03-4 Inheritance

CSI 500

Spring 2018

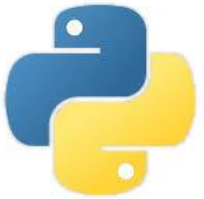
Course material derived from:
Downey, Allen B. 2012. "Think Python, 2nd Edition". O'Reilly Media Inc., Sebastopol CA.

"How to Think Like a Computer Scientist" by Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers. Oct 2012
http://openbookproject.net/thinkcs/python/english3e/index.html

Card objects

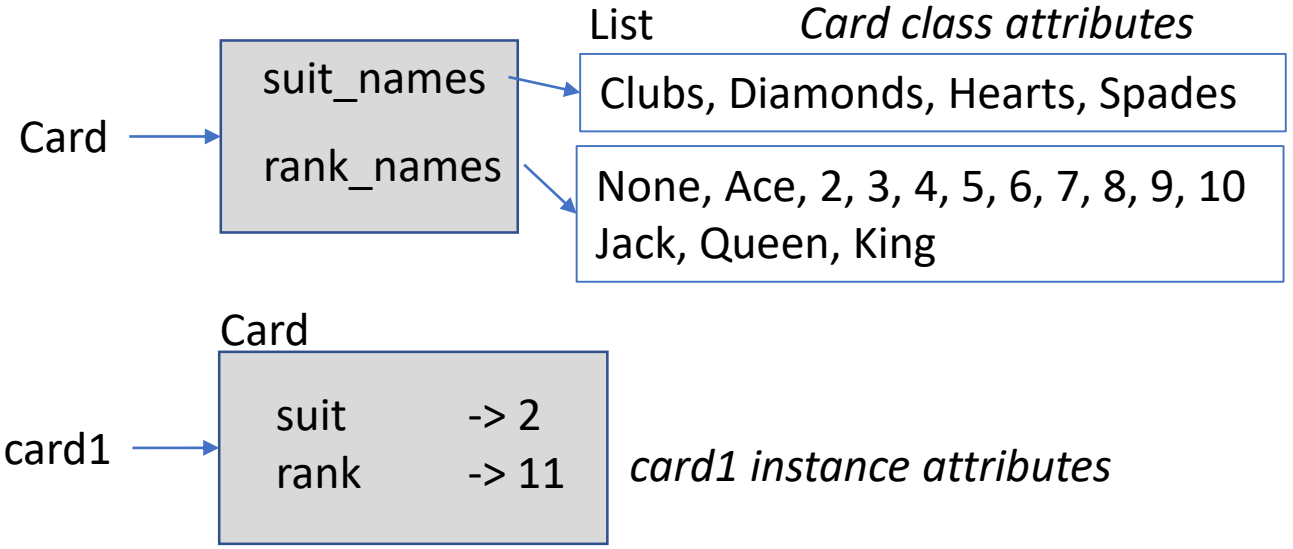
- Let's use object oriented techniques to model playing cards
  - suits: Spades, Diamonds, Diamonds, Clubs
  - rank: Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King
- How to implement suits?
  - direct: use Strings for each item
  - encode: map items to integers
    - Clubs -> 0
    - Diamonds -> 1
    - Hearts -> 2
    - Spades -> 3

```
class Card:
    """ represent playing cards """
    def __init__(self, suit=0, rank=2):
            self.suit = suit
            self.rank = rank
```



Class attributes

- to make things easier to read, we should include strings that map to the encoded values
  - **class** attributes: appear at the Class level
  - **instance** attributes: values associated with a particular instance of a class (or object)




```
class Card:
   """ represent deck of standard playing cards """

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

   # these are class attributes
   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.suit_names[ self.suit ],
              Card.rank_names[ self.rank ] )

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

Comparing cards

- Built-in types already have support for comparison operators <,>,==, etc
- We can supply operators for our Classes
  - the __lt__ method implements the "<" less than operator

```
class Card:
   """ represent deck of standard playing cards """

   def __init__(self, suit=0, rank=2):
      self.suit = suit
      self.rank = rank
   # these are class attributes
   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 ] )

   def __lt__(self, other):
      # check the suits
      if self.suit < other.suit: return True
      if self.suit > other.suit: return False
      # suits must be the same, compare ranks
      return self.rank < other.rank

 card1 = Card( 2, 11)   # Jack of Hearts
 card2 = Card( 2, 13)   # King of Hearts
 card1 < card2
True                                    # yup
```

Decks

- Now that we have Cards, the next step is to make a deck of 52 playing cards
- We'll make another Class, called Deck
  - iterate over the 4 possible suits
  - iterate over the 13 possible ranks

```
class Deck:
    """ implement a standard 52-card deck """
    def __init__(self):
        self.cards = []
        for suit in range(1,4):
            for rank in range(2, 14):
                card = Card( suit, rank )
                self.cards.append( card )
```

Printing the deck

- A __str__ method can be used by print() to print the deck
  - create a temporary list called "res"
  - go thru each card, collect it's printed form
  - clever trick: append a '\n' to each element of the "res" list using the built-in string method "join"

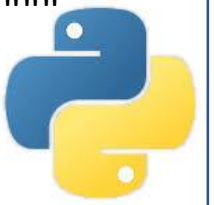
```
yum = ['spam', 'eggs', 'bacon' ]
print( yum )
['spam', 'eggs', 'bacon']

print ( '\n'.join( yum ) )
spam
eggs
bacon
```

```
class Deck:
    """ implement a standard 52-card deck """
    def __init__(self):
        self.cards = []
        for suit in range(1,4):
            for rank in range(2, 14):
                card = Card( suit, rank )
                self.cards.append( card )

    def __str__(self):
        res = []
        for card in self.cards:
            res.append( str(card) )

        # clever trick
        return '\n'.join( res )

d = Deck()              # make a deck
print( d ) # print it out in order…
2 of Diamonds
3 of Diamonds
4 of Diamonds
5 of Diamonds
…
```

Add, remove, shuffle, sort

- To deal cards, we need a method to retrieve a card from the deck
  - pop() removes bottom card from list
- To add a card, we can append to the cards list
  - insert the new card at the end of the list
- To shuffle, we can use the random module
  - randomizes elements in a list
- To sort
  - exercise for the reader…

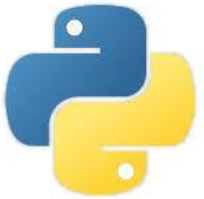
```
class Deck:
    """ implement a standard 52-card deck """

    # new stuff goes here

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

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

    import random
    def shuffle( self ):
        random.shuffle( self.card )

    def sort( self ):
        # exercise for the reader
        pass
```



Inheritance

- Inheritance is the ability to define a new class that is modified from an existing class
- Let's create a "hand" class
  - includes a set of cards like a Deck
  - has similar operations
  - has some differences, too
- Inheritance specified in Class def
  - Existing class is referenced in parenthesis

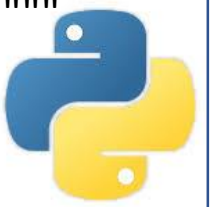
```
class Hand( Deck ):
    """ implements a hand of cards """
    def __init__( self, label=''):
        self.cards = []
        self.label = label

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

# use inherited methods from Deck
 deck = Deck()
 card = deck.pop_card()
 hand.add_card( card )
 print( hand )
King of Spades
```



Inheritance (2)

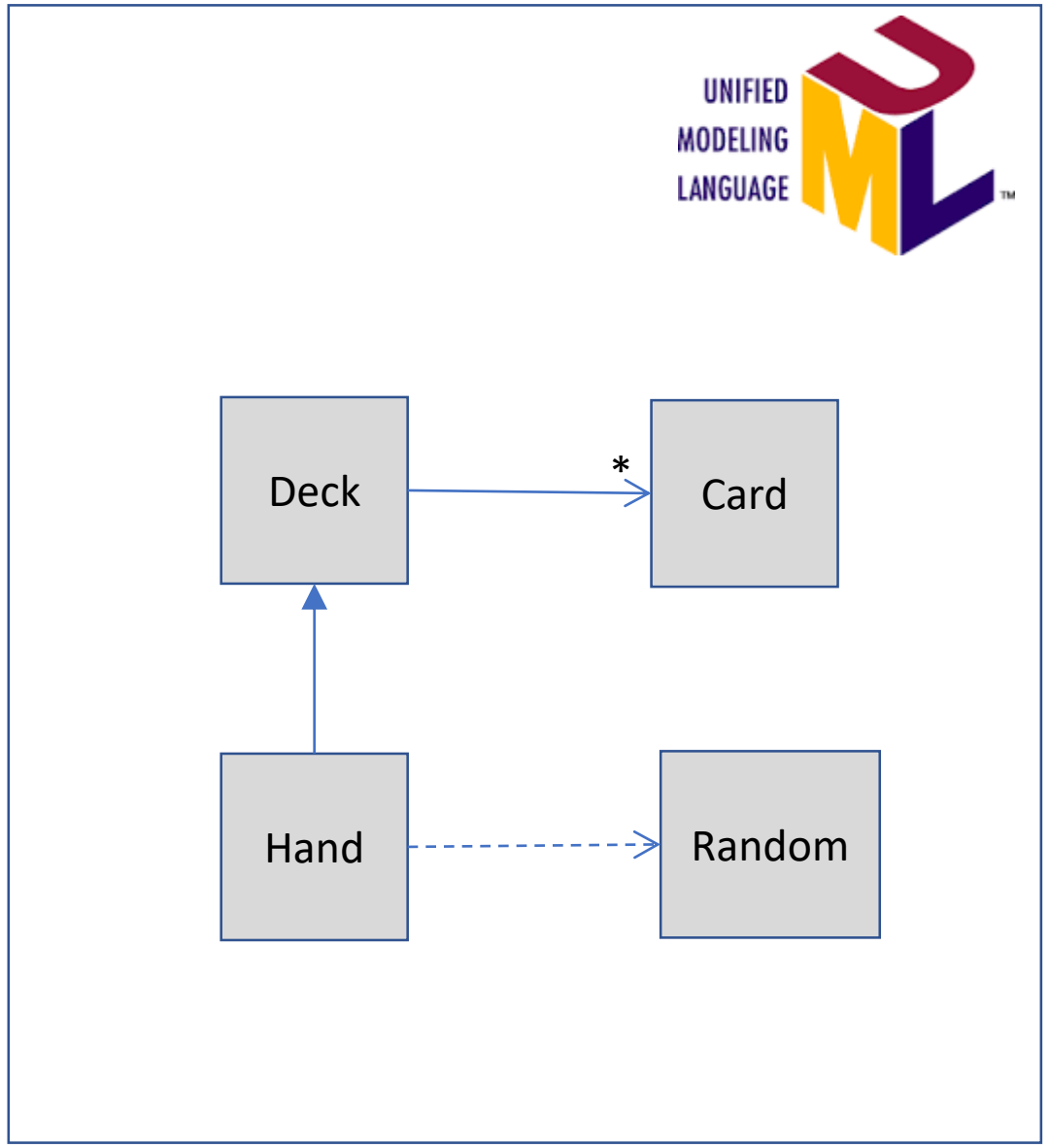
- A next step is to modify Deck so that it can allocate a specified number of cards to a hand

```
class Deck:
    """ implement a standard 52-card deck """

    # new stuff goes here

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



Class diagrams

- A class diagram is an abstract representation of a program
  - indicates classes and their relationships
- Types of relationships
  - HAS-A : one class contains references to another. A Rectangle has a Point (for it's center point)
  - IS-A: one class inherits from another class. A Hand is a type of Deck
  - Dependency: one class may depend on another class to do its work
- Unified Modeling Language (UML) used to depict class relationships
  - triangle arrow shows IS-A. Hand IS-A Deck
  - std arrow shows HAS-A. A Deck has Cards
  - the star "*" indicates multiplicity of 0 or more
  - Dashed arrows (if shown) indicate dependencies



Summary

- A Python class can "inherit" from a parent class
  - the class gets access to all the parent's methods and attributes
  - the class may specifiy its own methods and attributes
  - the class is said to be a "subclass" of the parent class