DM536
Introduction to Programming

Peter Schneider-Kamp

petersk@imada.sdu.dk

http://imada.sdu.dk/~petersk/DM536/

CLASSES & METHODS

Object-Oriented Features

- object-oriented programming in a nutshell:
  - programs consists of class definitions and functions
  - classes describe real or imagined objects
  - most functions and computations work on objects
- so far we have only used classes to store attributes
- i.e., functions were not linked to objects

- methods = functions defined inside a class definition
  - first argument is always the object the method belongs to
  - calling by using dot notation
  - Example: `"Slartibartfast".count("a")`

Printing Objects

- printing can be done by a normal function
- better done with a method
- Example:

```
class Time(object):
    """represents time of day using hours, minutes, seconds"""'
    def print_time(time):
        t = (time.hours, time.minutes, time.seconds)
        print "%02dh %02dm %02ds" % t

def print_time(time):
    t = (time.hours, time.minutes, time.seconds)
    print "%02dh %02dm %02ds" % t
```

Printing Objects

- printing can be done by a normal function
- better done with a method
- Example:

```
class Time(object):
    """represents time of day using hours, minutes, seconds"""'
    def print_time(self):
        t = (self.hours, self.minutes, self.seconds)
        print "%02dh %02dm %02ds" % t

def print_time(time):
    t = (time.hours, time.minutes, time.seconds)
    print "%02dh %02dm %02ds" % t
```

Printing Objects

- printing can be done by a normal function
- better done with a method
- Example:

```
class Time(object):
    """represents time of day using hours, minutes, seconds"""
    def print_time(self):
        t = (self.hours, self.minutes, self.seconds)
        print "%02dh %02dm %02ds" % t
end = Time()
end.hours = 12;  end.minutes = 15;  end.seconds = 37
Time.print_time(end)          # what really happens
end.print_time()              # how to write it!
```

Incrementing as a Method

- Example: add increment as a method

```
class Time(object):
    """represents time of day using hours, minutes, seconds"""
    def time_to_int(self):
        return self.seconds + 60 * (self.minutes + 60 * self.hours)
    def int_to_time(self, seconds):
        minutes, self.seconds = divmod(seconds, 60)
        self.hours, self.minutes = divmod(minutes, 60)
    def increment(self, seconds):
        return self.int_to_time(seconds + self.time_to_int())
```

Comparing with Methods

- Example: add is_after as a method

```
class Time(object):
    """represents time of day using hours, minutes, seconds""""
    def time_to_int(self):
        return self.seconds + 60 * (self.minutes + 60 * self.hours)
    def int_to_time(self, seconds):
        minutes, self.seconds = divmod(seconds, 60)
        self.hours, self.minutes = divmod(minutes, 60)
    def increment(self, seconds):
        return self.int_to_time(seconds + self.time_to_int())
    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

Initializing Objects

- special method __init__(self, …) to create new objects
- usually first method written for any new class!
- Example: initialize Time objects using __init__

```
class Time(object):
    """represents time of day using hours, minutes, seconds""""
    def __init__(self, hours, minutes, seconds):
        self.hours = hours
        self.minutes = minutes
        self.seconds = seconds
start = Time(12, 23, 42)
start = Time()
start.hours = 12;  start.minutes = 23; start.seconds = 42
```

String Representation of Objects

- special method __str__(self) to convert objects to strings
- Example: print Time objects using __str__

```
class Time(object):
    """represents time of day using hours, minutes, seconds"""'
    def __init__(self, hours, minutes, seconds):
        self.hours = hours
        self.minutes = minutes
        self.seconds = seconds
   def __str__(self):
        t = (self.hours, self.minutes, self.seconds)
        return "%dh %dm %ds" % t
print Time(7, 42, 23)
```

Representation of Objects

- special method __repr__(self) to represent objects
- Example: make Time objects more usable in lists

```
class Time(object):
    """represents time of day using hours, minutes, seconds"""
    def __str__(self):
        t = (self.hours, self.minutes, self.seconds)
        return "%dh %dm %ds" % t
    def __repr__(self):
        t = (self.hours, self.minutes, self.seconds)
        return "Time(%s, %s, %s)" % t
print [Time(7, 42, 23), Time(12, 23, 42)]
```

Representation of Objects

- special method __repr__(self) to represent objects
- Example: make Time objects more usable in lists

```
class Time(object):
    """represents time of day using hours, minutes, seconds""""
    def as_tuple(self):
        return (self.hours, self.minutes, self.seconds)
    def __str__(self):
        return "%dh %dm %ds" % self.as_tuple()
    def __repr__(self):
        return "Time(%s, %s, %s)" % self.as_tuple()
print [Time(7, 42, 23), Time(12, 23, 42)]
```

Overloading Operators

- special method __add__(self, other) to overload "+" operator
- likewise, you can use __mul__(self, other) etc.
- Example: add Time objects using __add__

```
class Time(object):
    """represents time of day using hours, minutes, seconds""""
    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return self.int_to_time(seconds)
t1 = Time(2, 40, 19)
t2 = Time(10, 2, 23)
print t1 + t2
```

Type-Based Dispatch

- we want to add both Time objects and seconds
- use isinstance(object, class) to determine type of argument
- Example:

```
class Time(object):
    def __add__(self, other):
        if isinstance(other, Time):   return self.add_time(other)
        else:                         return self.add_seconds(other)
    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return self.int_to_time(seconds)
    def add_seconds(self, seconds):
        return self.int_to_time(seconds + self.time_to_int())
```

Polymorphism

- polymorphic = working on different argument types

- Examples:
  - histogram(s) can be used for lists & tuples of elements, that can be used as dictionary keys

  - sum(t) can be used for lists & tuples of elements, for which "+" works, i.e., also for Time

- to use e.g. Time as dictionary keys, implement __hash__(self)
- important that returned integer identical for identical objects

Debugging by Introspection

- hard to work with objects where attributes are added
- try to always use __init__(self, …) to create attributes
- do not create attributes (or methods) from "outside"

- you can use dir(object) to get list of attributes and methods

- special attribute __dict__ maps attributes to values
- Example: print all atributes and their values and types

```
for var, value in time.__dict__.items():
    print "%s -> %s (%s)" % (var, value, type(value))
```

INHERITANCE

Card Objects

- **Goal:** represent cards as objects
- **Design:**
  - represent Spades, Hearts, Diamonds, Clubs by 3, 2, 1, 0
  - represent different cards by 1 … 10 and 11, 12, 13
- Example:

```
class Card(object):
    """represents a standard playing card"""
    def __init__(self, suit = 2, rank = 13)       # Queen of Hearts
        self.suit = suit
        self.rank = rank
queen_of_hearts = Card()
ten_of_spades = Card(3, 10)
```

Class Attributes

- class attribute = same for each object of a given class
- class attributes are defined by assignments inside the class
- Example:

```
class Card(object):
    """represents a standard playing card"""
    def __init__(self, suit = 2, rank = 13)       # Queen of Hearts
        self.suit = suit
        self.rank = rank
    suits = ["Clubs", "Diamonds", "Hearts", "Spades"]
    ranks = [None, "Ace", "2", "3", "4", "5", "6", "7", "8", "9", "10",
                "Jack", "Queen", "King"]
card = Card(suits.find("Diamonds"), ranks.find("Ace"))
```

Comparing Cards

- special method __cmp__(self, other) for comparing values
- return value 0 for equality, > 0 for greater, < 0 for smaller
- used by built-in function cmp(x, y)
- Example:

```
class Card(object):
    …
    def __cmp__(self, other):
        if self.suit > other. suit:        return  1
        if self.suit < other. suit:        return -1
        if self.rank > other. rank:        return  1
        if self.rank < other. rank:        return -1
        return 0
```

Comparing Cards

- special method __cmp__(self, other) for comparing values
- return value 0 for equality, > 0 for greater, < 0 for smaller
- used by built-in function cmp(x, y)
- Example:

```
class Card(object):
    …
    def __cmp__(self, other):
        return cmp((self.suit, self.rank), (other.suit, other.rank))
print queen_of_hearts > ten_of_spades      # False
```

Decks

- **Goal:** represent decks of cards
- **Design:** use a list of cards as attribute
- Example:

```
class Deck(object):
    """represents a deck as a list of cards"""
    def __init__(self):
        self.cards = []
        for suit in range(len(Card.suits)):
            for rank in range(1, len(Card.ranks)):
                card = Card(suit, rank)
                self.cards.append(card)
```

Printing Decks

- printing can be done using the __str__(self) method
- Example:

```
class Deck(object):
    """represents a deck as a list of cards"""
    …
    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(car))
        return "\n".join(res)
```

Popping and Adding a Card

- removing and adding are basic operations
- both can be implemented using list methods
- Example:

```
class Deck(object):
    """represents a deck as a list of cards"""
    …
    def pop_card(self):
        return self.cards.pop()
    def add_card(self, card):
        self.cards.append(card)
```

Shuffle a Deck

- likewise, functionality like shuffling can be implemented easily
- idea is to use shuffle(list) from random module
- Example:

```
import random
class Deck(object):
    """represents a deck as a list of cards"""
    …
    def shuffle(self):
        random.shuffle(self.cards)
deck = Deck()
deck.shuffle()
print deck
```

Inheritance

- inheritance = define new class as modification of old class
- old class is called *parent*, new class is called *child*
- useful e.g. for representing a hand based on a deck
- Example:

```
class Hand(Deck):
    """represents a hand of playing cards"""
    def __init__(self, label = ""):
        self.cards = []
        self.label = label
```

- Hand inherits all methods (including __init__) from Deck
- BUT: we do not want all cards in a hand
- Solution: override __init__ method

Move Cards from Deck to Hand

- cards can be moved using pop_card and add_card
- Example:

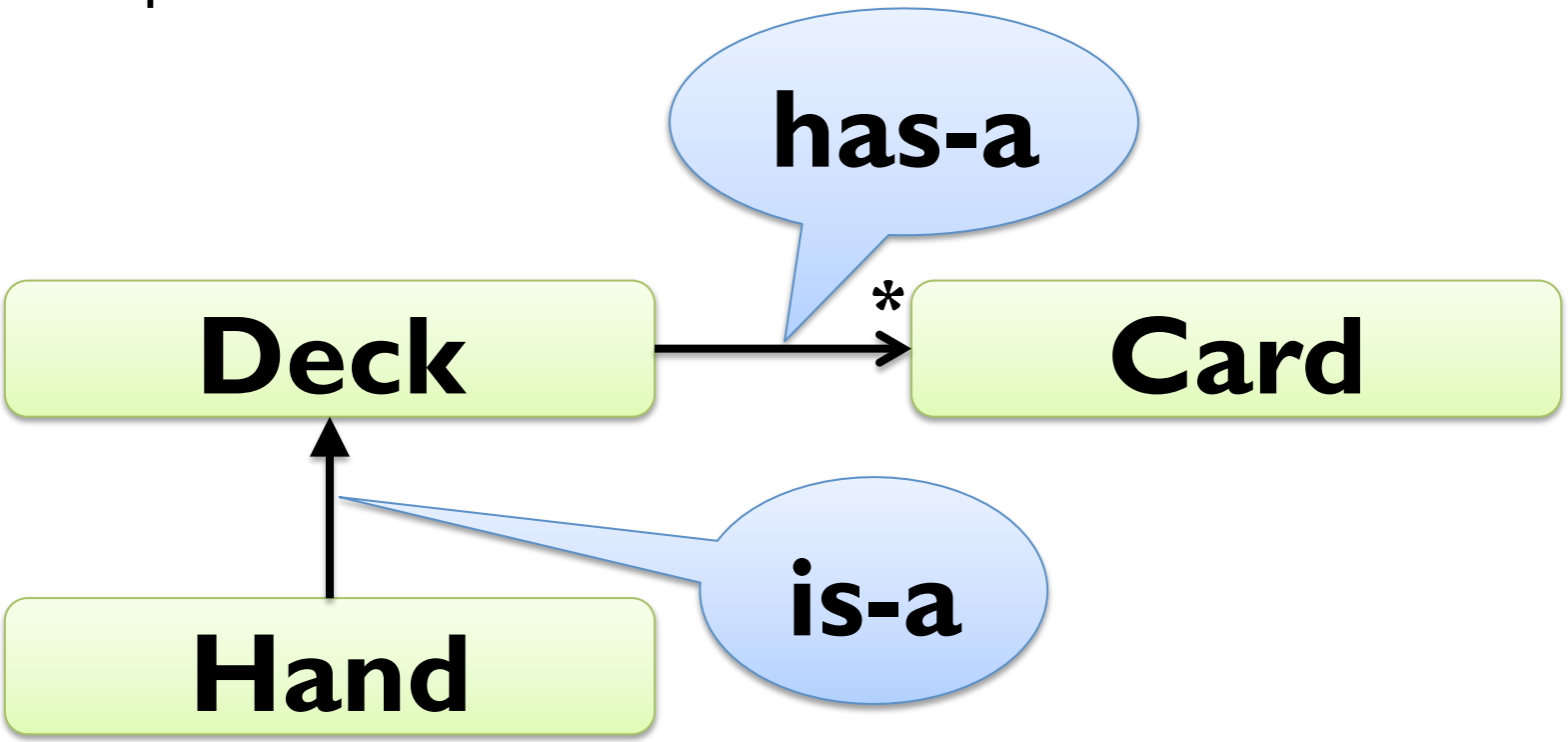
```
deck = Deck();  hand = Hand("my hand")
hand.add_card(deck.pop_card())
```

- tedious for giving a hand – better add a method to Deck
- Example:

```
class Deck(object):
    """represents a deck as a list of cards"""
    …
    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

Class Diagrams

- class diagram = family tree and friends of classes
- in contrast to state diagrams, class diagrams are static
- Example:



Debugging and Inheritance

- harder to determine control flow when using inheritance
- add print statements to methods to see which is called
- alternatively, use the following method:

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

- whenever you override a method, use the same contract
- same pre-conditions, same post-conditions, same argument list

The End

- we are finished with Python for this course
- you should understand and be able to use all concepts
- use some time to develop your Python skill
- list comprehensions, libraries for networking, …
- scratch your itches with Python
- … and if you continue with Programming B …