22. Collections of objects

22.1. Composition

By now, we have seen several examples of composition. One of the first examples was using a method invocation as part of an expression. Another example is the nested structure of statements; we can put an `if` statement within a `while` loop, within another `if` statement, and so on.

Having seen this pattern, and having learned about lists and objects, we should not be surprised to learn that we can create lists of objects. We can also create objects that contain lists (as attributes); we can create lists that contain lists; we can create objects that contain objects; and so on.

In this chapter and the next, we will look at some examples of these combinations, using `Card` objects as an example.

22.2. `Card` objects

If you are not familiar with common playing cards, now would be a good time to get a deck, or else this chapter might not make much sense. 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 we are playing, the rank of Ace may be higher than King or lower than 2. The rank is sometimes called the face-value of the card.

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. By encode, we do not mean what some people think, which is to encrypt or translate into a secret code. What a computer scientist means by encode is to define a mapping between a sequence of numbers and the items I want to represent. For example:

```
Spades   -->  3
Hearts   -->  2
Diamonds -->  1
Clubs    -->  0
```

An obvious feature of this mapping is that the suits map to integers in order, so we can compare suits by comparing integers. The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

```
Jack   -->  11
Queen  -->  12
King   -->  13
```

The reason we are using mathematical notation for these mappings is that they are not part of the Python program. They are part of the program design, but they never appear explicitly in the code. The class definition for the `Card` type looks like this:

```python
class Card:
    def __init__(self, suit=0, rank=0):
        self.suit = suit
        self.rank = rank
```

As usual, we provide an initialization method that takes an optional parameter for each attribute.

To create some objects, representing say the 3 of Clubs and the Jack of Diamonds, use these commands:

```
three_of_clubs = Card(0, 3)
card1 = Card(1, 11)
```

In the first case above, for example, the first argument, `0`, represents the suit Clubs.

Save this code for later use ...

In the next chapter we assume that we have save the `Cards` class, and the upcoming `Deck` class in a file called `Cards.py`.

22.3. Class attributes and the `__str__` method

In order to print `Card` objects in a way that people can easily read, we want to map the integer codes onto words. A natural way to do that is with lists of strings. We assign these lists to **class attributes** at the top of the class definition:

```
class Card:
    suits = ["Clubs", "Diamonds", "Hearts", "Spades"]
    ranks = ["narf", "Ace", "2", "3", "4", "5", "6", "7",
             "8", "9", "10", "Jack", "Queen", "King"]

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

    def __str__(self):
        return (self.ranks[self.rank] + " of " + self.suits[self.suit])
```

A class attribute is defined outside of any method, and it can be accessed from any of the methods in the class.

Inside `__str__`, we can use `suits` and `ranks` to map the numerical values of `suit` and `rank` to strings. For example, the expression `self.suits[self.suit]` means use the attribute `suit` from the object `self` as an index into the class attribute named `suits`, and select the appropriate string.

The reason for the `"narf"` in the first element in `ranks` is to act as a place keeper for the zero-eth element of the list, which will never be used. The only valid ranks are 1 to 13. This wasted item is not entirely necessary. We could have started at 0, as usual, but it is less confusing to encode the rank 2 as integer 2, 3 as 3, and so on.

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

```
>>> card1 = Card(1, 11)
>>> print(card1)
Jack of Diamonds
```

Class attributes like `suits` are shared by all `Card` objects. The advantage of this is that we can use any `Card` object to access the class attributes:

```
>>> card2 = Card(1, 3)
>>> print(card2)
3 of Diamonds
>>> print(card2.suits[1])
Diamonds
```

Because every `Card` instance references the same class attribute, we have an aliasing situation. The disadvantage is that if we modify a class attribute, it affects every instance of the class. For example, if we decide that Jack of Diamonds should really be called Jack of Swirly Whales, we could do this:

```
>>> card1.suits[1] = "Swirly Whales"
>>> print(card1)
Jack of Swirly Whales
```

The problem is that *all* of the Diamonds just became Swirly Whales:

```
>>> print(card2)
3 of Swirly Whales
```

It is usually not a good idea to modify class attributes.

22.4. Comparing cards

For primitive types, there are six relational operators (`<`, `>`, `==`, etc.) that compare values and determine when one is greater than, less than, or equal to another. If we want our own types to be comparable using the syntax of these relational operators, we need to define six corresponding special methods in our class.

We'd like to start with a single method named `cmp` that houses the logic of ordering. By convention, a comparison method takes two parameters, `self` and `other`, and returns 1 if the first object is greater, -1 if the second object is greater, and 0 if they are equal to each other.

Some types are completely ordered, which means that we can compare any two elements and tell which is bigger. For example, the integers and the floating-point numbers are completely ordered. Some types are unordered, which means that there is no meaningful way to say that one element is bigger than another. For example, the fruits are unordered, which is why we cannot compare apples and oranges, and we cannot meaningfully order a collection of images, or a collection of cellphones.

Playing cards are partially ordered, which means that sometimes we can compare cards and sometimes not. For example, we know that the 3 of Clubs is higher than the 2 of Clubs, and the 3 of Diamonds is higher than the 3 of Clubs. But 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 make cards comparable, we have to decide which is more important, rank or suit. To be honest, the choice is arbitrary. For the sake of choosing, we will say that suit is more important, because a new deck of cards comes sorted with all the Clubs together, followed by all the Diamonds, and so on.

With that decided, we can write `cmp`:

```
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
```

In this ordering, Aces appear lower than Deuces (2s).

Now, we can define the six special methods that do the overloading of each of the relational operators for us:

```
def __eq__(self, other):
    return self.cmp(other) == 0

def __le__(self, other):
    return self.cmp(other) <= 0

def __ge__(self, other):
    return self.cmp(other) >= 0

def __gt__(self, other):
    return self.cmp(other) > 0

```

```
def __lt__(self, other):
    return self.cmp(other) < 0

def __ne__(self, other):
    return self.cmp(other) != 0
```

With this machinery in place, the relational operators now work as we'd like them to:

```
>>> card1 = Card(1, 11)
>>> card2 = Card(1, 3)
>>> card3 = Card(1, 11)
>>> card1 < card2
False
>>> card1 == card3
True
```

22.5. Decks

Now that we have objects to represent `Card`s, the next logical step is to define a class to represent a `Deck`. Of course, a deck is made up of cards, so each `Deck` object will contain a list of cards as an attribute. Many card games will need at least two different decks — a red deck and a blue deck.

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

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

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. Since the outer loop iterates four times, and the inner loop iterates thirteen times, the total number of times the body is executed is fifty-two (thirteen times four). Each iteration creates a new instance of `Card` with the current suit and rank, and appends that card to the `cards` list.

With this in place, we can instantiate some decks:

```
red_deck = Deck()
blue_deck = Deck()
```

22.6. Printing the deck

As usual, when we define a new type we want a method that prints the contents of an instance. To print a `Deck`, we traverse the list and print each `Card`:

```
class Deck:
    ...
    def print_deck(self):
        for card in self.cards:
            print(card)
```

Here, and from now on, the ellipsis (`...`) indicates that we have omitted the other methods in the class.

As an alternative to `print_deck`, we could write a `__str__` method for the `Deck` class. The advantage of `__str__` is that it is more flexible. Rather than just printing the contents of the object, it generates a string representation that other parts of the program can manipulate before printing, or store for later use.

Here is a version of `__str__` that returns a string representation of a `Deck`. To add a bit of pizzazz, it arranges the cards in a cascade where each card is indented one space more than the previous card:

```python
class Deck:
    ...
    def __str__(self):
        s = ""
        for i in range(len(self.cards)):
            s = s + " " * i + str(self.cards[i]) + "\n"
        return s
```

This example demonstrates several features. First, instead of traversing `self.cards` and assigning each card to a variable, we are using `i` as a loop variable and an index into the list of cards.

Second, we are using the string multiplication operator to indent each card by one more space than the last. The expression `" " * i` yields a number of spaces equal to the current value of `i`.

Third, instead of using the `print` command to print the cards, we use the `str` function. Passing an object as an argument to `str` is equivalent to invoking the `__str__` method on the object.

Finally, we are using the variable `s` as an **accumulator**. Initially, `s` is the empty string. Each time through the loop, a new string is generated and concatenated with the old value of `s` to get the new value. When the loop ends, `s` contains the complete string representation of the `Deck`, which looks like this:

```
>>> red_deck = Deck()
>>> print(red_deck)
Ace of Clubs
 2 of Clubs
  3 of Clubs
   4 of Clubs
     5 of Clubs
       6 of Clubs
        7 of Clubs
         8 of Clubs
          9 of Clubs
           10 of Clubs
            Jack of Clubs
             Queen of Clubs
              King of Clubs
               Ace of Diamonds
                2 of Diamonds
                 ...
```

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

22.7. Shuffling the deck

If a deck is perfectly shuffled, then any card is equally likely to appear anywhere in the deck, and any location in the deck is equally likely to contain any card.

To shuffle the deck, we will use the `randrange` function from the `random` module. With two integer arguments, `a` and `b`, `randrange` chooses a random integer in the range `a <= x < b`. Since the upper bound is strictly less than `b`, we can use the length of a list as the second parameter, and we are guaranteed to get a legal index. For example, if `rng` has already been instantiated as a random number source, this expression chooses the index of a random card in a deck:

```python
rng.randrange(0, len(self.cards))
```

An easy way to shuffle the deck is by traversing the cards and swapping each card with a randomly chosen one. It is possible that the card will be swapped with itself, but that is fine. In fact, if we precluded that possibility, the order of the cards would be less than entirely random:

```python
class Deck:
    ...
    def shuffle(self):
        import random
        rng = random.Random()         # Create a random generator
        num_cards = len(self.cards)
        for i in range(num_cards):
            j = rng.randrange(i, num_cards)
            (self.cards[i], self.cards[j]) = (self.cards[j], self.cards[i])
```

Rather than assume that there are fifty-two cards in the deck, we get the actual length of the list and store it in `num_cards`.

For each card in the deck, we choose a random card from among the cards that haven't been shuffled yet. Then we swap the current card (`i`) with the selected card (`j`). To swap the cards we use a tuple assignment:

```
(self.cards[i], self.cards[j]) = (self.cards[j], self.cards[i])
```

While this is a good shuffling method, a random number generator object also has a `shuffle` method that can shuffle elements in a list, in place. So we could rewrite this function to use the one provided for us:

```
class Deck:
    ...
    def shuffle(self):
        import random
        rng = random.Random()        # Create a random generator
        rng.shuffle(self.cards)      # uUse its shuffle method
```

22.8. Removing and dealing cards

Another method that would be useful for the `Deck` class is `remove`, which takes a card as a parameter, removes it, and returns `True` if the card was in the deck and `False` otherwise:

```
class Deck:
    ...
    def remove(self, card):
        if card in self.cards:
            self.cards.remove(card)
            return True
        else:
            return False
```

The `in` operator returns `True` if the first operand is in the second. If the first operand is an object, Python uses the object's `__eq__` method to determine equality with items in the list. Since the `__eq__` we provided in the `Card` class checks for deep equality, the `remove` method checks for deep equality.

To deal cards, we want to remove and return the top card. The list method `pop` provides a convenient way to do that:

```
class Deck:
    ...
    def pop(self):
        return self.cards.pop()
```

Actually, `pop` removes the *last* card in the list, so we are in effect dealing from the bottom of the deck.

One more operation that we are likely to want is the Boolean function `is_empty`, which returns `True` if the deck contains no cards:

```python
class Deck:
    ...
    def is_empty(self):
        return self.cards == []
```

22.9. Glossary

encode
: To represent one type of value using another type of value by constructing a mapping between them.

class attribute
: A variable that is defined inside a class definition but outside any method. Class attributes are accessible from any method in the class and are shared by all instances of the class.

accumulator
: A variable used in a loop to accumulate a series of values, such as by concatenating them onto a string or adding them to a running sum.

22.10. Exercises

1. Modify `cmp` so that Aces are ranked higher than Kings.