_C12_POO-Ch17&18-Polymorphism&Inheritance_90

May 25, 2020

1 Chapter 17. Classes and Methods

1.1 Type-based dispatch

In the previous section, *Operator overloading*, we added two Time objects, but we also might want to add an integer to a Time object, meaning a class (user-defined) object and a different type object.

The following is a version of **__add__** that **checks** the type of *other* and **invokes EITHER add_time OR increment**. This operation is called a ***type-based dispatch*** because it *dispatches / sends / delegates* the computation to different method *based on the type* of arguments.

THIS IMPLEMENTATION OF addition IS NOT COMMUTATIVE. The solution is based on the special method **__radd__** (right-side add). This method is invoked when *a class object appears on the right side of the + operator*.

```
In [5]: class Time(object):
            def __init__(self,hour=0,minute=0,second=0):
                    self.hour = hour
                    self.minute = minute
                    self.second =second
            def __str__(self):
                return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
            def time_to_int(self):
                return self.hour*3600+self.minute*60+self.second

            def __add__(self, other):
                # the built-in function "isinstance" takes a value and a class object,
                # and returns True if the value is an instance of that class
                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)
```

```
        def __radd__(self, other):
            return self.__add__(other)

    def int_to_time(seconds):
        t = Time()
        minutes,t.second = divmod(seconds,60)
        t.hour,t.minute = divmod(minutes,60)
        return t

    start = Time(9,45)
    duration = Time(1,35)
    print start + duration

    print start + 1337
    print 1337 + start
```

```
11:20:00
10:07:17
10:07:17
```

1.2 Polymorphism

Using the OOP feature POLYMORPHISM, we can avoid the type-based dispatch by writing functions that work correctly for **arguments with different types**.

```
In [18]: 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)
```

```
Out[18]: {'bacon': 1, 'egg': 1, 'spam': 4}
```

Functions that **can 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.

```
In [8]: #Since Time objects provide an add method, they work with sum
        class Time(object):
            def __init__(self, hour=0, minute=0, second=0):
```

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

        def __add__(self, other):
            # the built-in function "isinstance" takes a value and a class object,
            # and returns True if the value is an instance of that class
            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)
        def __radd__(self, other):
            return self.__add__(other)

        def time_to_int(time):
            minutes = time.hour * 60 + time.minute
            seconds = minutes * 60 + time.second
            return seconds
    def int_to_time(seconds):
        time = Time()
        minutes, time.second = divmod(seconds, 60)
        time.hour, time.minute = divmod(minutes, 60)
        return time

    t1 = Time(7, 43)
    t2 = Time(7, 41)
    t3 = Time(7, 37)
    #total = t1 + t2 + t3 #uses the __add__ method, to add two Time objects repeatedly
    total = sum([t1, t2, t3]) #uses the __radd__ method !!!
    print total
```

```
23:01:00
```

In general, if all of the operations inside a function work with a given type, then the function 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.

1.3 Interface and implementation

. . . next course. . .

2 Inheritance

2.1 Class attributes

The **class attributes** are variables which are defined INSIDE a class but OUTSIDE of any method. ***They are associated with the class object***.

All the simialr variables we have had before are called **instance attributes** because they are associated with A PARTICULAR INSTANCE / OBJECT.

Both kinds of attribute are accessed using *dot notation*, but the "owner" / the subject is a class or an instance, respectively.

```
In [12]: class Card(object):
             """standard playing card"""

             def __init__(self, suit=0, rank=2):
                 #instance attributes
                 self.suit=suit
                 self.rank=rank

             #class attributes
             rank_names=[None,'Ace','Two','Three','Four','Five','Six','Seven','Eight','Nine','Te
             #the first element is None and it is a "place-keeper" :)
             suit_names=['Clubs','Diamonds','Hearts','Spades']

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

         card1 = Card(int(raw_input('suit=')),int(raw_input('rank=')))
         card2 = Card(int(raw_input('suit=')),int(raw_input('rank=')))

         # Every card has its own suit and rank - attributes for each instance,
         # BUT there is ONLY ONE COPY of suit_names and rank_names - attributes for the class

         print card1
         print card2

suit=2
rank=12
suit=2
rank=13
Queen of Hearts
King of Hearts
```

Note that the OBJECT DIAGRAM for the Card instances is described in the handbook!!!

2.2 Comparing objects

For user-defined types, we can **override** the behavior of the built-in operators by providing a method named **__cmp__**. It 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.

For the interest of this topic, we assume that suit is more important than rank.

```
In [14]: class Card(object):
             """standard playing card"""

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

             rank_names=[None,'Ace','Two','Three','Four','Five','Six','Seven','Eight','Nine','Te
             suit_names=['Clubs','Diamonds','Hearts','Spades']

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

             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, then check the ranks
                 if self.rank > other.rank:return 1
                 if self.rank < other.rank:return -1
                 #both suits and ranks are the same... it's a tie
                 return 0

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

         card1 = Card(int(raw_input('suit=')),int(raw_input('rank=')))
         card2 = Card(int(raw_input('suit=')),int(raw_input('rank=')))

         if card1 > card2:
             print card1, '>', card2
         elif card1 < card2:
             print card1, '<', card2
         else:
             print card1, '=', card2

         print card1 > card2
```

```
suit=2
rank=12
suit=3
rank=12
Queen of Hearts < Queen of Spades
False
```

Note that in Python 3, the function *cmp()* on tuples no longer exists and the **__cmp__** method is no supported!!! Instead, we could provide **__lt__**, which returns True if *self* is less than *other*. We can implement **__lt__** using tuples and the < operator.

3 Decks

A **deck** is made up of cards. So, each deck contains *a list of cards* as **an attribute**.

```
In [16]: import random

         class Deck(object):
             """It's a deck!"""

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

             def __str__(self):
                 res=[]
                 for card in self.cards:
                     res.append(str(card))
                     #the built-in function "str" invokes the __str__ method on each card
                 return '\n'.join(res)# joins the list' elements separated by newlines

             #removes the last card from the deck and returns it
             def pop_card(self):
                 return self.cards.pop()

             # add a card
             def add_card(self,card):
                 return self.cards.append(card)

             # shuffle the cards of a deck
             def shuffle_cards(self):
                 random.shuffle(self.cards)

         pachet = Deck()
```

```
        print pachet
        print '----------------------------------------'

        card = pachet.pop_card()
        print 'Extrag ultima carte din pachet: ', card
        print pachet
        print '----------------------------------------'

        pachet.shuffle_cards()
        print 'Pachetul amestecat: \n', pachet
        print '----------------------------------------'

        print 'Intorc cartea in pachet, la sfarsit'
        pachet.add_card(card)
        print pachet
```

```
Ace of Clubs
Two of Clubs
Three of Clubs
Four of Clubs
Five of Clubs
Six of Clubs
Seven of Clubs
Eight of Clubs
Nine of Clubs
Ten of Clubs
Jack of Clubs
Queen of Clubs
King of Clubs
Ace of Diamonds
Two of Diamonds
Three of Diamonds
Four of Diamonds
Five of Diamonds
Six of Diamonds
Seven of Diamonds
Eight of Diamonds
Nine of Diamonds
Ten of Diamonds
Jack of Diamonds
Queen of Diamonds
King of Diamonds
Ace of Hearts
Two of Hearts
Three of Hearts
Four of Hearts
Five of Hearts
Six of Hearts
```

```
Seven of Hearts
Eight of Hearts
Nine of Hearts
Ten of Hearts
Jack of Hearts
Queen of Hearts
King of Hearts
Ace of Spades
Two of Spades
Three of Spades
Four of Spades
Five of Spades
Six of Spades
Seven of Spades
Eight of Spades
Nine of Spades
Ten of Spades
Jack of Spades
Queen of Spades
King of Spades
----------------------------------------
Extrag ultima carte din pachet:  King of Spades
Ace of Clubs
Two of Clubs
Three of Clubs
Four of Clubs
Five of Clubs
Six of Clubs
Seven of Clubs
Eight of Clubs
Nine of Clubs
Ten of Clubs
Jack of Clubs
Queen of Clubs
King of Clubs
Ace of Diamonds
Two of Diamonds
Three of Diamonds
Four of Diamonds
Five of Diamonds
Six of Diamonds
Seven of Diamonds
Eight of Diamonds
Nine of Diamonds
Ten of Diamonds
Jack of Diamonds
Queen of Diamonds
King of Diamonds
```

```
Ace of Hearts
Two of Hearts
Three of Hearts
Four of Hearts
Five of Hearts
Six of Hearts
Seven of Hearts
Eight of Hearts
Nine of Hearts
Ten of Hearts
Jack of Hearts
Queen of Hearts
King of Hearts
Ace of Spades
Two of Spades
Three of Spades
Four of Spades
Five of Spades
Six of Spades
Seven of Spades
Eight of Spades
Nine of Spades
Ten of Spades
Jack of Spades
Queen of Spades
----------------------------------------
Pachetul amestecat:
Ten of Spades
Six of Spades
Three of Spades
Seven of Clubs
Nine of Hearts
King of Hearts
Three of Clubs
Queen of Hearts
Six of Clubs
Eight of Clubs
Jack of Spades
Ten of Clubs
Six of Hearts
Queen of Diamonds
Nine of Clubs
Five of Spades
King of Diamonds
Three of Hearts
Five of Diamonds
Eight of Hearts
Eight of Spades
```

```
Ace of Hearts
Five of Clubs
Four of Hearts
King of Clubs
Ace of Clubs
Six of Diamonds
Three of Diamonds
Four of Diamonds
Seven of Hearts
Queen of Spades
Jack of Hearts
Nine of Spades
Two of Spades
Seven of Spades
Ace of Spades
Jack of Clubs
Ten of Hearts
Four of Spades
Four of Clubs
Ace of Diamonds
Seven of Diamonds
Two of Diamonds
Two of Hearts
Five of Hearts
Queen of Clubs
Eight of Diamonds
Two of Clubs
Jack of Diamonds
Ten of Diamonds
Nine of Diamonds
----------------------------------------
Intorc cartea in pachet, la sfarsit
Ten of Spades
Six of Spades
Three of Spades
Seven of Clubs
Nine of Hearts
King of Hearts
Three of Clubs
Queen of Hearts
Six of Clubs
Eight of Clubs
Jack of Spades
Ten of Clubs
Six of Hearts
Queen of Diamonds
Nine of Clubs
Five of Spades
```

```
King of Diamonds
Three of Hearts
Five of Diamonds
Eight of Hearts
Eight of Spades
Ace of Hearts
Five of Clubs
Four of Hearts
King of Clubs
Ace of Clubs
Six of Diamonds
Three of Diamonds
Four of Diamonds
Seven of Hearts
Queen of Spades
Jack of Hearts
Nine of Spades
Two of Spades
Seven of Spades
Ace of Spades
Jack of Clubs
Ten of Hearts
Four of Spades
Four of Clubs
Ace of Diamonds
Seven of Diamonds
Two of Diamonds
Two of Hearts
Five of Hearts
Queen of Clubs
Eight of Diamonds
Two of Clubs
Jack of Diamonds
Ten of Diamonds
Nine of Diamonds
King of Spades
```

4 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*.

```
In [17]: #The definition of a child class is like other class definitions,
         #but the name of the parent class appears in parentheses
```

```
class Hand(Deck):
    """it's the hand extracted from a deck"""
```

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__**, but IT DOESN'T REALLY DO WHAT WE WANT for a Hand: to initialize cards with an empty list (instead of populating the hand with 52 new cards, like it does for a Deck). So, we provide a specific **__init__** method in the Hand class and **it OVERRIDES the one in the Deck class**.

```
In [18]: class Hand(Deck):
             """it's the hand extracted from a deck"""
             #we have a new attribute for a Hand object - label !!!
             def __init__(self,label=''):
                 self.cards=[]
                 self.label=label

         deck=Deck()

         card=deck.pop_card()

         #the __init__ method is overrided
         hand=Hand('new hand')
         print hand.cards
         print hand.label

         #the other methods are inherited from the parent Deck
         hand.add_card(card)
         print hand
```

```
[]
new hand
King of Spades
```

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

```
In [ ]: def move_cards(self, hand, num):
            for i in range(num):
                hand.add_card(self.pop_card())
        #move_cards takes two arguments: a Hand object and the number of cards to deal;
        #it modifies both self and hand and returns None
```

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

Inheritance is a useful feature. Some programs that would be repetitive without inheritance can be written more elegantly with it. Inheritance **can facilitate code reuse**, since you can *customize*

the behavior of parent classes without having to modify them. In some cases, the inheritance structure reflects the natural structure of the problem, which makes the 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.

5 Class diagram

Note the CLASS DIAGRAM in the handbook!!!

It is a more abstract representation of the structure of a program than the object diagrams and/or the stack diagrams (to show the state of a program). Instead of showing individual objects, a class diagram **shows classes and the relationships between them**.

The main relations between classes are - HAS-A for encapsulation and - IS-A for inheritance.

The * (star) is a multiplicity - a class has many references to the other class objects.

6 Data encapsulation

... next course... with *Interface and Implementation* and *Refactoring*...