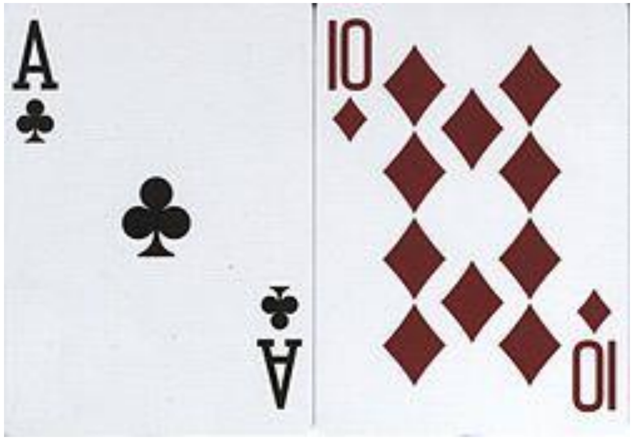
Blackjack

OOA, OOD, AND OOP IN PYTHON



AGENDA

- This is a long term project that will keep us busy until the end of the semester (this is also the last project for this course)
- The main goal is to put you in a real life scenario in which your mission is not completely clear (as it is in many “industrial situations”) and you have to work to find your way to a clear working project
- We want to develop a software for simulating Blackjack games in order to test the quality of several playing strategies (before using them in a Casino)
- Our software should enable us to simulate thousand (or even millions) of real Blackjack games in a very short time in order to check if a player strategy is any good?
- Start reading this presentation, think about the problem, and please come up with some ideas for next class

PORJECT GOALS

- After completing this project, the student should gain a basic experience with the following major topics
- **Software Modelling**
  - Learning how to play blackjack and then writing the whole game in software is a complex process called “Modeling”
- **Object Oriented Analysis and Design**
  - Before software modelling, we need to analyse and design our classes, objects, attributes, and methods
- **Common Object Oriented Programming Techniques**
- **Software Simulation Skills**
  - After implementing our model in a concrete programming language, we will be able to rapidly simulate thousands of “virtual” games and experiment with player strategies, statistical date, and more
  - This can save a lot of time and resources compared to the effort needed for doing such research in real Casino games

Game Story

- Description based on http://en.wikipedia.org/wiki/Blackjack
- Before software modelling, a developer is required to understand the story and rules of the domain he is trying to model in software
- Blackjack (also called "21" or "twenty-one") is the most popular Casino cards game
- There are more than 100 variations of Blackjack in different Casino houses
- We will use the simple double exposure variation in order to make the software modelling readable and clear example for the **OOA**, **OOD**, and **OOP** processes
  (to make it simpler, we will not use “splits” and “double bets”)

Game Rules

- We will use only one **deck** of 52 cards:
  - **13 ranks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']**
  - **4 suits = ['Hearts', 'Clubs', 'Spades', 'Diamonds']**
  - **Total 52 cards**
- **Dealer**
  - The Casino representative
  - Deals the cards
- **Players:** 2-6 (including the dealer)

- **Double Exposure Variation**
  - To simplify, we will use the game variation in which all the dealer's cards are exposed, and we will not use "splits" and "double bets"
  - (In many Casinos, dealer's first card is hidden)

Game Rules

- Blackjack is a comparing card game between each player and the dealer
- It means that players compete <u>against the dealer</u> but not against each other
- The object of the game is to "beat the dealer“, which can be done in a number of ways:
  - Get **21** points on your <u>first two cards</u> (called a **blackjack**), without a dealer blackjack
  - Reach a final score higher than the dealer without exceeding 21
  - Or let the dealer draw additional cards until his hand exceeds 21
  - All other cards are counted as the numeric value shown on the card

Card Values

- Each card in ['2', '3', '4', '5', '6', '7', '8', '9', '10'] has a **value** equal to its number
- All the cards ['J', 'Q', 'K'] have **value** of 10
- The Ace card 'A' has two possible values: 1 or 11 (according to player’s choice)

Game Open: Initial Two-card Hand

- At the start of the game (Open), each player is dealt an initial two-card hand by the dealer
- The dealer is the last player to get cards
- A player and the dealer can count his or her own Ace as 1 point or 11 points
- All other cards are counted as the numeric value shown on the card
- All dealer's and players cards are face-up (visible to all)
- This variation of blackjack is called **Double Exposure Variation**

Hit and Stand

- After receiving their initial two cards, players have the option of getting a **“hit”**, which means taking an additional card or a **“stand”** (no more cards)
- A player may **‘hit’** the dealer as many times as it wants (as long as he’s willing to take the risk of “busting out”)
- Scoring higher than **21** (called “busting” or “going bust”) results in a <u>loss of the game</u>
- As soon as the player is satisfied with his score he declares a **‘stand’** which means he stops getting cards from the dealer

Winning

- A player may win by having any final score equal to or less than 21 if the <u>dealer busts</u>
- In a given game, the player or the dealer wins by having a score of 21 or by having the highest score that is less than 21
- If the player and dealer do not bust and have equal scores, then no one win or loses (they both keep their bet). This is called a “Tie” or a “Push”.
  (but in most double exposure variations, the dealer wins in such case)

Soft and Hard Hands

- If a player holds an Ace valued as 11, the hand is called **"soft"**, meaning that the player cannot go bust by taking an additional card
    - 11 plus the value of any other card can always be less than or equal to 21
    - Otherwise, the hand is **"hard"**
- The dealer **must** take hits unless his hand value is 17 or more (even as a soft hand!) – in such case he **must** stand!
- For example, if the dealer has ['A', '6'] he **cannot** take more cards ! He must declare ‘stand’ !
    - Must stop and wait for other players to stand or bust

Win/Lose Rules Summary 1

- Players win if they do not bust and have a total that is higher than the dealer
- The dealer loses if he busts or has a lesser hand than the player who has not busted
- If the hand value exceeds 21 points, it **busts**, and all bets on it are <u>immediatel</u>y forfeit, cards removed (the player exits the game, but the game itself continues with the other players)
- If the player and dealer have the <u>same point total</u>, this is called a **"push"**, and the player does not win or lose money on that hand (in some versions, the dealer wins a “push”)

Going Bust (important note)

- If a player’s hand value exceeds 21 points, it **busts**
- In such case, the dealer immediately removes the player bet and cards, before proceeding to the next player!
- Since the dealer is the last one to draw cards, it may happen that after dealing with all players, he is also bust!
- Nevertheless, he still keeps the bets of all players that went bust before him
- This is were the Casino makes its profit …

Win/Lose Rules Summary 2

- The dealer never stands! Must always take a card! (until reaching 17 and up, in which he must stop)
- If the dealer busts, all remaining player hands that did not bust) win and the game is over
- If the dealer does not bust, each remaining bet wins if its hand is higher than the dealer's, and loses if it is lower
- In the case of a tied score (a "push“) bets are returned to their players with no loss or gain

When a Game is Over?

- After the two-cards round, the dealer has blackjack
  - All players with less than 21 lose their bet
  - Players with 21 keep their bet
  - Game is over
- When the dealer busts (hand value exceeded 21)
  - All non-busted players win their bet
  - All cards returned to deck, game is over
- Dealer hand is 17-21 and each player either committed a “stand” or busted
  - Each player win/lose/draws according to his hand value compared to the dealer hand

Bets

- To make it simple, all bets are on 1 chip
- So we need not model bets in our software model
- Each player will have a ‘budget’ attribute (in chips)
- When he wins, we add 1 to budget
- When he loses, subtract 1
- When he draws equal,
  no change to budget

Bets settled

Dealing Order

- Players have a natural order, and are numbered from 1
- The dealer starts with player 1 to the last player, and he is the last one to get cards
- After the initial two cards, the dealer deals with each player (according to order) until he stands (or busts)
- That means, each player gets all the cards he can, until it either stands or busts
- The dealer then proceeds to deal with the next player

Bets settled

Dealer's Advantage

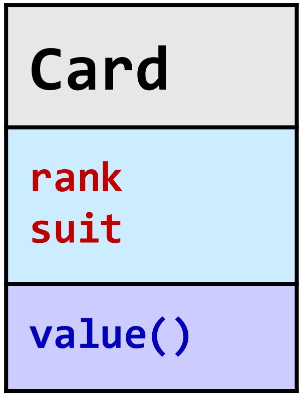
- The reason the dealer has an advantage over the players is because the **dealers turn is always after the players**
- So if the dealer busts and the player busts, the dealer still takes the busted players money (since the player always busts first!)
- The dealer also has the advantage by **always having enough money to stake against the players (Casino budget is usually much higher than player budget)**
- In some version of the double exposure variation, the dealer also wins in case of a tie ("push"). We may explore this version later if needed.

Good Links on BlackJack

- http://www.wikihow.com/Play-Blackjack
- http://en.wikipedia.org/wiki/Blackjack
- http://www.pagat.com/banking/blackjack.html
- http://www.maxgames.com/play/black-jack-card-game.html
- http://www.blackjack.org/rules/
- https://www.youtube.com/watch?v=Up9Eq2fv_-g
- Double Exposure Variation
- https://www.youtube.com/watch?v=47qguu7ODqo
- https://www.youtube.com/watch?v=QzzMi8RAnls
- Blackjack Online : MIT-Blackjack-Team Movie

OOD

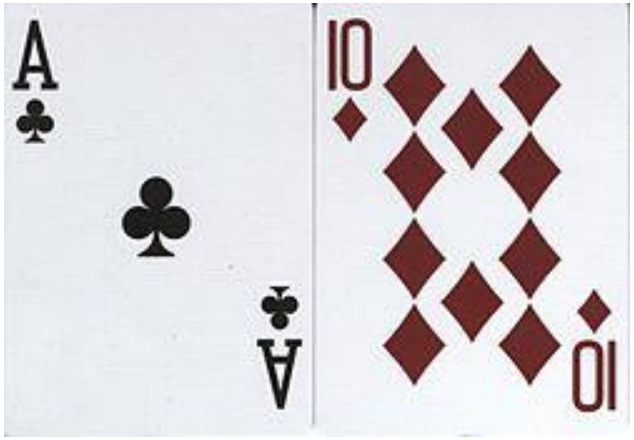
OBJECT ORIENTED DESIGN

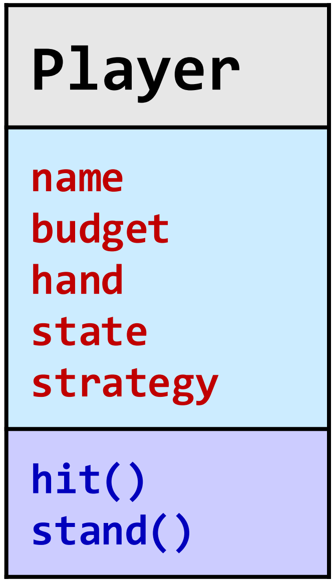


Card

rank
suit

value()



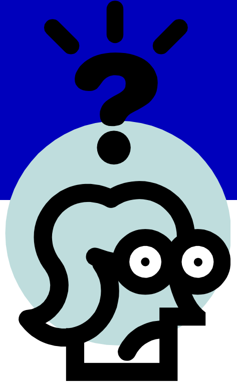


Player

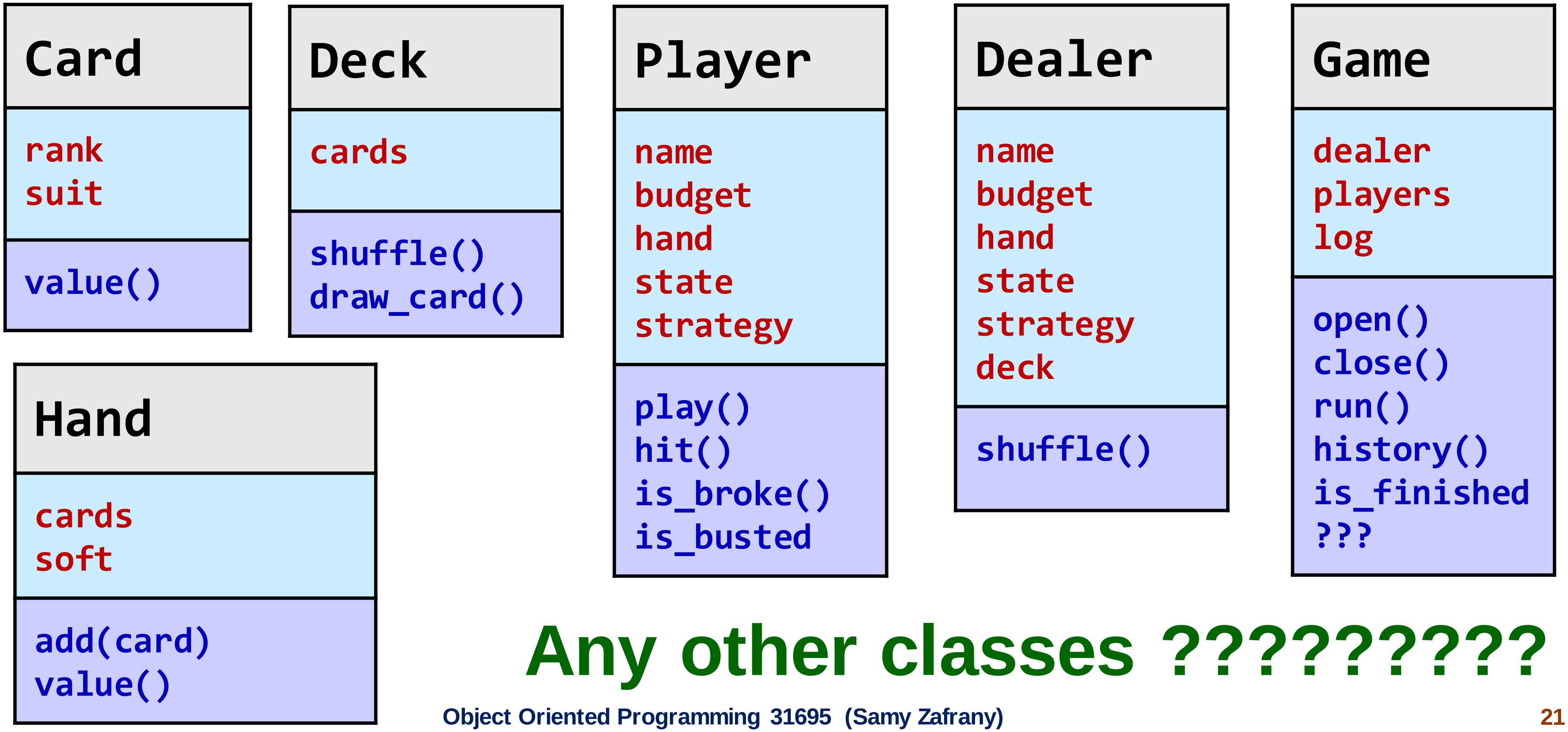
name
budget
hand
state
strategy

hit()
stand()

Classes ?



- Here are some ideas for classes we want to consider – just a suggestion! Nothing final yet …
- Do some thinking on what classes you think we should have? And what sort of attributes and methods should they have?



Card

rank
suit

value()

Deck

cards

shuffle()
draw_card()

Player

name
budget
hand
state
strategy

play()
hit()
is_broke()
is_busted

Dealer

name
budget
hand
state
strategy
deck

shuffle()

Game

dealer
players
log

open()
close()
run()
history()
is_finished
???

Hand

cards
soft

add(card)
value()

Any other classes ?????????

AGENDA

- **OOD** brainstorming in class (but please start thinking about **OOD** before the class)
- We need to decide what are our classes? How do they relate to each other?
- **OOP**
  - After OOD we need to implement our specification in some programming language
  - Naturally we will start with Python
  - Your last assignment in this course is to convert our Python implementation to another language such as Java, C++, or C# - we will discuss this in class

SIMULATION

- Our main goal in this project is to test several player strategies by simulating a few thousand games with our software environment
- A strategy is any function f(hand1, hand2) which accepts the player's and dealer's hands and returns the move to make next (usually 'hit' or 'stand')
- The dealer's strategy is very simple:

```
If hand_value < 17:
    'hit'
else:
    'stand'
```

- The players strategy is usually much more complicated and can involve many different factors

Simple Player Strategy

```
def strategy1(player_hand, dealer_hand):
    player_value = player_hand.value()
    dealer_value = dealer_hand.value()
    if player_value < dealer_value:
        return 'hit'
    if player.hand.soft:
        if player_value < 17:
            return 'hit'
        elif player_value > 18:
            return 'stand'
        else:
            if random.choice([0,1]):
                return 'hit'
            else:
                return 'stand'
    else:
        if player_value < 11:
            return 'hit'
        elif player_value > 17:
            return 'stand'
        else:
            return 'hit'
```

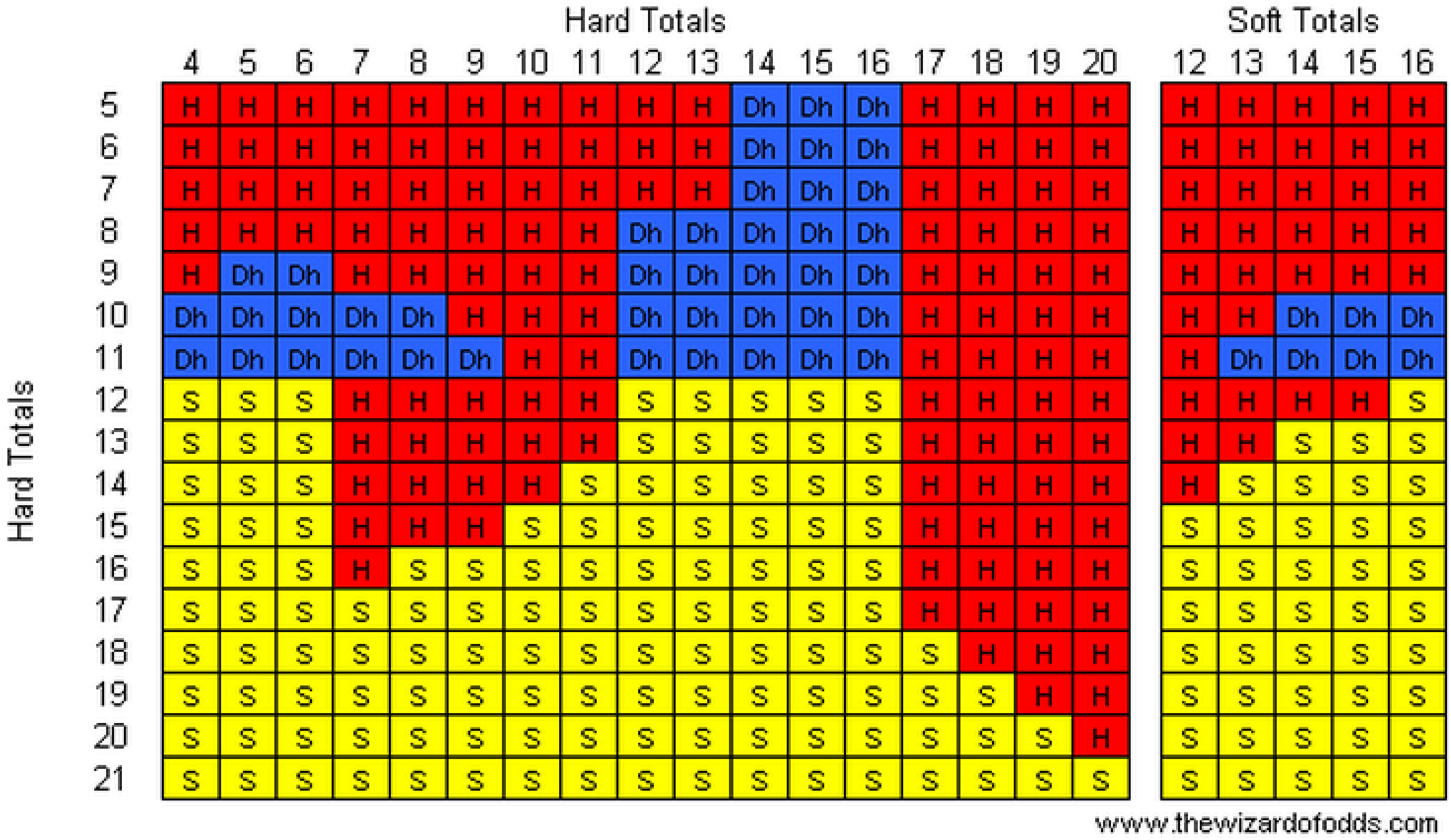
Tabular Strategies

- Professional strategies are sometimes too hard to express in simple functions like in the previous slide
- In most cases we need 4 different tables to describe the strategy
- These tables can be expressed well by a Python dictionary which we define inside a “strategy file” (look next)
- See next slides for an expert example (Michael Shackleford, http://wizardofodds.com/site/about)

Player Strategy Example (tables 1,2)

Dealer

Player

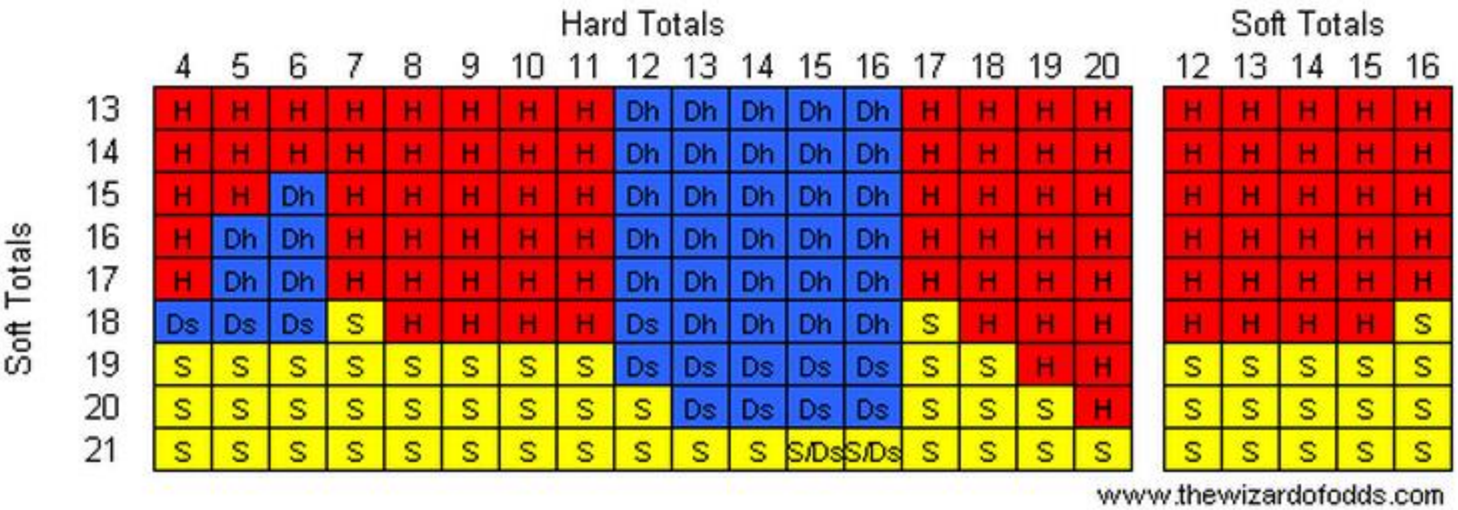


Hard Totals	Hard Totals																	Soft Totals				
	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	12	13	14	15	16
5	H	H	H	H	H	H	H	H	H	H	Dh	Dh	Dh	H	H	H	H	H	H	H	H	H
6	H	H	H	H	H	H	H	H	H	H	Dh	Dh	Dh	H	H	H	H	H	H	H	H	H
7	H	H	H	H	H	H	H	H	H	H	Dh	Dh	Dh	H	H	H	H	H	H	H	H	H
8	H	H	H	H	H	H	H	H	Dh	Dh	Dh	Dh	Dh	H	H	H	H	H	H	H	H	H
9	H	Dh	Dh	H	H	H	H	H	Dh	Dh	Dh	Dh	Dh	H	H	H	H	H	H	H	H	H
10	Dh	Dh	Dh	Dh	Dh	H	H	H	Dh	Dh	Dh	Dh	Dh	H	H	H	H	H	H	Dh	Dh	Dh
11	Dh	Dh	Dh	Dh	Dh	Dh	H	H	Dh	Dh	Dh	Dh	Dh	H	H	H	H	H	Dh	Dh	Dh	Dh
12	S	S	S	H	H	H	H	H	S	S	S	S	S	H	H	H	H	H	H	H	H	S
13	S	S	S	H	H	H	H	H	S	S	S	S	S	H	H	H	H	H	H	S	S	S
14	S	S	S	H	H	H	H	S	S	S	S	S	S	H	H	H	H	H	S	S	S	S
15	S	S	S	H	H	H	S	S	S	S	S	S	S	H	H	H	H	S	S	S	S	S
16	S	S	S	H	S	S	S	S	S	S	S	S	S	H	H	H	H	S	S	S	S	S
17	S	S	S	S	S	S	S	S	S	S	S	S	S	H	H	H	H	S	S	S	S	S
18	S	S	S	S	S	S	S	S	S	S	S	S	S	S	H	H	H	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	H	H	S	S	S	S	S
20	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	H	S	S	S	S	S
21	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S

www.thewizardofodds.com

As we do not use double bets, ignore **Dh**
And replace it with **H**

Player Strategy Example (tables 3,4)



		Hard Totals																	Soft Totals				
		4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	12	13	14	15	16
Soft Totals	13	H	H	H	H	H	H	H	H	Dh	Dh	Dh	Dh	Dh	H	H	H	H	H	H	H	H	H
	14	H	H	H	H	H	H	H	H	Dh	Dh	Dh	Dh	Dh	H	H	H	H	H	H	H	H	H
	15	H	H	Dh	H	H	H	H	H	Dh	Dh	Dh	Dh	Dh	H	H	H	H	H	H	H	H	H
	16	H	Dh	Dh	H	H	H	H	H	Dh	Dh	Dh	Dh	Dh	H	H	H	H	H	H	H	H	H
	17	H	Dh	Dh	H	H	H	H	H	Dh	Dh	Dh	Dh	Dh	H	H	H	H	H	H	H	H	H
	18	Ds	Ds	Ds	S	H	H	H	H	Ds	Dh	Dh	Dh	Dh	S	H	H	H	H	H	H	H	S
	19	S	S	S	S	S	S	S	S	Ds	Ds	Ds	Ds	Ds	S	S	H	H	S	S	S	S	S
	20	S	S	S	S	S	S	S	S	S	Ds	Ds	Ds	Ds	S	S	S	H	S	S	S	S	S
	21	S	S	S	S	S	S	S	S	S	S	S	S/Ds	S/Ds	S	S	S	S	S	S	S	S	S

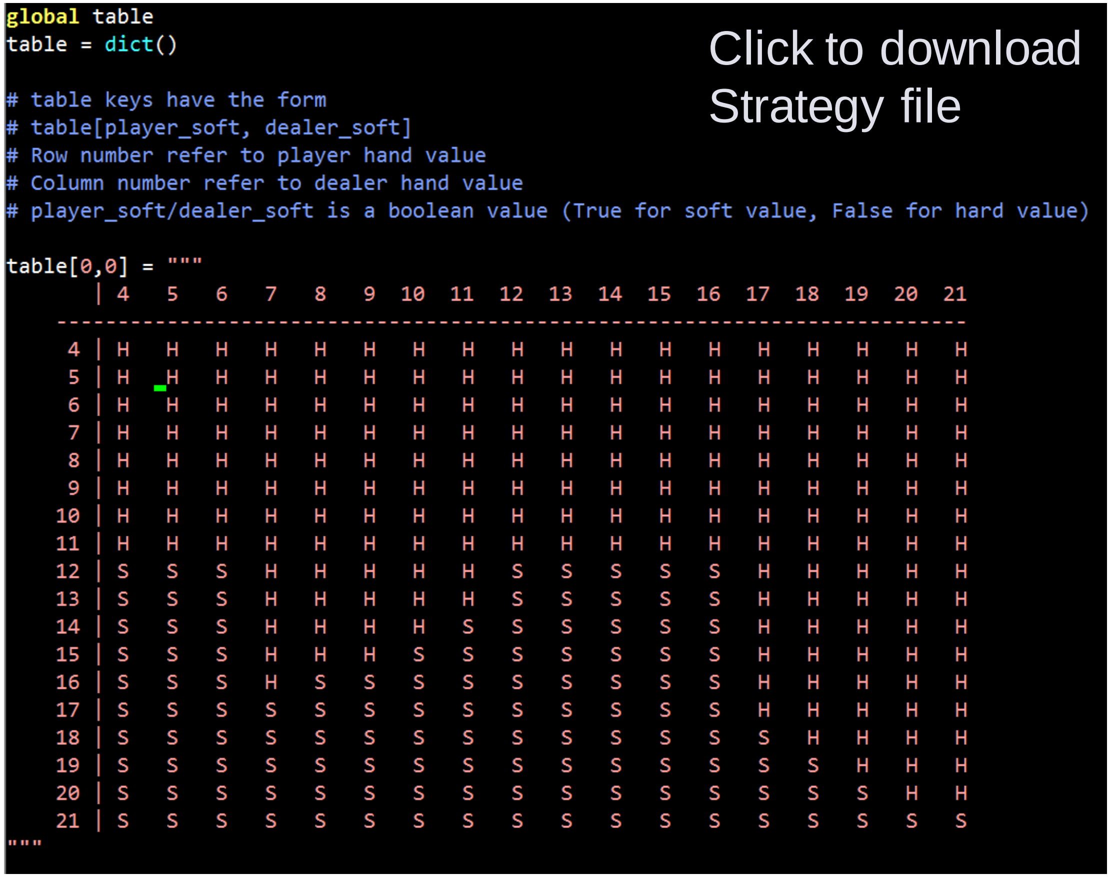
www.thewizardofodds.com

S = Stand
H = Hit
Dh = Double if allowed, otherwise hit (**H** in our case – as we do not have double bets)
Ds = Double if allowed, otherwise stand (**S** in our case – we do not have splits)
S/Ds = Stand on first two cards, double if after splitting and allowed, otherwise stand

As we do not use double bets, ignore **Dh a**nd replace it with **H,**
Ignore **Ds** and replace it with **S**

Strategy File

- A strategy file defines these 4 tables by Python dictionary and enables us to define a strategy function based on these 4 tables
- Here is a strategy file that defines Michael Shackleford strategy



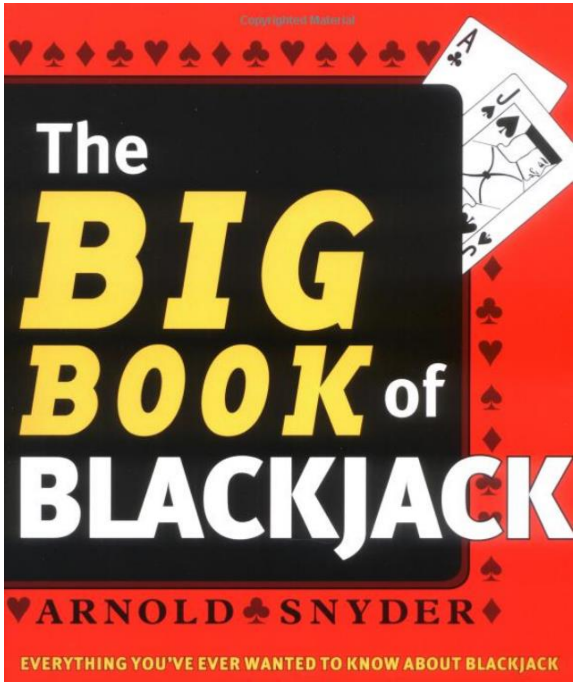
```
global table
table = dict()

# table keys have the form
# table[player_soft, dealer_soft]
# Row number refer to player hand value
# Column number refer to dealer hand value
# player_soft/dealer_soft is a boolean value (True for soft value, False for hard value)

table[0,0] = """
      | 4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21
    ---------------------------------------------------------------------------
    4 | H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H
    5 | H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H
    6 | H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H
    7 | H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H
    8 | H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H
    9 | H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H
   10 | H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H
   11 | H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H   H
   12 | S   S   S   H   H   H   H   H   S   S   S   S   S   H   H   H   H   H
   13 | S   S   S   H   H   H   H   H   S   S   S   S   S   H   H   H   H   H
   14 | S   S   S   H   H   H   H   S   S   S   S   S   S   H   H   H   H   H
   15 | S   S   S   H   H   H   S   S   S   S   S   S   S   H   H   H   H   H
   16 | S   S   S   H   S   S   S   S   S   S   S   S   S   H   H   H   H   H
   17 | S   S   S   S   S   S   S   S   S   S   S   S   S   H   H   H   H   H
   18 | S   S   S   S   S   S   S   S   S   S   S   S   S   S   H   H   H   H
   19 | S   S   S   S   S   S   S   S   S   S   S   S   S   S   S   H   H   H
   20 | S   S   S   S   S   S   S   S   S   S   S   S   S   S   S   S   H   H
   21 | S   S   S   S   S   S   S   S   S   S   S   S   S   S   S   S   S   S
"""
```

Click to download Strategy file

Click to download The strategy file reader

"The Simplified Basic Strategy"

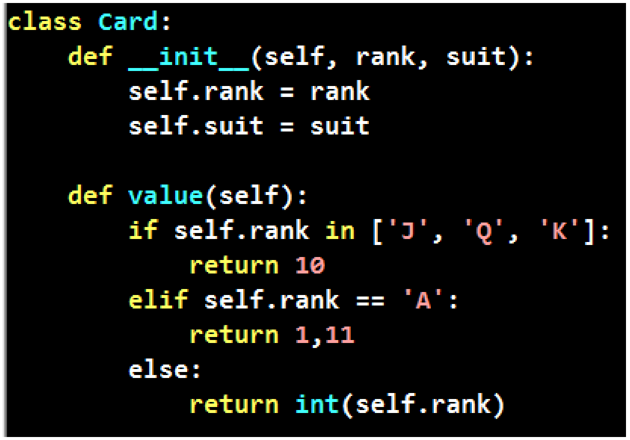
```python
def strategy3(player_hand, dealer_hand):
    pvalue = player_hand.value()
    dvalue = dealer_hand.value()
    psoft = player_hand.soft
    dsoft = dealer_hand.soft
    phard = not player_hand.soft
    dhard = not dealer_hand.soft

    if 17 <= dvalue <= 21:  # Dealer pat hand
        if pvalue < dvalue:
            return 'hit'
    elif 7 <= dvalue <= 11:
        if pvalue <= dvalue or 12 <= pvalue <= 15:
            return 'hit'
    elif dvalue < 7:
        if pvalue < 12:
            return 'hit'
        elif psoft and pvalue < 16:
            return 'hit'

    if dhard and 12 <= dvalue <= 16:  # Dealer "stiff" hand
        if pvalue < dvalue:
            return 'hit'
        elif psoft and pvalue <= 16:
            return 'hit'

    if dsoft and 12 <= dvalue <= 16:
        if pvalue <= 12:
            return 'hit'
        elif psoft and pvalue <= 18:
            return 'hit'

    return 'stand'
```



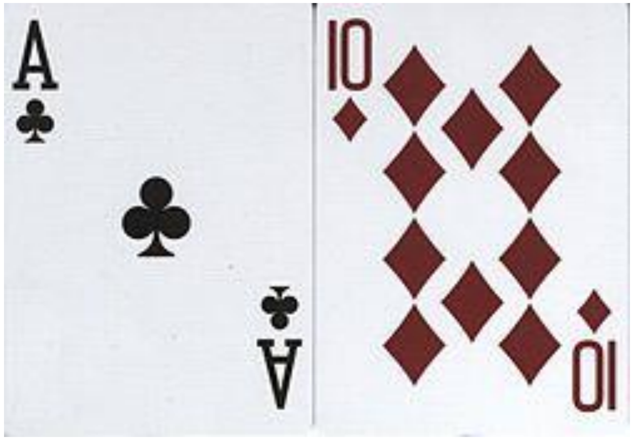
Strategy 3:
"The Simplified Basic Strategy"
Copied from the "Big Book of Blackjack" By Arnold Snyder
http://www.amazon.com/Big-Book-Blackjack-Arnold-Snyder/dp/1580421555

OOP

OBJECT ORIENTED PROGRAMMING IN PYTHON



```
class Card:
    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit

    def value(self):
        if self.rank in ['J', 'Q', 'K']:
            return 10
        elif self.rank == 'A':
            return 1,11
        else:
            return int(self.rank)
```



AGENDA

- Remember our long term goal: create a convenient software environment for simulating thousands of Blackjack games in order to test player strategies (so we know how good they are before we use them in a Casino …)
- Please start by designing a few more classes toward this goal
- We will complete the work in the course laboratory sessions (but you must be prepared with a few classes of yours!
  So get started …)
- To get you started, here are client tests and two suggestion for classes that give you a taste for what we are trying to do
- Remember that writing tests (many of them) before you write classes can actually help you make better design choices!

The Card Class (for a start ...)

```
ranks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']
suits = ['Hearts', 'Clubs', 'Spades', 'Diamonds']

class Card:
    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit

    def value(self):
        if self.rank in ['J', 'Q', 'K']:
            return 10
        elif self.rank == 'A':
            return 1,11
        else:
            return int(self.rank)

    def __str__(self):
        return self.rank + '-' + self.suit
```

The Deck Class (for a start ...)

```
class Deck:
    def __init__(self):
        self.cards = []
        for rank in ranks:
            for suit in suits:
                c = Card(rank, suit)
                self.cards.append(c)

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

    def draw_card(self):
        if not self.cards:
            raise Exception("No more cards: empty deck!")
        card = self.cards.pop()
        return card

    def __str__(self):
        cards = []
        for c in self.cards:
            cards.append(str(c))
        return str(cards)
```

Test 1: simple cards

```
def test1():
    card1 = Card('9', 'Spades')
    card2 = Card('Q', 'Hearts')
    card3 = Card('9', 'Hearts')
    card4 = Card('K', 'Diamonds')
    print card1, card2, card3, card4
```

Test 2: Deck, shuffle

```
def test2():
    deck = Deck()
    print '---------  Before Shuffle  --------'
    print deck
    deck.shuffle()
    print '---------  After Shuffle  --------'
    print deck
```

Test 3: Hand

```
def test3():
    deck = Deck()
    deck.shuffle()
    c1 = deck.draw_card()
    c2 = deck.draw_card()
    c3 = deck.draw_card()
    h = Hand([c1, c2, c3])
    c4 = deck.draw_card()
    h.add(c4)
    print h
    print h.value()
```

Test 4: Making a Random Hand

```
def random_hand():
    "Random hand of 2 to 5 cards"
    deck = Deck()
    deck.shuffle()
    n = random.randint(2,5)
    cards = []
    for i in range(n):
        c = deck.draw_card()
        cards.append(c)
    return Hand(cards)
```

Test 5: Running One Game

```
def test5():
    dealer = Dealer('Eli', 10000)
    a = Player('Alice', 100, strategy1)
    b = Player('Bob', 200, strategy2)
    c = Player('Clod', 100, strategy3)
    d = Player('Dian', 250, strategy4)

    print "Dealer:", dealer.name
    print "Players:", a.name, b.name, c.name, d.name

    players = [a, b, c, d]
    g = Game(dealer, players)
    g.run()
    print g.log   # should print all game history
```

Test 6: Simulating 3000 Games !!!

Which is better? strategy1 or strategy2 ???

```
def test6():
    strategy2 = read_strategy_file('strategy2.py')
    dealer = Dealer('Eli', 10000)
    a = Player('Alice', 500, strategy2)
    b = Player('Bob', 500, strategy1)
    c = Player('Clod', 500, strategy1)

    for i in range(3000):
        g = Game(dealer, [a, b, c])
        g.run()

    print a.name, a.budget    # which budget is higher?
    print b.name, b.budget
    print c.name, c.budget
```

Test 7: Simulation Graphs !

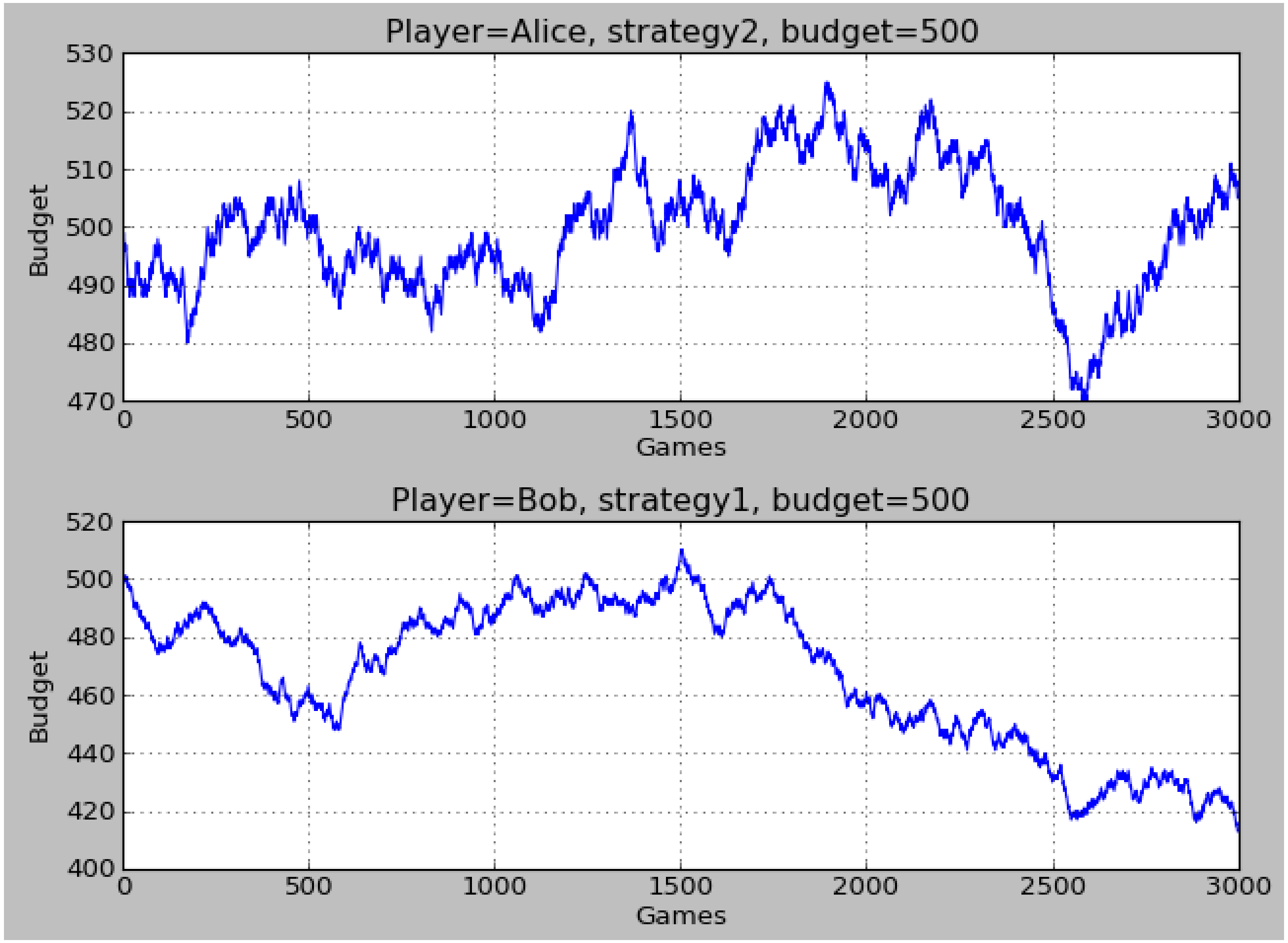
Which is better? strategy1 or strategy2 ???

```
def test7():
    import matplotlib.pyplot as plt
    strategy2 = read_strategy_file('strategy2.py')
    dealer = Dealer('Eli', 10000)
    a = Player('Alice', 500, strategy2)
    b = Player('Bob', 500, strategy1)
    c = Player('Clod', 500, strategy1)

    a_init_budget = a.budget
    b_init_budget = b.budget

    a_budgets = []
    b_budgets = []
    for i in range(3000):
        g = Game(dealer, [a, b, c])
        g.run()
        a_budgets.append(a.budget)
        b_budgets.append(b.budget)
```

```
    plt.subplot(211)
    plt.plot(range(3000), a_budgets)
    plt.grid(True)
    plt.title("Player=%s, %s, budget=%d" %
                   (a.name, "strategy2", a_init_budget))
    plt.xlabel('Games')
    plt.ylabel('Budget')

    plt.subplot(212)
    plt.plot(range(3000), b_budgets)
    plt.grid(True)
    plt.title("Player=%s, %s, budget=%d" %
                   (b.name, "strategy1", b_init_budget))
    plt.xlabel('Games')
    plt.ylabel('Budget')

    plt.tight_layout()
    plt.show()
```

Simulation of 3000 Games (1)

Alice is using strategy2
Bob is using strategy1

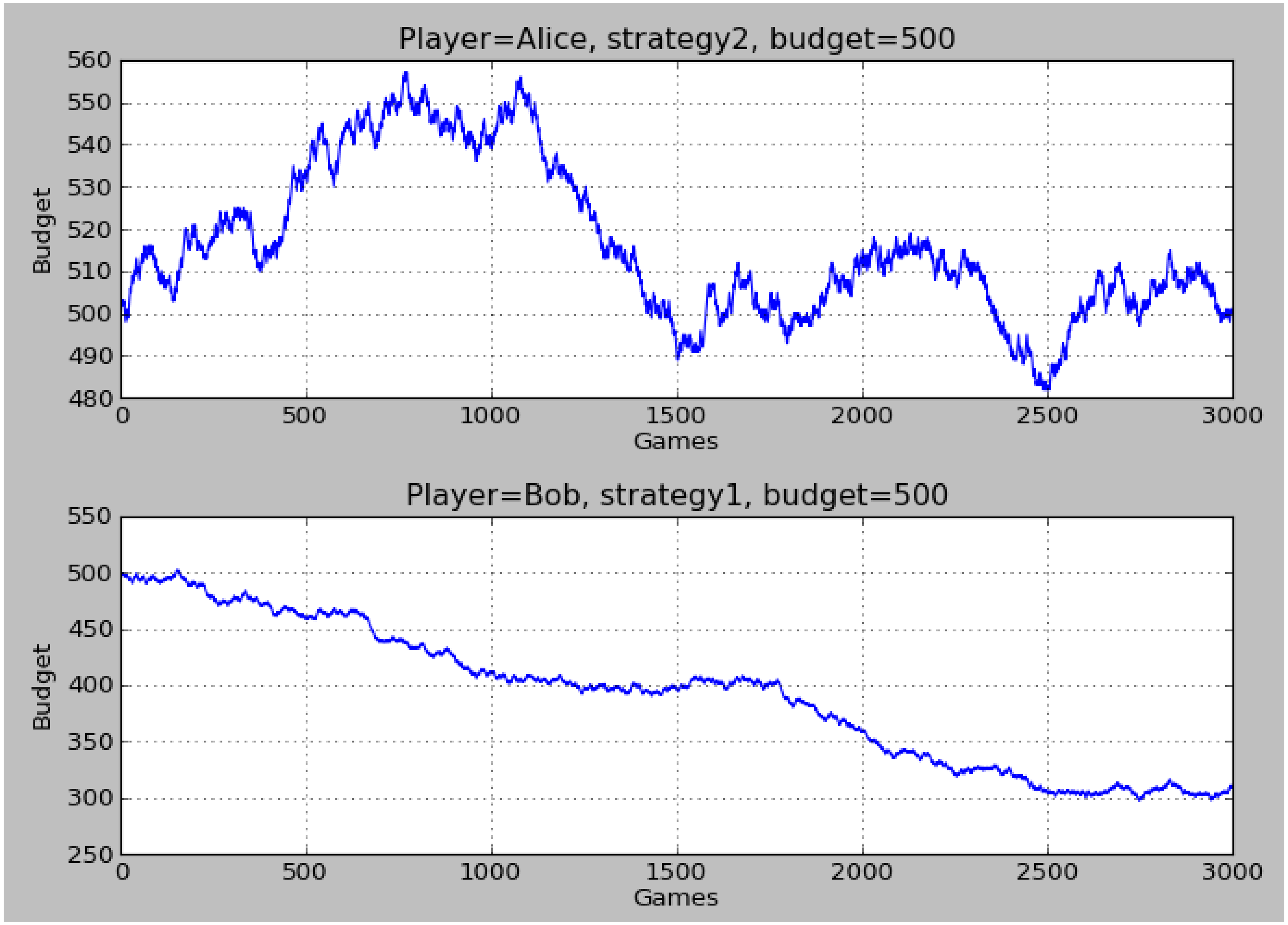


Simulation of 3000 Games (2)

Alice is using **strategy2**
Bob is using **strategy1**
If budget is 500K
(one bet = 1K)
Then it takes 10 hours
to make 50K
(assuming 100 games
per hour)

In fact, Alice can start
with a much smaller
budget: 30K
As she does not lose
more than 20K for the
first 3000 games !!
But has a potential to
win 50K !

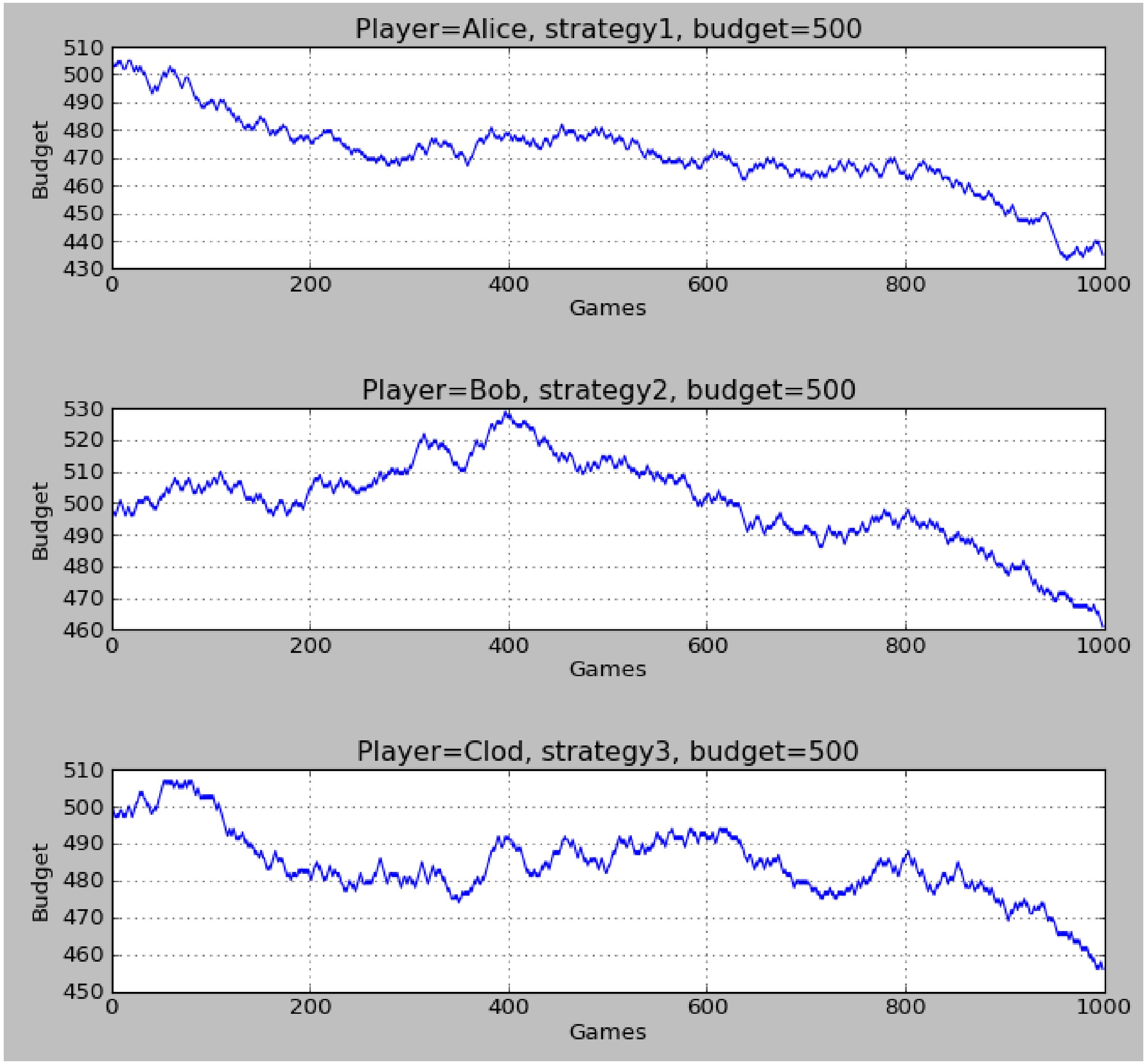
Warning!!! this is just
a simplistic example!
Do not try it in
a Casino ! ☺



Simulation of 1000 Games, 3 strategies

Alice: strategy1
Bob: strategy2
Clod: strategy3
Budget: 500K
Games: 1000

100 games per hour



Advanced Challenge: Machine Learning

- The previous experiments are useful for comparing existing strategies
- How about playing millions of games and improving our best strategy?
- After playing millions of games, we may find that our best strategy (strategy2) has some defects and can be fixed by some small changes to the tables
- Probabilistic strategies: after many games we can learn things such as: when player total is soft 18 and dealer is soft 15, then play should hit at probability 0.76 and stand in probability 0.26. These are probabilistic strategies
- Ideas for a future final project …