# Inheritance



## Dogs and police dogs

We defined a class of dogs all of which share:

- Attributes: gender, breed, height, weight, age.
- Operations (or behaviors): bark, walk, run, sit, jump, swim, ...

Now, we wish to create another class for police dogs.



## Dogs and police dogs

- Clearly, police dogs are dogs, so police dogs have all the attributes and behaviors of regular dogs.
- However, police dogs also have additional attributes (such as unit, department, etc.) and behaviors (sniff, attack, etc.)
- Can we somehow create a police-dog class as a special case of the class `Dog`?

## Inheritance

- *Inheritance* refers to the notion of one class inheriting the attributes and behaviors of another class.
- In Python, inheritance occurs when we define a class as a *subclass* of another.
- For example, a police-dog class may be defined as a subclass of the class `Dog`.



## A subclass of `Dog`

```
class Police_Dog(Dog):
```

`Police_Dog`'s constructor

`Police_Dog`'s additional methods

This `Police_Dog` class automatically inherits all the methods from the class `Dog`.



## A subclass of `Dog`

```
class Police_Dog(Dog):

  def __init__(self, b = 'German Shepherd',
    g = 'male', h = 20, w = 65, a = 5, u = 1,
    d = 'Hartford'):
    self.breed = b
    self.gender = g
    self.height = h
    self.weight = w
    self.age = a
    self.unit = 1
    self.department = 'Hartford'
```

`Police_Dog`'s additional methods

## A subclass of Dog

```
class Police_Dog(Dog):

  def __init__(self, b = 'German Shepherd',
    g = 'male', h = 20, w = 65, a = 5, u = 1,
    d = 'Hartford'):
      .
      .
      .
    self.unit = 1
    self.department = 'Hartford'

  def attack(self):
    print 'WWWWOOOOFFFF!!!!'
```



## Invoking methods

```
>>> eddie = Police_Dog()
>>> print eddie.weight
65
>>> eddie.gain_weight(5)
>>> print eddie.weight
70
>>> eddie.attack()
WWWWOOOOFFFF!!!!
>>> lucy = Dog()
>>> lucy.attack()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Dog' object has no attribute
'attack'
```



## Playing cards

## A card as an object

- The *suit* of a card is clubs = 0, diamonds = 1, hearts = 2 or spades = 3.
- The *rank* is its face value, where jack = 11, queen = 12, king = 13 and ace = 1.

```
class Card(object):
  """represents a standard playing card."""

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



## A card as an object

**Example.** What cards do these represent?

```
card1 = Card(1, 12)   # Queen of diamonds
card2 = Card(3, 1)    # Ace of spades
card3 = Card(4, 3)    # No such card
```



## Attributes

- *Class attributes* are defined inside a class. They are associated with the class itself.
- *Instance attributes* are associated with an instance (or object) of the class.

```
# Inside the class Card:

  suit_names = ['Clubs', 'Diamonds', 'Hearts',
    'Spades']
  rank_names = [None, 'Ace', '2', '3', '4', '5',
    '6', '7', '8', '9', '10', 'Jack', 'Queen',
    'King']
```

## Attributes

This __str__ method extracts the suit and rank *from the class attributes.*

```
# Inside the 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])
```



## Attributes

```
# Inside the class Card:

  suit_names = ['Clubs', 'Diamonds', 'Hearts',
    'Spades']
  rank_names = [None, 'Ace', '2', '3', '4', '5', '6',
    '7', '8', '9', '10', 'Jack', 'Queen', 'King']


>>> card1 = Card(1, 12)
>>> print card1
Queen of Diamonds
>>> card2 = Card(3, 1)
>>> print card2
Ace of Spades
```



## Operator overloading

We can override <, > and == by defining the __cmp__ method.

```
# Inside the class Card:
  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, so check ranks.
    if self.rank > other.rank: return 1
    if self.rank < other.rank: return -1
    # Ranks are the same, so it is a tie.
    return 0
```

## Operator overloading

We can override <, > and == by defining the __cmp__ method.

```
>>> card1 = Card(1, 11)
>>> card2 = Card(1, 6)
>>> card1 > card2
True
>>> card1 == card2
False
>>> card3 = Card(1, 11)
>>> card3 == card1
True
```



## A deck of cards

Consider a deck of cards as a list of cards.

```
class Deck(object):

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



## A deck of cards

This __str__ method first constructs a list of the string representation of the cards and then converts the list to a string.

```
#Inside the class Deck:

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

## A deck of cards

This `__str__` method first constructs a list of the string representation of the cards and then converts the list to a string.

```
>>> deck = Deck()
>>> print deck
Ace of Clubs
2 of Clubs
3 of Clubs
...
Queen of Spades
King of Spades
```



## Dealing, adding and shuffling

These methods use Python's built-in `list` methods and the module `random`.

```
#Inside the 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)
```



## `Hand` as a subclass of `Deck`

A hand in a game of cards is similar to a deck. It is a list of cards that has operations such as dealing, adding, shuffling, etc.

```
class Hand(Deck):
  """Represents a hand of playing cards."""

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

## Hand as a subclass of Deck

Both Deck and Hand have the same methods.

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