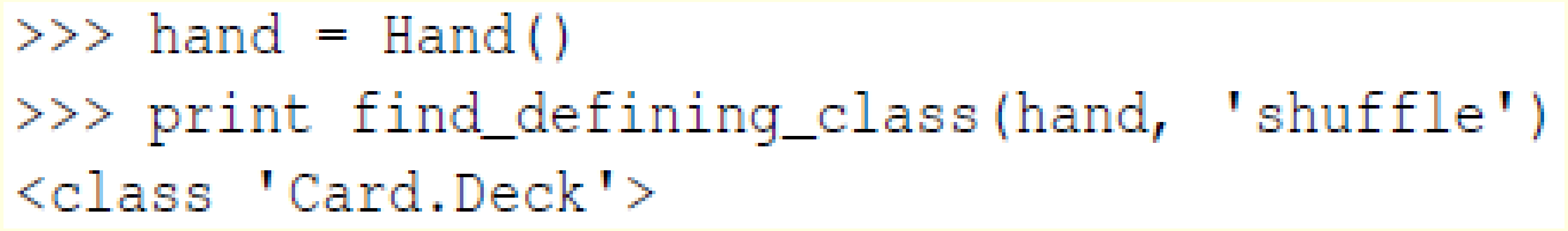
Debugging (cont'd)

- 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()
>>> print find_defining_class(hand, 'shuffle')
<class 'Card.Deck'>
```

- find_defining_class uses the mro method to get the list of class objects (types) that will be searched for methods. "MRO" stands for "method resolution order."

Debugging (cont'd)

- Here's a program design suggestion: whenever you override a method, the interface of the new method should be the same as the old. It should take the same parameters, return the same type, and obey the same preconditions and postconditions. If you obey this rule, you will find that any function designed to work with an instance of a superclass, like a Deck, will also work with instances of subclasses like a Hand or PokerHand.
- If you violate this rule, your code will collapse like (sorry) a house of cards.