CODE REUSE

- Classes meant to be reused

- Different ways to build old classes from new

  - Composition: class parts also classes

  - Inheritance: one class extends another
COMPOSE CLASSES

1 FACE

HAS

= 1 HEAD

2 EYES

1 NOSE

1 MOUTH

‘HAS A’ RELATIONSHIP
class Face(object):
    def __init__(self):
        self.head=Head()
        self.left_eye=Eye()
        self.right_eye=Eye()
        self.nose=Nose()
        self.mouth=Mouth()
CLASS COMPOSITION

HAS A LIST OF

'HAS A' RELATIONSHIP
suits = ('Clubs', 'Diamonds', 'Hearts', 'Spades')
ranks = ('Ace', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King')
_card_values = (11,) + tuple(range(2,11)) +(10,)*3
card_value_map = dict([(rnk+1,val) for rnk,val in enumerate(_card_values)])
import random
class Card(object):
    def __init__(self, suit=0, rank=0):
        assert 0<suit<5
        assert 0<rank<14
        self.suit = suit
        self.rank = rank
    def __str__(self):
        return '%s of %s' % (ranks[self.rank-1],suits[self.suit-1])
    def __cmp__(self, other):
        # check the ranks
        if self.rank > other.rank: return 1
        if self.rank < other.rank: return -1
        # ranks are the same... check suits
        if self.suit > other.suit: return 1
        if self.suit < other.suit: return -1
        # suits are the same... its a tie
        return 0
    def __int__(self):
        return card_value_map[self.rank]
class CardStack(object):
    def __init__(self, cards=()):
        self.cards = []
        for card in cards:
            self.cards.append(Card(suit=card.suit, rank=card.rank))
    def dealer_show(self, hide=1):
        val = ['X of X' for ind in range(hide)]
        val = val + [str(card) for card in self.cards[hide:]]
        return ', '.join(val)
    def shown_val(self, hide=1):
        return str(CardStack(self.cards[hide:]))
    def sort(self):
        self.cards.sort()
    def shuffle(self):
        for ind in range(7):
            random.shuffle(self.cards)
    def deal(self, num_cards=1):
        self.cards.reverse()
        cards_dealt = [self.cards.pop() for ind in range(num_cards)]
        self.cards.reverse()
        return CardStack(cards_dealt)
    def __str__(self):
        return ', '.join(str(card) for card in self.cards)
    def __int__(self):
        values = [int(card) for card in self.cards]
        while (11 in values) and (sum(values) > 21):
            values[values.index(11)] = 1
        else:
            return sum(values)
    def __add__(self, other):
        return CardStack(self.cards + other.cards)
    def __len__(self):
        return len(self.cards)
DECK PROBLEM

- Deck is like a card stack but with different init
- Composition a bit nasty for extending

```python
class BasicClass(object):
    def __init__(self):
        pass
    def do_one_thing(self):
        pass
    def do_another_thing(self):
        pass

class ExtendedClass(object):
    def __init__(self):
        self.core=BasicClass()
    def do_one_thing(self):
        self.core.do_one_thing()
    def do_another_thing(self):
        self.core.do_another_thing()
    def do_something_different(self):
        pass
```

We have to repeat each old function
INHERITANCE

`IS A`

class BasicClass(object):
    def __init__(self):
        pass
    def do_one_thing(self):
        pass
    def do_another_thing(self):
        pass

class ExtendedClass(BasicClass):
    def do_something_different(self):
        pass
COMPOSITION VS INHERITANCE

“QUIZ”

WHICH IS IT?

☐ A CAR, WHEELS

☐ A MEDICAL SPECIALIST, A DOCTOR

☐ AN ANIMAL, A FISH

☐ A SQUARE, A CORNER

☐ A SHAPE, A SQUARE
CLASS INHERITANCE

CardStack

IS A

Deck

PARENT CLASS
BASE CLASS
SUPER CLASS

SAME THING

CHILD CLASS
SUB CLASS

SAME THING

‘IS A’ RELATIONSHIP
class Deck(CardStack):
    def __init__(self, cards=None):
        if cards == None:
            cards = []
        for suit in range(1,5):
            for rank in range(1,14):
                cards.append(Card(suit=suit,
                                   rank=rank))
        super(Deck, self).__init__(cards=cards)
from user_input import get_user_bounded_int, get_user_choice
from cards import Deck, CardStack, Card
outcomes = ['win', 'lose', 'draw']
actions = ['hit', 'stay']

# Body of the program goes here

def main():
    game = Game()
    game.run()

if __name__ == '__main__':
    main()
CLASS USES OTHER CLASSES

USES

Game ➔ Round
class Game(object):
    def __init__(self):
        self.dealer = Dealer()
        self.player = Player()
        self.inital_balance = self.player.balance
    def run(self):
        self.welcome()
        while True:
            round = Round(self.dealer, self.player)
            round.play()
            if (self.player.balance < 1 or
                not self.ask_play_again()):
                break
        self.goodbye()
    def ask_play_again(self):
        # stuff here
    def welcome(self):
        # stuff here
    def goodbye(self):
        # stuff here
class Game(object):
    def __init__(self):
        # see previous slide
    def run(self):
       # see previous slide
    def ask_play_again(self):
        message = "Do you want to quit? [y/N]"
        choices = ['y','yes','n','no','']
        choice = get_user_choice(choices,message=message)
        if choice in ['n','no','']:
            return True
        else:
            return False
    def welcome(self):
        print "Welcome to the Game of 21"
    def goodbye(self):
        print "You have $"+str(self.player.balance)
        if self.player.balance > self.inital_balance:
            print "You have won",
            print "$"+str(self.player.balance-self.inital_balance),
            print "congrats!"
        elif self.player.balance < self.inital_balance:
            print "You have lost",
            print "$"+str(self.player.balance-self.inital_balance),
            print "gambling sucks!"
        else:
            print "You broke even. Hope it was fun."
        print "Thank you for playing. See you soon."
class Dealer(object):
    def __init__(self):
        self.deck = Deck()
        self.deck.shuffle()
        self.hand = CardStack([])
    def deal(self, other=None, num_cards=2):
        try:
            new_cards = self.deck.deal(num_cards=num_cards)
        except Exception, e:
            self.deck = Deck()
            new_cards = self.deck.deal(num_cards=num_cards)
        if other == None:
            self.hand = self.hand + new_cards
            return new_cards
        else:
            other.hand = other.hand + new_cards
            return new_cards
    def show(self):
        return self.hand.dealer_show()
    def get_action(self):
        if int(self.hand) <= 16:
            return 'hit'
        else:
            return 'stay'
class Player(object):
    def __init__(self, balance=500):
        self.hand = CardStack([])
        self.balance = balance
        self.bet = 0
    def get_bet(self):
        message = "Place your bet 
        self.bet = get_user_bounded_int(1, self.balance, message=message)
        return self.bet
    def show(self):
        return str(self.hand)
    def get_action(self):
        message = "What do you want to do?[h/S]: "
        choice_dict = {'hit': ['hit', 'h'], 'stay': ['stay', 's', '']}
        choices = []
        for inputs in choice_dict.values():
            for input in inputs:
                choices.append(input)
        choices = list(set(choices))
        choice = get_user_choice(choices, message=message)
        for key in choice_dict:
            if choice in choice_dict[key]:
                return key
    def update(self, outcome):
        if outcome == 'win':
            self.balance += self.bet
        elif outcome == 'lose':
            self.balance -= self.bet
        self.bet = 0
class Round(object):
    def __init__(self, dealer, player):
        self.dealer, self.player = dealer, player
        self.player.hand, self.dealer.hand = (CardStack([]), CardStack([]))
    def play(self):
        # This is where the "main" code for playing a round is
    def get_outcome(self):
        if int(self.player.hand) > 21:
            return 'lose'
        elif (int(self.dealer.hand) > 21 or
            int(self.player.hand) == 21 or
            int(self.dealer.hand) < int(self.player.hand)):
            return 'win'
        elif int(self.player.hand) < int(self.dealer.hand):
            return 'lose'
        else:
            return 'draw'
    def display(self):
        # code to display the player and dealer (hiding some dealer cards)
    def full_display(self):
        # code to display the player and dealer (showing all dealer cards)
class Round(object):
    def __init__(self, dealer, player):
        # initialize the round
    def play(self):
        # This is where the "main" code for playing a round is
    def get_outcome(self):
        # Update the player state
    def display(self):
        val = 'Dealer Hand: ' + self.dealer.show()
        val += ', Value shown: ' + str(self.dealer.hand.shown_val()) + '
        val += 'Player Hand: ' + self.player.show()
        val += ', Value: ' + str(int(self.player.hand))
        print val
    def full_display(self):
        val = 'Dealer Hand: ' + str(self.dealer.hand)
        val += ', Value shown: ' + str(int(self.dealer.hand)) + '
        val += 'Player Hand: ' + self.player.show()
        val += ', Value: ' + str(int(self.player.hand))
        print val
class Round(object):
...
    def play(self):
        self.player.get_bet()
        self.dealer.deal()
        self.dealer.deal(other=self.player)
        dealer_action, player_action = 'start', 'start'
        no_dealer_turn = False
        while not (player_action == 'stay'):
            self.display()
            if int(self.player.hand) == 21:
                no_dealer_turn = True
                print "Player 21"
                break
            player_action = self.player.get_action()
            if player_action == 'hit':
                card = self.dealer.deal(other=self.player, num_cards = 1)
                print 'You get', card
                if int(self.player.hand) > 21:
                    no_dealer_turn = True
                    print "Player Busts"
                    break
        while not (dealer_action == 'stay') and not no_dealer_turn:
            dealer_action = self.dealer.get_action()
            if dealer_action == 'hit':
                card = self.dealer.deal(num_cards = 1)
                print 'Dealer gets', card
                if int(self.dealer.hand) > 21:
                    print "Dealer Busts"
                    break
        self.full_display()
        outcome = self.get_outcome()
        print 'You', outcome
        self.player.update(outcome)
        print 'Current balance:', self.player.balance
POLYMORPHISM

IN INHERITANCE PYTHON FIGURES OUT WHICH CLASS'S FUNCTION TO CALL

DO THE RIGHT THING FOR THAT TYPE
PILLARS OF OOP

- Abstraction
- Data + Action
- Encapsulation
- Information Hiding
- Inheritance
- Reuse by Extending
- Polymorphism
- Child or Parent Method Determined
THANKS