Think Python 2e, Chapter 18 Notes

Inheritance

November 11, 2022

Card objects

Encoding of suits:

Spades	$\rightarrow$	3
Hearts	$\rightarrow$	2
Diamonds	$\rightarrow$	1
Clubs	$\rightarrow$	0

Encoding of ranks:

Ace	$\rightarrow$	1
...		
Jack	$\rightarrow$	11
Queen	$\rightarrow$	12
King	$\rightarrow$	13

Class definition of card:

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

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

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

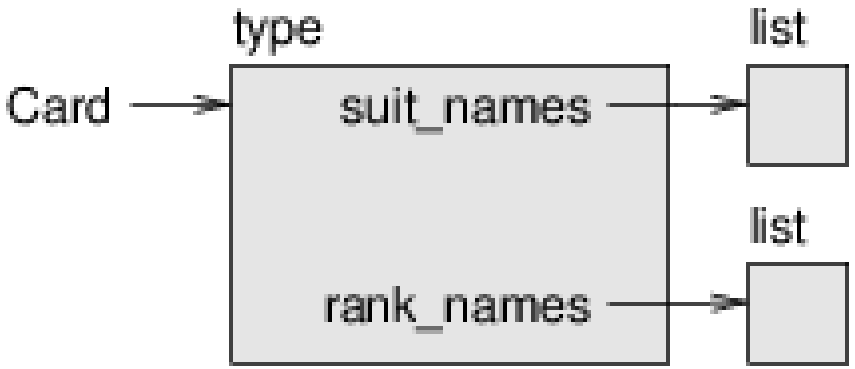
Class attributes

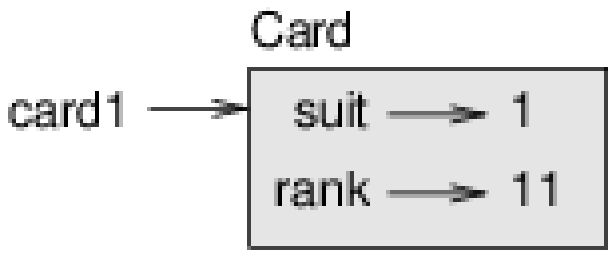
```
# inside class Card:

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 declared inside a class but outside of any method, like `suit_names` and `rank_names` are **class variables**.
- They are associated with the class `Card`
- Variables like `suit` and `rank` are **instance variables**

Creating cards

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





Comparing cards

```
# 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
```

More succinctly:

```
# inside class Card:

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

Decks

```
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(card)
```

Printing a deck

```
# inside class Deck:

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

```
>>> 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
```

Add, remove, and shuffle cards

```
# inside class Deck:

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

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

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

- A method like these that doesn't do much is called a **veneer**.
- It improves the interface of the implementation.

Inheritance

```
1 class Hand(Deck):
2     """Represents a hand of playing cards."""
3
4     def __init__(self, label=''):
5         self.cards = []
6         self.label = label
```

- `Hand` **inherits** all attributes and methods from `Deck`.
- `Deck` is the **parent** and `Hand` is the **child**.

Inheritance

```
class Hand(Deck):
    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

```
>>> hand = Hand('new hand')
>>> hand.cards
[]
>>> hand.label
'new hand'
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)
King of Spades
```

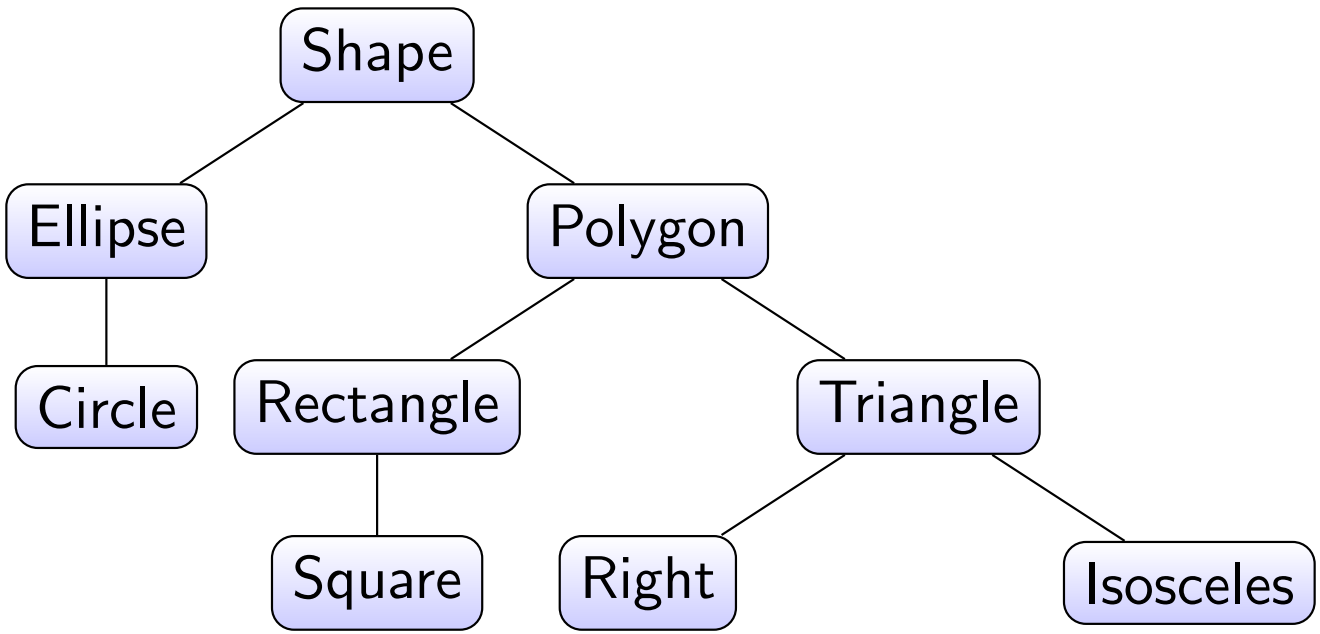
- `__str__` and `add_card` don't have to be defined for `Hand`

Inheritance

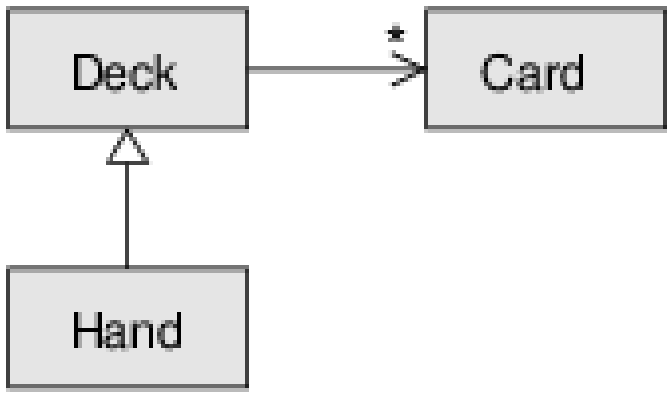
```
# inside class Deck:

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

Inheritance

- We can now move cards from the deck to a hand, from a hand to a hand, from a hand to the deck, *etc.*
- Inheritance can facilitate code reuse.
- Inheritance sometimes mimics the natural structure of a problem.
- Inheritance can make it difficult to debug.



Class diagrams



Class relationships:

- A deck **HAS-A** card
  - the * shows **multiplicity**
- A hand **IS-A** deck
- A class required by another class for other reasons is a **dependency**
  - *e.g.* used as parameters or for internal computation
  - shown with dotted lines

Debugging

- Inheritance can make it hard to find which method is invoked.
- Can use this function:

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

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

- `mro` stands for "method resolution order"

Liskov substitution principle

- When you override a method:
  - new method should be same as old method
  - take same parameters
  - return same type
  - obey same preconditions
  - obey same postconditions
- Any function designed to work with a parent class will also work with a child class.

Data encapsulation

- In the `markov` function from chapter 13 we used two global variables:
  - `suffix_map = dict()`
  - `prefix = ()`
- If we wanted to read two texts in the same program their prefixes and suffixes would get mixed up.
- Encapsulate each with an object:

```
1 class Markov:
2
3     def __init__(self):
4         self.suffix_map = dict()
5         self.prefix = ()
```

Data encapsulation

- We now translate functions with global variables into methods:

```
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 no entry for this prefix, make one
        self.suffix_map[self.prefix] = [word]

    self.prefix = shift(self.prefix, word)
```

- An important example of **refactoring**

Object oriented development

- Start with global variables when necessary
- Finish working program
- Look for associations between global variables
- Encapsulate related variables as attributes
- Transform associated functions into methods
- We will do this wiht the Mandelbrot program!

Vocabulary

encode: To represent one set of values using another set of values by constructing a mapping between them.

class attribute: An attribute associated with a class object. Class attributes are defined inside a class definition but outside any method.

instance attribute: An attribute associated with an instance of a class.

veneer: A method or function that provides a different interface to another function without doing much computation.

Vocabulary

inheritance: The ability to define a new class that is a modified version of a previously defined class.

parent class: The class from which a child class inherits.

child class: A new class created by inheriting from an existing class; also called a "subclass".

Vocabulary

IS-A relationship: A relationship between a child class and its parent class.

HAS-A relationship: A relationship between two classes where instances of one class contain references to instances of the other.

dependency: A relationship between two classes where instances of one class use instances of the other class, but do not store them as attributes.

Vocabulary

class diagram: A diagram that shows the classes in a program and the relationships between them.

multiplicity: A notation in a class diagram that shows, for a HAS-A relationship, how many references there are to instances of another class.

Vocabulary

data encapsulation: A program development plan that involves a prototype using global variables and a final version that makes the global variables into instance attributes.