# LECTURE 18: INHERITANCE

Hung-Yu Wei

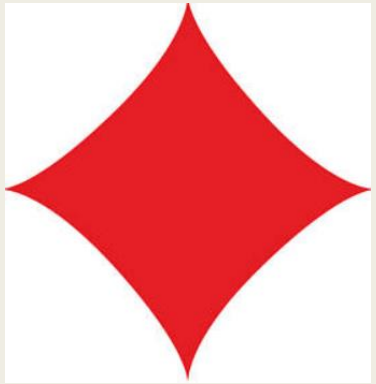
# Poker Card

- A 2 3 4 5 6 7 8 9 10 J Q K

$$\begin{aligned} \text{Jack} &\mapsto 11 \\ \text{Queen} &\mapsto 12 \\ \text{King} &\mapsto 13 \end{aligned}$$

- Spade, Hearts, Diamonds, Clubs






$$\begin{aligned} \text{Spades} &\mapsto 3 \\ \text{Hearts} &\mapsto 2 \\ \text{Diamonds} &\mapsto 1 \\ \text{Clubs} &\mapsto 0 \end{aligned}$$

# Class attributes

- Class attributes v.s. instance attributes
  - *Class attributes defined inside a class (associated with class)*

```
# 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])
```

# Comparing cards

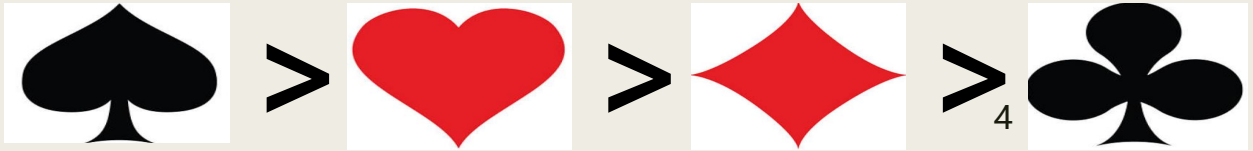
- **Less than**
  __lt__

```
# 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
```

- Exemplary rule
  - Compare suit first
  - Then compare rank (for the same suit)



# Implement __lt__ with tuple comparison

```
# inside class Card:

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

# Other comparison methods

- object.__lt__(self, other) <
- object.__le__(self, other) <=
- object.__eq__(self, other) ==
- object.__ne__(self, other) !=
- object.__gt__(self, other) >
- object.__ge__(self, other) >=

- https://docs.python.org/3/reference/datamodel.html

```
import random

class Card:
    """ suit: integer 0-3
        rank: integer 1-13
    """
    suit_names = ["Clubs", "Diamonds", "Hearts", "Spades"]
    rank_names = [None, "A", "2", "3", "4", "5", "6", "7",
                  "8", "9", "10", "J", "Q", "K"]

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

    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank],
                             Card.suit_names[self.suit])
    def __eq__(self, other):
        return self.suit == other.suit and self.rank == other.rank

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

# Deck

```
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)
```

List cards[]

List method: append()

# Printing the deck with __str__

```
#inside class Deck:

    def __str__(self):
        res = []

        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

List method: join()

```
>>> deck = Deck()
>>> print(deck)
```

# Review join a list → convert to string

- **join**
  - *Concatenate strings into a list*

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> s = delimiter.join(t)
>>> s
'pining for the fjords'
```

Join a list and convert to string

# Add, remove, shuffle

```
#inside class Deck:

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

List method: append()

```
#inside class Deck:

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

List method: pop()

Random module: shuffle()

```
# inside class Deck:

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

```
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)

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

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

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

    def remove_card(self, card):
        self.cards.remove(card)

    def pop_card(self, i=-1):
        return self.cards.pop(i)

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

    def sort(self):
        self.cards.sort()

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

# Inheritance

- Inheritance
  - *Parent class*
  - *Child class*
- Reuse (inherited from Parent class) with the same
  - *attributes*
  - *methods*
- Child class can override with a revised method
- Why inheritance?
  - *Inheritance can facilitate code reuse*
  - *Customize the behavior of parent classes without having to modify them*
  - *The inheritance structure reflects the natural structure of the problem*
    - E.g. NTU Student → NTUEE Student → NTUEE Freshman

# Example: Deck (Parent) → Hand(Child)

- Inheritance example

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

- Override with a new method
  - __init__

```
# inside class Hand:

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

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

# Example Continued

- Inherited from Parent class (Deck)
  - *Pop_card()*
  - *Add_card()*

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)
King of Spades
```

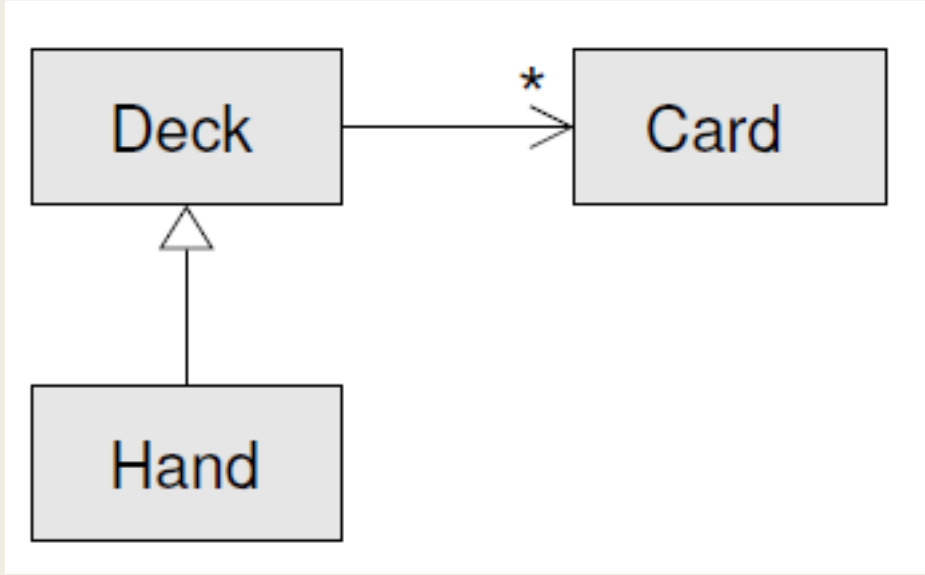
# Example Continued

```
#inside class Deck:

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

# Class diagram

- HAS-A
  - *Deck "has" multiple (*) cards*
- IS-A
  - *Hand "is a" modified class from Deck*



# Debugging

- Return the class that provides the definition of the method
  - *MRO: method resolution order*

```
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 'Card.Deck'>
```

```
class Hand(Deck):

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


def find_defining_class(obj, method_name):

    for ty in type(obj).mro():
        if method_name in ty.__dict__:
            return ty
    return None
```

```
deck = Deck()
deck.shuffle()


hand1 = Hand("Player #1's hand")
print(find_defining_class(hand1, 'shuffle'))

deck.move_cards(hand1, 13)
hand1.sort()
print(hand1)
```

# Tip: Liskov substitution principle

- Liskov substitution principle
  - *https://en.wikipedia.org/wiki/Liskov_substitution_principle*
- Subtypes must be substitutable for their base types
- When overriding a method, the **interface** of the new method should be the same as the old.
  - *same parameters, return the same type …*

# Data encapsulation

- Hide your data
  - *As attribute of your objects*
  - *Avoid using global variables*

```
class Markov:

    def __init__(self):
        self.suffix_map = {}
        self.prefix = ()
```

```
def process_word(self, word, order=2):
    if len(self.prefix) < order:
        self.prefix += (word,)
        return
```

# A development process

1. Start by writing functions that read and write global variables (when necessary).
2. Once you get the program working, look for associations between global variables and the functions that use them.
3. Encapsulate related variables as attributes of an object.
4. Transform the associated functions into methods of the new class.

# Reading

- Chapter 18 in textbook “Think Python”