

```
#####
#
#           CLIMATE GAME           #
#       written by:                 #
#       Cezar Ionescu              #
#       ionescu@pik-potsdam.de     #
#
#       documentation:             #
#       Hannah Foerster            #
#       foerster@pik-potsdam.de   #
#
#####
'''
This is a game about the earth. Climate change needs to
be tackled in an effective way in order to prevent a rise
of temperature. This task is achieved by co-operation of the
players but at the same time stands in conflict to the
individual welfare maximising behaviour of each player.

This computer game is used as an analytical tool in order
to calibrate the boardgame version of the game in a senseful
way.
'''

from random import Random

class Features:
    '''This class initialises the features of the game. It defines the
    different tracks. Then it sets limits, prices, rewards, welfare
    units for auction, the insurance factor and the temperature reduction
    for each track. Then a list of climate cards indicates a name for each
    card. The same happens for policy cards. After this there are two functions.
    The first one generates a temperature table. The second one has the purpose
    to compute if a player can buy cities or not'''

    def __init__(self):

        # Board configuration

        self.tracks = []
        track = {"length":30, "climate":[5, 13, 28], \
                "policy":[3, 8, 11, 17, 20, 24, 26]}
        self.tracks.append(track)
        track = {"length":38, "climate":[3, 7, 9, 11, 14, 15, 17, 26, \
                28, 30, 31, 37], "policy":[1, 5, 18, 21, 24, 33, 35]}
        self.tracks.append(track)
        track = {"length":48, "climate":[0, 3, 5, 8, 9, \
                13, 16, 18, 19, 20, 23, 25, 28, 32, 38, 39, 43, \
                44], "policy":[1, 4, 10, 12, 15, 17, 22, 30, 34, \
                36, 37, 41, 46]}
        self.tracks.append(track)

        # Player limits
        self.max_cities = 9
        self.max_schools = 3

        # Prices
        self.gray_city_price = 6
        self.green_city_price = 10
        self.school_price = 20
        self.insurance_prices = [10,50]
        self.patent_chance_price = [30, 40, 50, 50, 60, 60]
        self.patent_chance = 3 # i.e., one in the number indicated

        # Rewards
        self.gray_city_reward = [5, 5, 5]
        self.green_city_reward = 5
```

```

self.school_reward_factor = 2

# Number of welfare units put on auction
self.wf_units_factor = [1, 1, 1]

# Insurance factor
self.insurance_factor = [8, 4, 2]

# Temperature reduction for each track
self.track_temp = [0.1, 0.2, 0.3]

# Climate cards
self.climate_cards = ["hurricane", "water", "sea_level", \
    "floods", "storms", "droughts", "malaria", "gulfstream", \
    "rain_forests", "friendly_weather", "yields", "methane", \
    "air_pollution", "arctic_ocean"]

# Policy cards
self.policy_cards = ["more_policy", "less_policy", \
    "breakthrough", "capital_export", "incentives", "migration", \
    "social_improvements", "local_crisis", "larger_crisis", \
    "corrupt_dictator", "strikes", "development"]

# Temperature table (represented as function)
def temp_table(self,p, n):
    ''' Computes a temperature table. '''
    return ((n + p-1) / p) * 0.1

def can_buy_cities(self, player, n):
    '''
    This method checks if the number of a players
    cities is smaller than his given maximum of cities.
    Returns true if the number of a players cities
    is less than his maximum numbers of cities. Returns
    false otherwise.
    '''

    return player.nr_cities() + n <= self.max_cities

class Bank:
    '''Class that contains the methods for the bank.'''

    def __init__(self, features=Features(), current_track=0):
        '''Initialisation values'''
        self.current_track = current_track
        self.features = features
        self.current_winner = -1
        self.current_bid = 0
        self.done = 0

    def auction(self, bundle, players):
        '''
        players = list of players
        self.current_winner = gets index -1
        self.current_bid = set to zero at the beginning of auction
        self.done = set to zero indicates that process is not finished

        The list players is checked, and if the players position is
        equal to the value of self.current_winner, then the function
        breaks. Otherwise it calls the function player.get_bid in order
        to obtain this players bid.

        If the obtained bid is larger than the current bid, then self.current_bid
        is set to that value (was zero at the beginning). self.current_winner
        is changed to the players position who did this bid.

        This starts over again for the next player. And so on.
        '''

```

If at the end the self.current_bid is equal to zero then the function return no winner and self.current_bid.
 If there is a winner it returns self.current_winner and the corresponding bid.

```
'''
self.current_winner = -1
self.current_bid = 0
self.done = 0
while 1:
    self.done = 1
    for player in players:
        if player.position == self.current_winner:
            break
        bid = player.get_bid(bundle, self.current_winner, \
            self.current_bid)
        if bid > self.current_bid:
            self.current_bid = bid
            self.current_winner = player.position
            self.done = 0
    if self.done == 1:
        break

if self.current_bid == 0:
    return (None, self.current_bid)

return (self.current_winner, self.current_bid)
```

```
def compute_wf_units(self, players):
```

```
''' Computes welfare units.
wfu      = welfare units
nr_players = length of list "players"
factor    = number of welfare units on auction on the current track
bundle    = welfare units / number of players
bundles   = list containing the bundle (number) for each player
```

Welfare units are computed by multiplying the sum of a players cities and his schools by factor. Should the wfus be zero, an empty list is returned.

Then bundle is computed as the result of the division welfare units / nr. players. bundles is set to be an empty list. Then to bundles is appended the bundle of each player. It is appended as an integer value.

For the range of the number of players modulo 2 is taken. Should that be equal to zero, to bundles is appended the rounded integer value of bundle.

Otherwise the integer value is appended

Then a reduce function is applied (don't know how this works) and the result is assigned to sum. If sum is now less than wfu: to the smallest number of the list bundles this rest is added.

If the sum is larger than wfu on the other hand, then sum is subtracted from the largest element of bundles.

Function then returns the list bundles (these are put on auction at the beginning of a round).

```
'''
wfu = 0
nr_players = len(players)
factor = self.features.wf_units_factor[self.current_track]
for player in players:
    wfu += factor * player.nr_cities() + player.nr_schools()
if wfu == 0: return []
bundle = (wfu*1.0)/(nr_players*1.0)
```

```

bundles = []
for i in range(nr_players):
    if i % 2 == 0:
        bundles.append(int(round(bundle)))
    else:
        bundles.append(int(bundle))
sum = reduce(lambda x, y: x+y, bundles)
if sum < wfu: # add rest to smallest
    bundles[bundles.index(min(bundles))] += wfu - sum
if sum > wfu: # subtract from biggest
    bundles[bundles.index(max(bundles))] += wfu - sum

return bundles

```

class Game:

```

def __init__(self, players, features=Features()):
    '''Initialises some things for the game.
    Creates a dictionary for handling
    events that contains a string which gets assigned to a function.'''
    self.players = players
    self.climate_cards = Cards("Climate cards", \
                               features.climate_cards, seed)
    self.policy_cards = Cards("Policy cards", \
                               features.policy_cards, seed)
    self.features = features
    self.current_player = 0

    self.current_track = 0
    self.current_temp = 0
    self.current_round = 0
    self.board = Board()
    self.bank = None

    self.educated_players = []

    self.handle_event = { \
# climate events
"hurricane":self.handle_hurricane, \
"water":self.handle_water, \
"sea_level":self.handle_sea_level, \
"floods":self.handle_floods, \
"storms":self.handle_storms, \
"droughts":self.handle_droughts, \
"malaria":self.handle_malaria, \
"gulfstream":self.handle_gulfstream, \
"rain_forests":self.handle_rain_forests, \
"friendly_weather":self.handle_friendly_weather, \
"yields":self.handle_yields, \
"methane":self.handle_methane, \
"air_pollution":self.handle_air_pollution, \
"arctic_ocean":self.handle_arctic_ocean, \
# policy events
"more_policy":self.handle_more_policy, \
"less_policy":self.handle_less_policy, \
"breakthrough":self.handle_breakthrough, \
"capital_export":self.handle_capital_export, \
"incentives":self.handle_incentives, \
"migration":self.handle_migration, \
"social_improvements":self.handle_social_improvements, \
"local_crisis":self.handle_local_crisis, \
"larger_crisis":self.handle_larger_crisis, \
"corrupt_dictator":self.handle_corrupt_dictator, \
"strikes":self.handle_strikes, \
"development":self.handle_development \

```

```

}

for cell in self.board.tracks[self.current_track]:
    print str(cell), " ",
print ""

def one_round(self,nr):
    '''Function depicting one round of the game.
    At the beginning, round is increased by one (counting!).
    self.bank is class bank with all its features.
    wf_units = computation of welfare units from class Bank.

    01) auction of welfare bundles:
    For all elements of the list wf_units a bundle is assigned
    to be the element of wf_units at place i. It is printed, that
    this bundle (a number) is now put to an auction. The winner
    and the amount of that auction are the results of the function
    self.bank.auction from the above class. It might be that there
    is no winner, then it is printed that there is no winner.If on
    the other hand there is a winner, winname is set to that players
    name and it is printed that he has won the auction. Then the winners
    welfare is changed, and the amount is subtracted from his budget.
    Then the auction is finished.

    ...

    self.current_round += 1
    self.bank = Bank(self.features, self.current_track)
    wf_units = self.bank.compute_wf_units(self.players)
    print "Begging auction of " + str(wf_units) + " units"
    for i in range(len(wf_units)):
        bundle = wf_units[i]
        print "Auctioning bundle " + str(i) + " of " + \
            str(bundle) + " units"
        (winner, amount) = self.bank.auction(bundle, self.players)
        if winner is not None:
            winname = self.players[winner].name
        else:
            winname = "None"
        print "Auction has been won by player " + winname + \
            " with a bid of " + str(amount)
        if winner is not None:
            players[winner].change_wf(bundle)
            players[winner].change_budget(-amount)

    print "Auction finished"

    '''sanctioned = an empty list
    02) computation of income
    03) if a player has obligations, then he has to see if he can resolve it
    -> resolve_obligations is called
    04) deals with other players are made, if s > -1 then sanction is filled
    with s
    05) investments take place via method player.invest
    06) players move to a new position, via method player.move
    07) an event occurs, either regular or policy
    08) the event has to be handles via method handle_event
    09) something with sanctioned happens

    ...

    sanctioned = []
    for player in self.players:
        self.current_player = player

```

```

self.add_income(player)
player.resolve_obligations(self.players)
s = player.make_deals(self.players)
if s > -1: sanctioned.append(s)
player.invest(self.players)
pos = player.move(self.board)
print "Player " + player.name + " moves to position " \
      + str(pos)
kind = self.board.tracks[self.current_track][pos].kind
if kind != "regular":
    if kind == "climate":
        event = self.climate_cards.get_random_card()
        print "Player " + player.name + " is exposed to" \
              + " a climate event"
    if kind == "policy":
        event = self.policy_cards.get_random_card()
        print "Player " + player.name + " is exposed to" \
              + " a political event"

self.handle_event[event]()

for s in sanctioned:
    votes = sanctioned.count(s)
    if votes*2 > len(self.players):
        splayer = self.players[s]
        splayer.obligations.append(splayer.resolve_sanction)
        print "Player " + self.players[s].name + " has been "\
              + "sanctioned for the next round"

'''
11) insurances are set to zero if they have not been used
12) information about each player is printed
'''

# Unused insurance is lost
for player in self.players:
    player.insurance = 0

for player in self.players:
    print str(player)

'''
13) the new temperature is computed via compute_temperature
'''
print "CURRENT ROUND IS: NR. ", self.current_round
temp = self.compute_temperature()
print "Total temperature increase is ", self.current_temp
print "Temperature increment is ", temp
print

'''
14) if the temperature is greater/equal to 6, then all lose. via
    all_lose function

15) if temp (change of temperature) is less than zero, the game
    is over, via game_over function
16) if the current temperature is larger than 1 and the current
    track is 0, then next track is computed via next_track

17) if the current temperature is larger than 3 and the current
    track is 1, then next track is computed via next_track

18) if game is not over returns 0
'''

if self.current_temp >= 6:
    self.all_lose()

```

```
        return 1

    if temp < 0:
        self.game_over()
        return 1

    if self.current_temp > 1 and self.current_track == 0:
        self.next_track()

    if self.current_temp > 3 and self.current_track == 1:
        self.next_track()

    return 0

def bankrupt(self, player):
    ''' This function computes if a player is characterised as being bank-
    rupt or not. If the number of cities of a player is equal to zero and
    his budget is less than 5 then return 1 (bankrupt), else return 0.'''

    if player.nr_cities() == 0:
        if player.budget < 5:
            return 1
    return 0

def reset(self, player):
    '''This function resets the attributes of a player
    to some fixed values. His number of cities are set to 1, his
    budget is set to 20. He is not under dictatorship, and his welfare is
    set to zero. '''

    player.gray_cities = 1
    player.budget = 20
    player.dictator = 0
    player.wf = 0

def add_income(self, player):
    '''This function adds income to a players budget.
    Firstly, it is determined if the player is bankrupt. If it
    is so, his initial attributes are reset regarding the reset method.
    If the player is under dictatorship, then
    player.dictator is set to zero. If the player is not (yet) an educated
    player and(!) he has more than one school, then he is added to the list
    of educated players. If the player is an educated player his school_factor
    is 2, otherwise it is zero.

    Players income is zero. To the players income is added: reward for gray
    cities at the current track times the number of gray cities, the green
    city award (independent of track) times the number of green cities.

    If the player is on a strike, his income is divided by 2 (really?) and the
    strice indicator is set to zero.

    To the players budget is added the school_factor (2 or 1) times the income.
    '''
    if self.bankrupt(player):
        self.reset(player)
        return
    if player.dictator != 0:
        player.dictator = 0
        return
    if player not in self.educated_players and player.schools > 0:
        self.educated_players.append(player)
    if player in self.educated_players:
        school_factor = 2
    else:
        school_factor = 1
```

```

income = 0
income += (self.features.gray_city_reward[self.current_track] \
          * player.gray_cities)
income += self.features.green_city_reward * player.green_cities
if player.strike:
    income /= 2
    player.strike = 0
player.budget += school_factor * income

```

```

def next_track(self):
    '''This method computes the next track if a player changes tracks.
    Computes length1 as length of current track and length2 as length
    of new track.
    Then computes for each player his old position and his new position.
    The new position is computed as the length2 times the old position
    divided by length1.
    The new position on the board is then the rounded integer value of the
    new position. The players track is set to be this new track. The
    current track is also set to that track. For each player the current
    track is set to new_track.
    '''
    print "Changing tracks"
    new_track = self.current_track + 1
    # reposition the players
    length1 = self.features.tracks[self.current_track]["length"]
    length2 = self.features.tracks[new_track]["length"]
    print "The players are moving their pawns as follows:"
    for player in self.players:
        old_pos = player.board_pos
        new_pos = (length2*old_pos*1.0)/(length1*1.0)
        player.board_pos = int(round(new_pos))
        player.track = new_track
        print "\tPlayer " + player.name + " moves from " \
              + str(old_pos) + " to " + str(player.board_pos)
    # change track
    self.current_track = new_track
    for player in self.players:
        player.current_track = new_track
    print "Current track is now ", self.current_track

def compensate(self, player):
    '''
    track = current track
    factor = insurance factor of this track (from features)
    This function changes the players budget by adding to
    it factor times insurance of the player (if he has one)
    Then players insurance is then set to 0 (only helps once?)
    '''
    track = self.current_track
    factor = self.features.insurance_factor[track]
    player.change_budget(factor*player.insurance)
    print "Player" + player.name + " has been compensated by the " \
          + " insurance with " + str(factor*player.insurance) \
          + " units"
    player.insurance = 0

# event handling starts
# first climate events

def handle_hurricane(self):
    '''This function handles a hurricane event
    amounts = list containing integers
    cities = list containing integers

    If the current track is track zero, then just
    '''

```

```
    return.

    player is the current player and the actual amount
    is the integer from the list amounts indicated by
    the current track (1,2,3)
    actual_cities is the same with respect to the list
    cities.
    Then the player gets reduced his cities by the actual
    amount by the reduce_cities method. His welfare units
    are lowered by actual_amount, too. Then a message con-
    taining this information is printed. Then the compensate
    method is called, which will compensate him some of his
    damage if he has an insurance.
    '''
    amounts = [10, 50, 150]
    cities = [0, 0, 0]

    player = self.current_player
    actual_amount = amounts[self.current_track]
    actual_cities = cities[self.current_track]

    player.reduce_cities(actual_cities)

    player.change_wf(-actual_amount)

    print "Player " + player.name + " has been hit by a hurricane"
    print "He has lost " + str(actual_amount) + " units of welfare"
    # print "      " + str(actual_cities) + " cities"
    self.compensate(player)

def handle_water(self):

    '''This function handles a water event
    amounts = list containing integers
    cities = list containing integers

    If the current track is track zero, then just
    return.

    player is the current player and the actual amount
    is the integer from the list amounts indicated by
    the current track (1,2,3)
    actual_cities is the same with respect to the list
    cities.
    Then the player gets reduced his cities by the actual
    amount by the reduce_cities method. His welfare units
    are lowered by actual_amount, too. Then a message con-
    taining this information is printed. Then the compensate
    method is called, which will compensate him some of his
    damage if he has an insurance.
    '''
    amounts = [0, 80, 160]
    cities = [0, 0, 0]

    if self.current_track == 0: return

    player = self.current_player
    actual_amount = amounts[self.current_track]
    actual_cities = cities[self.current_track]

    player.reduce_cities(actual_cities)

    player.change_wf(-actual_amount)

    print "Player " + player.name + " has been hit by water \
    shortage"
```

```
print "He has lost " + str(actual_amount) + " units of welfare"
# print "          and " + str(actual_cities) + " cities"
self.compensate(player)

def handle_sea_level(self):
    '''Analogue to above event, with different numbers in
    amounts. '''
    amounts = [0, 70, 200]
    cities = [0, 0, 0]

    if self.current_track == 0: return

    player = self.current_player
    actual_amount = amounts[self.current_track]
    actual_cities = cities[self.current_track]

    player.reduce_cities(actual_cities)

    player.change_wf(-actual_amount)

    print "Player " + player.name + " has been hit by sea-" + \
          "level rise"
    print "He has lost " + str(actual_amount) + " units of welfare"
    # print "          and " + str(actual_cities) + " cities"
    self.compensate(player)

def handle_floods(self):
    '''Analogue to above event, with different numbers in
    amounts. '''
    amounts = [10, 20, 35]
    cities = [0, 0, 0]

    player = self.current_player
    actual_amount = amounts[self.current_track]
    actual_cities = cities[self.current_track]

    player.reduce_cities(actual_cities)

    player.change_wf(-actual_amount)

    print "Player " + player.name + " has been hit by floods"
    print "He has lost " + str(actual_amount) + " units of welfare"
    # print "          and " + str(actual_cities) + " cities"
    self.compensate(player)

def handle_storms(self):
    '''Analogue to above event, with different numbers in
    amounts. '''
    amounts = [30, 70, 90]
    cities = [0, 0, 0]

    player = self.current_player
    actual_amount = amounts[self.current_track]
    actual_cities = cities[self.current_track]

    player.reduce_cities(actual_cities)

    player.change_wf(-actual_amount)

    print "Player " + player.name + " has been hit by storms"
    print "He has lost " + str(actual_amount) + " units of welfare"
    # print "          and " + str(actual_cities) + " cities"
    self.compensate(player)

def handle_droughts(self):
    '''Analogue to above event, with different numbers in
```

```

    amounts. '''
    amounts = [30, 60, 110]
    cities = [0, 0, 0]

    player = self.current_player
    actual_amount = amounts[self.current_track]
    actual_cities = cities[self.current_track]

    player.reduce_cities(actual_cities)

    player.change_wf(-actual_amount)

    print "Player " + player.name + " has been hit by droughts"
    print "He has lost " + str(actual_amount) + " units of welfare"
    # print "          and " + str(actual_cities) + " cities"
    self.compensate(player)

def handle_malaria(self):
    '''Analogue to above event, with different numbers in
    amounts. '''
    amounts = [0, 70, 120]
    cities = [0, 0, 0]

    player = self.current_player
    actual_amount = amounts[self.current_track]
    actual_cities = cities[self.current_track]

    player.reduce_cities(actual_cities)

    player.change_wf(-actual_amount)

    print "Player " + player.name + " has been hit by malaria"
    print "He has lost " + str(actual_amount) + " units of welfare"
    # print "          and " + str(actual_cities) + " cities"
    self.compensate(player)

def handle_gulfstream(self):
    '''
    Function than handles gulfstream event.
    If the current track is less than 2 another
    event card has to be picked and this event has
    to be handled.
    Otherwise the player is the current player and it
    is printed that he has been hit by gulfstream and loses half
    his welfare units. Then his welfare units are reduced by half
    via method change_wf. The climate_card "gulstream" is removed
    from the list of climate cards.
    '''
    if self.current_track < 2:
        # Pick a different card
        event = self.climate_cards.get_random_card()
        self.handle_event[event]()
        return
    player = self.current_player
    print "Player " + player.name + " is exposed to gulfstream" \
        + " breakdown"
    print "\tand loses half his wf"
    player.change_wf(-player.wf/2)
    # player.reduce_cities(int(round(player.nr_cities()/2.0)))
    self.climate_cards.remove_card("gulfstream")

def handle_rain_forests(self):
    '''This function handles rain_forest cards. The current
    temperature is increased by 0.1. A message is printed.'''
    self.current_temp += 0.1
    print "Tropical deforestation leads to temperature increase"

```

```

    print "of .1 degree C"

def handle_friendly_weather(self):
    '''This function handles friendly_weather event.
    The player is the current player, cash is set to 40.
    To the players budget these 40 are added via method
    change_budget. The friendly_weather card is removed from
    list of climate_cards. A message is printed.
    '''
    player = self.current_player
    cash = 40
    player.change_budget(cash)
    self.climate_cards.remove_card("friendly_weather")
    print "Player " + player.name + " profits from nicer weather."
    print "He has earned" + str(cash) + " Terra."

def handle_yields(self):
    '''This function handles the climate event yields. It works
    analogously to handle_water, just that the values of the
    integers in amounts are negative, which means, that the current
    player gains from this event. His cities are increased, so are
    his welfare units. '''
    amounts = [-10, -50, 0]
    cities = [0, 0, 0]

    player = self.current_player
    actual_amount = amounts[self.current_track]
    actual_cities = cities[self.current_track]

    player.reduce_cities(actual_cities)

    player.change_wf(-actual_amount)

    print "Player " + player.name + " profits from higher yields."
    print "He has gained " + str(-actual_amount) + " units of" \
    + "welfare."
    self.compensate(player)

def handle_methane(self):
    '''This functions handles climate_event methane
    If the current track is less than one, another
    card is picked and this event is handled.
    Then the temperature is increased by 1 and the
    methane card is removed from the list of
    climate_cards. A message is printed.

    '''
    # TEMPERATURE INCREASE REQUIRED - TAKE CARD OUT
    if self.current_track < 1:
        event = self.climate_cards.get_random_card()
        self.handle_event[event]()
        self.current_temp += 1
        self.climate_cards.remove_card("methane")
        print "Methane release leads to temperature increase"
        print "of 1 degree C"

def handle_air_pollution(self):
    '''This function handles the climate event air_pollution. It works
    analogously to handle_water '''

    amounts = [10, 50, 100]
    cities = [0, 0, 0]

    player = self.current_player
    if player.nr_cities() < 2*player.gray_cities:

        actual_amount = amounts[self.current_track]

```

```

    actual_cities = cities[self.current_track]

    player.reduce_cities(actual_cities)

    player.change_wf(-actual_amount)

    print "Player " + player.name + " suffers from air pollution."
    print "He has lost" + str(actual_amount) + " + of welfare."

    self.compensate(player)

def handle_arctic_ocean(self):
    '''This method handles the climate_event arctic_ocean.
    cash is set to 100 and the current players' budget is increased
    by 100 via the change_budget method. The arctic_ocean card is
    removed from the list of climate_cards. A message is printed.
    '''
    cash = 100

    player = self.current_player

    player.change_budget(cash)
    self.climate_cards.remove_card("arctic_ocean")
    print "Player " + player.name + " profits from sea access " \
    "via arctic ocean"
    print "He has earned" + str(cash) + " Terra."

# then policy events
def handle_more_policy(self):
    '''This method handles policy event more_policy.
    player is the current player. If his number of gray cities
    is larger than one and(!) his overall number of cities is
    less than 2 times his gray cities he gets subtracted one gray
    city. A message is printed (doesnt he get subtracted 1 city?)
    '''
    player = self.current_player
    if player.gray_cities > 1 and \
        player.nr_cities() < 2*player.gray_cities:
        player.gray_cities -= 1
        print "Player " + player.name + " loses two gray cities."

def handle_less_policy(self):
    '''This method handles policy event less_policy.
    player is the current player. If he has more than one
    green city and his overall number of cities is less than
    2 times his green cities, then he gets subtracted one
    green city (or two, where does it say two?)

    '''
    player = self.current_player
    if player.green_cities > 1 and \
        player.nr_cities() < 2*player.green_cities:
        player.green_cities -= 1
        print "Player " + player.name + " loses two green cities."

def handle_breakthrough(self):
    '''This function handles policy event breakthrough.
    The price for a green city is reduced by two and the break-
    through card is removed from the list of policy cards. '''
    self.features.green_city_price -= 2
    self.policy_cards.remove_card("breakthrough")
    print "Price of green cities has been reduced by two units."

def handle_capital_export(self):
    '''This function handles policy event capital_export.
    player is the current player. his cex obligation is

```

```

increased by 2. to the obligation list list is appended
that the player has to resolve this obligation. ob is assigned
to player.resolve_obligations. If ob is larger than 0 then it is
printed that the player still has to resolve his obligation.
if it is zero, the player has resolved his obligation.
'''
player = self.current_player
player.cex_obligation += 2
player.obligations.append(player.resolve_cex_obligation)
ob = player.resolve_obligations(self.players)
print "Player " + player.name + " was obligated to build " \
      + "two green cities for the needy."
if ob > 0:
    print "He still has to build " + str(ob) \
          + " in future rounds."
else:
    print "He has resolved this obligation."

def handle_incentives(self):
    '''This method handles policy event handle_incentives
    player is the current players. incentive is set to 5.
    amount is a variable set to 5 times of the players
    gray cities. Then , via method change_budget, from the
    players budget is substracted the amount. A message is
    printed.
    '''
    player = self.current_player
    incentive = 5
    amount = player.gray_cities*incentive
    player.change_budget(-amount)
    print "Player " + player.name + " pays " + str(amount) \
          + " penalty for gray cities."

def handle_migration(self):
    '''This function handles policy event migration
    wfs is set to be an empty list. For all players
    to this list is appended their number of welfare
    units. Then indices for the loser and the winner
    are produced. Then loser and winner are computed
    The loser, the player with the least welfare units
    gets substracted a city. If the winner has more than
    9 cities he recieves 10 units of welfare, via change_wf.
    Otherwise the winner receives another gray city, by adding
    one gray city to his cities.
    '''
    wfs = []
    for p in players:
        wfs.append(p.wf)
    loser_index = wfs.index(min(wfs))
    winner_index = wfs.index(max(wfs))
    loser = players[loser_index]
    winner = players[winner_index]
    loser.reduce_cities(1)
    print "Player " + loser.name + " has lost a city."

    if winner.nr_cities() >= 9:
        winner.change_wf(10)
        print "Player " + winner.name + " has received 10 units" \
              + " of welfare."
    else:
        winner.gray_cities += 1
        print "Player " + winner.name + " has received an extra" \
              + " gray city."

def handle_social_improvements(self):
    '''This method handles policy event social_improvements.
    player is the current player. If he has less than 3 schools,

```

then it is printed that he has to build a school. If his budget is larger than the school price then his budget is reduced by the school price and he gets added one school to his schools. Then a message about that is printed. If his budget is less than the school price, he gets an obligation to build a school. To the obligation list this is appended. A message about that is printed.

```

...
player = self.current_player
if player.schools < 3:
    print "Player " + player.name + " must build a school"
    if player.budget >= self.features.school_price:
        player.budget -= self.features.school_price
        player.schools += 1
        print "and he does it"
        return
    else:
        player.obligations.append(player.resolve_own_school_ob)
        print "and he gets an obligation to do so"
        return
print "null event"

```

```

def handle_larger_crisis(self):
    '''This method handles policy event larger_crisis.
    player is the current player. pos is his position,
    nr_players is the length of the list with players.
    left = the position of the player left to current player
    player_left = player at position left
    right = position of right player
    player_right = player at position right

```

if player_right is unequal to player_left, a list
 plist is constructed with the three players, player,
 player_left and player_right in.
 otherwise plist contains only player and player_left.

For each player in this list, the welfare units are
 cut by half.

```

...
print "A financial crisis has occurred!"
player = self.current_player
pos = player.position
nr_players = len(self.players)
left = pos-1
player_left = self.players[left]
right = (pos+1) % nr_players
player_right = self.players[right]

if player_right != player_left:
    plist = [player, player_left, player_right]
else:
    plist = [player, player_left]

for p in plist:
    p.change_wf(-p.wf/2)
    print "Player " + p.name + " has lost half of the wf"

```

```

def handle_local_crisis(self):
    '''This method handles policy event local_crisis

    player is the current player. His welfare units are
    cut by half via method chage_wf. A message is printed

    ...
player = self.current_player

```

```
player.change_wf(-player.wf/2)
print "Player " + player.name + " has lost half his wf."

def handle_corrupt_dictator(self):
    '''This method handles policy event corrrput_dictator.
    player is the current player. This players' welfare units
    are cut half via method change_wf. Then player.dictator is
    set to 1 (was zero), meaning that he is under dictatorship.
    A message is printed that he gut cut welfare and will not
    receive income at the end of the round.
    '''

    player = self.current_player
    player.change_wf(-player.wf/2)
    player.dictator = 1
    print "Player " + player.name + " is under dictatorship."
    print "He has lost half his wf and will get no income at" \
        + " the end of this round."

def handle_strikes(self):

    '''This method handles policy event strikes.
    player is current player. player.strike is set to 1 (was 0)
    which indicates that he has been hit by a strike. He then
    gets the obligation to sell a city to the bank if method
    resolve_strike_ob returns 1. This obligation
    is appended to his list of obligations.
    If resolve_strike_ob returns 0 it is printed that he has sold
    a city to the bank (this is done via resolve_strike_ob)
    '''

    player = self.current_player
    player.strike = 1
    print "Player " + player.name + " has been hit by a strike."
    if player.resolve_strike_ob(self.players):
        print "He is obligated to sell a city to the bank."
        player.obligations.append(player.resolve_strike_ob)
    else:
        print "He has sold a city to the bank."

def handle_development(self):
    '''This method handles policy event development.

    school_ob and cities_ob are lists generated at the
    beginning, containing integers.

    The actual school and city obligations are determined
    and are dependent of the current track.

    player is the current player
    To his school and cities obligations he gets added the
    new (= actual) obligations (He could already have other
    school obligations).

    Then resolve_obligations is called and eventually the player
    will resolve his obligations. If he did so an appropriate
    message will be printed. Otherwise also.
    resolve_obligations will also take care of setting the values
    of the obligations to zero if he has fulfilled them.

    '''

    schools_ob = [2, 3, 3]
    cities_ob = [0, 1, 3]

    actual_schools_ob = schools_ob[self.current_track]
    actual_cities_ob = cities_ob[self.current_track]

    player = self.current_player
```

```

player.schools_ob += actual_schools_ob
player.cities_ob += actual_cities_ob
player.obligations.append(player.resolve_dev_ob)
ob = player.resolve_obligations(self.players)
print "Player " + player.name + " has been obligated " \
      + " to build " + str(actual_schools_ob) + " schools " \
      + " and " + str(actual_cities_ob) + " cities for the needy"
if ob != 0:
    print "...and his obligations will continue in the next" \
          + " rounds."
else:
    print "He has fulfilled these obligations."

# event handling ended

def compute_temperature(self):
    '''This method computes temperature.
    temp_table = temperature table from features
    track_temp = track temperature from features
    nr_gray_cities is set to 0.

    For all players it is computed how many gray cities
    they own. If the overall number of gray cities is
    equal to 0, 0 is returned.
    Otherwise the number of players is computed and then
    the temperature temp is determined as the combination
    of number of players and number of gray cities from the
    temp_table. The temperature is then decreased by the
    track temperature of the current track. Then the current
    temperature is increased by temp. temp is returned.
    '''

    temp_table = self.features.temp_table
    track_temp = self.features.track_temp
    nr_gray_cities = 0
    for player in self.players:
        nr_gray_cities += player.gray_cities
    if nr_gray_cities == 0:
        return 0.0
    nr_players = len(self.players)
    temp = temp_table(nr_players, nr_gray_cities)
    temp -= track_temp[self.current_track]
    self.current_temp += temp
    return temp

def all_lose(self):
    ''' The current temperature is printed and a message
    that everybody loses. Method returns 1.'''

    print "Temperature is now ", self.current_temp
    print "EVERYBODY LOSES!!!"
    return 1

def game_over(self):
    '''A message is printed, temperature has been decreased.
    wfs = empty list
    the welfare units of all players are appended to wfs.
    winner = player with the most welfare units
    The winner is announced.
    '''
    print "Temperature has been decreased!"
    wfs = []
    for player in players:
        wfs.append(player.wf)
    winner = players[wfs.index(max(wfs))]
    print "Winner is ", winner.name

```

```

    return 1

def __str__(self):
    '''print method for this class'''
    return "Nr of players: " + str(self.nr_players) + "\n" \
        + str(self.climate_cards) + str(self.policy_cards)

class Cards:

def __init__(self, name, cards, seed):
    self.name = name
    self.cards = cards
    self.rand = Random(seed)

def get_random_card(self):
    '''Returns a random element from self.cards.'''
    return self.rand.choice(self.cards)

def remove_card(self, card):
    '''Removes element card from list self.cards. '''
    self.cards.remove(card)

def __str__(self):
    '''Prints informationa about this class.'''
    return self.name + ":\n\t" + str(self.cards) + "\n"

class Prob_Strategy:
def __init__(self, probs, features=Features()):
    '''Initialisation of probabilities'''
    self.features = features
    self.bid_prob = probs["bid"]
    self.ask_bad_guy = probs["ask_bad_guy"]
    self.sanction_prob = probs["sanction"]
    self.pat_prob = probs["ask_pat"]
    self.sell_pat_prob = probs["sell_pat"]
    self.scrap_prob = probs["scrap"]
    self.gray_prob = probs["gray"]
    self.insure_prob = probs["insurance"]
    self.school_prob = probs["school"]
    self.buy_pat_prob = probs["buy_pat"]
    self.green_prob = probs["green"]
    self.more_pat_prob = probs["more_pat"]

def get_patent_price(self, players):
    '''Method to get price of a patent
    nr_patents is set to 0. For each player
    his number of patents is added to nr_patents.
    We obtain the overall number of patents. If this
    number is larger than 5 0 is returned.
    Otherwise method patent_chance_price with the
    number of patents as an argument is called.

    ...
    nr_patents = 0
    for player in players:
        nr_patents += player.patents
    if nr_patents > 5:
        return 0
    return self.features.patent_chance_price[nr_patents]

def get_bid(self, player, bundle, current_winner, \
            current_bid):

    '''Method that gets a bid.
    If the budget of the player is larger than the current
    bid then it is decided if he should bid or not by
    cheking bid probs. (VERBESSERN)

```

```

...
    # check if bid possible
if player.budget > current_bid:
    # decide whether to bid
    if player.random.random() < self.bid_prob:
        return current_bid + 1
    else:
        return 0

def make_deals(self, player, players):
    '''Method that makes deals.
    gray_cities is set to be an empty list.
    sanction is set to -1

    A list with the number of gray cities of each player
    is created. The bad guy is the player with the maximum
    of gray cities. If the bad guy is not player.position(???)
    and the player current track is not the 0th,.....

...
gray_cities = []
sanction = -1
for player in players:
    gray_cities.append(player.gray_cities)
bad_guy = gray_cities.index(max(gray_cities))
if bad_guy != player.position and \
    player.current_track > 0:
    if player.random.random() < self.ask_bad_guy \
        and players[bad_guy].gray_cities > 0:
        max_cities = players[bad_guy].gray_cities
        # ask_for = player.random.randint(1, max_cities)
        ask_for = 1
        offer = ask_for*player.random.randint(0, 5)
        if offer > player.budget:
            offer = player.budget
        print "Player " + player.name + " wants player " \
            + players[bad_guy].name + " to scrap " \
            + str(ask_for) + " city, offering to pay " \
            + str(offer) + " in return"
        ans = players[bad_guy].make_offer(player, \
            ask_for, offer)
        if ans == 0:
            print "...and is refused"
            if player.random.random() < self.sanction_prob:
                print "...and he decides to move for sanction"
                sanction = bad_guy
        else:
            print "...and his offer is accepted"
            player.budget -= offer

if player.patents == 0:
    for p in players:
        if p.patents > 0:
            print "Player " + player.name + " asks " \
                + p.name + " for patent"
            ans = p.ask_for_patent(player)
            if ans > 0 and ans <= player.budget:
                if player.random.random() < self.pat_prob:
                    p.notify_accept(ans)
                    player.budget -= ans
                    player.patents += 1
                    print "...and he gets it for " + \
                        str(ans) + " units"
                    break
            else:
                print "...but doesn't like the price"

```

```
        else:
            print "...but is refused"

    return sanction

def ask_for_patent(self, player, asker):
    '''Method to ask for patent.
    If the budget of the asker is less than 10, return 0.
    Otherwise a random number is picked and if this number
    is smaller than the sell_pat_prob, and the player has more
    than one patent, then the amount is something in the random
    interval of 10 and the askers budget.
    returns amount.
    '''
    if asker.budget < 10:
        return 0
    if player.random.random() < self.sell_pat_prob and \
        player.patents > 1:
        amount = player.random.randint(10, asker.budget)
        return amount
    return 0

def make_offer(self, player, buyer, ask_for, offer):
    '''Method to make an offer. If a random number is
    smaller than scrap_prob 0 is returned. Otherwise
    the players budget is increased by offer and his gray
    cities are reduced by ask_for.
    returns 1.
    '''
    if player.random.random() < self.scrap_prob:
        return 0
    player.budget += offer
    player.gray_cities -= ask_for
    return 1

def buy_grays(self, player, players):
    '''
    gray_price = price set in features
    can_buy = method can_buy from features
    name = name of player

    If players budget is larger than the price of a gray city and(!)
    can_buy is fulfilled then a random number is picked. If this
    number is less than the value of self.gray_prob then
    the player buys a gray city. A message is printed and his
    number of gray cities is increased by one and his budget is
    decreased by the price of the gray city.
    '''
    gray_price = self.features.gray_city_price
    can_buy = self.features.can_buy_cities
    name = player.name
    if player.budget >= gray_price and can_buy(player, 1):
        if player.random.random() < self.gray_prob:
            print "Player " + name + " buys a gray city"
            player.gray_cities += 1
            player.budget -= gray_price

def invest(self, player, players):
    '''LATER'''
    school_price = self.features.school_price
    gray_price = self.features.gray_city_price
    green_price = self.features.green_city_price
    ins_price = self.features.insurance_prices
```

```

    can_buy = self.features.can_buy_cities

    name = player.name

#     if player.budget >= gray_price and can_buy(player, 1):
#         if player.random.random() < self.gray_prob:
#             print "Player " + name + " buys a gray city"
#             player.gray_cities += 1
#             player.budget -= gray_price

    self.buy_grays(player, players)

    if player.budget >= min(ins_price) and \
        player.budget < max(ins_price):
        # player may buy only minimal insurance
        if player.random.random() < self.insure_prob:
            print "Player " + name + " buys " + str(ins_price[0]) \
                + " units of insurance"
            player.insurance += min(ins_price)
            player.budget -= min(ins_price)

    if player.budget >= max(ins_price):
        # player may buy small or big insurance
        if player.random.random() < self.insure_prob:
            ins = player.random.randint(1, 2)
            if ins < 2:
                print "Player " + name + " buys " + \
                    str(ins_price[ins]) \
                    + " units of insurance"
                player.insurance += ins_price[ins]
                player.budget -= ins_price[ins]

    if player.budget >= school_price and player.schools < 3:
        # decide whether to buy schools
        if player.random.random() < self.school_prob:
            print "Player " + player.name + " buys a school"
            player.schools += 1
            player.budget -= school_price

    if player.schools >= 3:
        price = self.get_patent_price(players)
        if price != 0 and player.budget >= price:
            if player.random.random() < self.buy_pat_prob:
                print "Player " + player.name \
                    + " tries to get a patent"
                patent_chance = self.features.patent_chance
                player.budget -= price
                if player.random.randint(1, patent_chance) == 1:
                    print "and gets it!"
                    player.patents += 1
                    # halve the probability of a patent aquisition
                    self.buy_pat_prob = self.more_pat_prob
                else:
                    print "but doesn't get it..."

    if player.patents > 0 and player.budget >= green_price and \
        can_buy(player, 1):
        if player.random.random() < self.green_prob:
            print "Player " + player.name + " buys a green city"
            player.green_cities += 1
            player.budget -= green_price

class Adaptive_Strategy(Prob_Strategy):
    '''This is the class containing methods for the Adaptive Strategy for the
    player LessRandom. '''

```

```
def get_bid(self, player, bundle, current_winner, \
           current_bid):

    '''Method that computes a bid.
    The old bid probability is set to self.bid_prob
    rest is the players budget minus the current bid minus one.
    If the player number of cities is 9 or the rest is larger than
    three times the gray city price, then the bid probability is
    set to 1, otherwise to 0.

    The bid is now set to be the result of the method get_bid
    from the class Prob_Strategy.

    Then the bid probability is reset to the old bid probability.

    ...

    self.old_bid_prob = self.bid_prob

    rest = player.budget - current_bid - 1
    if player.nr_cities == 9 or rest >= 3*self.features.gray_city_price:
        self.bid_prob = 1
    else:
        self.bid_prob = 0

    bid = Prob_Strategy.get_bid(self, player, bundle, current_winner, \
                               current_bid)

    self.bid_prob = self.old_bid_prob

    return bid
```

```
def invest(self, player, players):
    '''grays = an empty list
    the old gray probability is set to be gray_prob
    Now list gray is filled with the numbers of gray cities
    for each player. The bad guys index (position in list)
    is the player who belongs to the maximum of gray cities
    from the list.

    If the player has as many cities as the bad guy,
    then the gray probability is set to zero.

    This is also done if the player has more than one patent.

    Method invest from class Prob_Strategy is called.
    Then the gray probability is set to the old gray probability
    ...

    grays = []
    self.old_gray_prob = self.gray_prob
    for player in players:
        grays.append(player.gray_cities)
    bad_guy_index = grays.index(max(grays))
    if player.gray_cities == players[bad_guy_index].gray_cities:
        self.gray_prob = 0.0
    if player.patents > 0:
        self.gray_prob = 0.0
    Prob_Strategy.invest(self, player, players)
    player.gray_prob = self.old_gray_prob
```

```
class Player:
```

```
    '''Class containing player methods'''

    def __init__(self, name, position, strategy):
        self.name = name
```

```
self.position      = position
self.current_track = 0
self.green_cities = 0
self.gray_cities  = 3
self.schools      = 0
self.patents      = 0
self.budget       = 10
self.wf           = 0
self.random       = Random(24)
self.board_pos    = 0
self.track        = 0
self.insurance    = 0
self.strategy     = strategy
self.dictator     = 0
self.strike       = 0
# obligations
self.obligations  = []
self.cex_obligation = 0
self.schools_ob   = 0
self.cities_ob    = 0
self.own_school_ob = 0

def nr_cities(self):
    '''Returns the overall number of cities as the sum of
    gray and green cities.
    '''

    return self.green_cities + self.gray_cities

def nr_schools(self):
    '''Returns the number of schools'''
    return self.schools

def reduce_cities(self, actual_cities):
    '''Function that reduces the cities of a player
    by the amount actual_cities (coming from the handle_
    event methods).
    From his gray cities he gets subtracted actual_cities.
    If his gray cities should then be less than zero, he
    gets subtracted this amount from his green cities and
    his gray cities are set to zero. If his green cities should
    then be less than 0, these are set to zero, too.
    '''
    self.gray_cities -= actual_cities
    if self.gray_cities < 0:
        self.green_cities += self.gray_cities
        self.gray_cities = 0
        if self.green_cities < 0:
            self.green_cities = 0

def change_wf(self, amount):
    '''Method that changes welfare units of a player.
    amount = argument depicting the change
    To welfare units amount is added. If afterwards the
    welfare units should be negative, they are set to
    be zero.
    '''

    self.wf += amount
    if self.wf < 0: self.wf = 0

def change_budget(self, amount):
    '''Method that changes the budget of a player.
    amount = argument depicting the change
    To the budget amount is added. If then budget should
    be negative, it is set to zero.
    '''
```

```
self.budget += amount
if self.budget < 0: self.budget = 0

def get_bid(self, bundle, current_winner, current_bid):
    '''Method to get a bid.
    Returns the result of method strategy.get_bid.
    '''

    return self.strategy.get_bid(self, bundle, current_winner, \
        current_bid)

def resolve_sanction(self, players):
    '''
    Method to resolve sanctions.
    nr_cities = random number from interval 1-6
    then the nr_cities is set to be the minimum of
    the random number and the actual gray cities.
    The gray cities are then reduced by br_cities.

    '''
    nr_cities = self.random.randint(1, 6)
    nr_cities = min(nr_cities, self.gray_cities)
    self.gray_cities -= nr_cities
    return 0

def resolve_own_school_ob(self, players):
    '''Method that resolves a school obligation
    If the player is obliged to buy a school, this
    method checks if he can do so and if yes lets
    him buy the school.

    If the number of schools is less than three
    and then the budget is greater than the school price
    then the players budget is decreased by the school price
    and one school is added to his schools.
    Otherwise he cannot buy a new school.

    Returns 0 in case the obligation is resolved, 1 otherwise.

    '''
    if self.schools < 3:
        if self.budget >= self.strategy.features.school_price:
            self.budget -= self.strategy.features.school_price
            self.schools += 1
            return 0
        else:
            return 1
    else:
        return 0

def resolve_strike_ob(self, players):
    '''Method that resolves a strike.
    If the player has more than zero gray cities
    then he gets substracted one city and added
    10 welfare units to his welfare.

    But if he has more than zero green cities then
    he gets substracted one green city and added
    10 welfare units to his welfare. For both
    cases 0 is returned, meaning that obligation
    is resolved.
    Otherwise 1 is returned.

    '''

    if self.gray_cities > 0:
```

```
        self.gray_cities -= 1
        self.wf += 10
        return 0
    else:
        if self.green_cities > 0:
            self.green_cities -= 1
            self.wf += 10
            return 0
    return 1

def resolve_cex_obligation(self, players):
    '''Method that resolves cex_obligation'''

    self.features = self.strategy.features
    # find out who the beneficiary is
    greens = []
    for p in players:
        greens.append(p.green_cities)
    beneficiary_index = greens.index(min(greens))
    beneficiary = players[beneficiary_index]
    if self.green_cities == beneficiary.green_cities \
        or beneficiary.green_cities == 9:
        self.cex_obligation = 0
        return 0

    while self.schools < 3 and self.patents < 1:
        if self.budget < self.features.school_price:
            break
        else:
            self.schools += 1
            self.change_budget(-self.features.school_price)

    if self.schools == 3:
        price = self.strategy.get_patent_price(players)
        while self.patents < 1:
            if self.budget < price:
                break
            else:
                patent_chance = self.features.patent_chance
                player.budget -= price
                if player.random.randint(1, patent_chance) == 1:
                    player.patents += 1

    if self.patents >= 1:
        while self.cex_obligation > 0 \
            and beneficiary.green_cities < 9:
            if self.budget < self.features.green_city_price:
                break
            else:
                self.change_budget( \
                    -self.features.green_city_price)
                beneficiary.green_cities += 1
                if beneficiary.nr_cities() > 9:
                    beneficiary.gray_cities -= 1
                if beneficiary.green_cities == 9:
                    self.cex_obligation = 0
                else:
                    self.cex_obligation -= 1

    return self.cex_obligation

def resolve_dev_ob(self, players):
    '''Method that resolves a development obligation'''

    self.features = self.strategy.features
    # find out who the beneficiary is
    greens = []
```

```

for p in players:
    greens.append(p.nr_cities())
beneficiary_index = greens.index(min(greens))
beneficiary = players[beneficiary_index]
if self.nr_cities() == beneficiary.nr_cities() \
    or beneficiary.green_cities == 9:
    self.schools_ob = 0
    self.cities_ob = 0
    return 0

school_price = self.features.school_price
city_price = self.features.green_city_price
patent_price = self.strategy.get_patent_price(players)

while self.schools_ob > 0:
    if self.budget < school_price:
        break
    else:
        self.change_budget(-school_price)
        beneficiary.schools += 1
        self.schools_ob -= 1

while self.schools < 3 and self.patents < 1:
    if self.budget < self.strategy.features.school_price:
        break
    else:
        self.schools += 1
        self.change_budget(-self.strategy.features.school_price)

if self.schools == 3:
    while self.patents < 1:
        if self.budget < patent_price:
            break
        else:
            patent_chance = self.features.patent_chance
            self.budget -= patent_price
            if self.random.randint(1, patent_chance) == 1:
                self.patents += 1

if self.patents >= 1:
    while self.cities_ob > 0 \
        and beneficiary.green_cities < 9:
        if self.budget < self.features.green_city_price:
            break
        else:
            self.change_budget( \
                -self.features.green_city_price)
            beneficiary.green_cities += 1
            if beneficiary.nr_cities() > 9:
                beneficiary.gray_cities -= 1
            if beneficiary.green_cities == 9:
                self.cities_ob = 0
            else:
                self.cities_ob -= 1

return (self.cities_ob + self.schools_ob)

def resolve_obligations(self, players):
    '''Method that resolves obligations (!).
    obc = value
    new_obs = empty list

```

For the obligations in the players list of obligations new_ob is set to be the value of ob from players. If new_ob is now greater than zero it is appended to list new_obs. Then obc is increased by one.

```
self.obligations is set to be the list new_obs. This list
contains the number of obligations to resolve.

NOT CLEAR
'''
obc = 0
new_obs = []
for ob in self.obligations:
    new_ob = ob(players)
    if new_ob > 0:
        new_obs.append(ob)
    obc += new_ob

self.obligations = new_obs
return obc

def make_deals(self, players):
    '''Method to make deals.
    If the obligations list is the empty
    list returns. otherwise
    returns result from make_deals from corresponding class (Prob_Strategy or Adaptive_Strategy).
    '''
    if self.obligations != []:
        return

    return self.strategy.make_deals(self, players)

def invest(self, players):
    '''Method that computes investments.
    If the obligations list is empty, and if result of
    resolve_obligations is larger than zero, returns.
    Otherwise calls method invest from corresponding class.
    '''
    if self.obligations != []:
        if self.resolve_obligations(players) > 0:
            return

    self.strategy.invest(self, players)

def make_offer(self, buyer, ask_for, offer):
    '''Method that computes an offer.
    Returns result of method make_offer from corresponding class.
    '''
    return self.strategy.make_offer(self, buyer, ask_for, offer)

def ask_for_patent(self, asker):
    '''Method to ask for a patent.
    Returns result of method ask_for_patent from corresponding class.
    '''
    return self.strategy.ask_for_patent(self, asker)

def notify_accept(self, sum):
    '''Method to compute a notification of acceptance.
    The number of patents are reduced by one and the budget
    is increased by sum.
    '''
    self.patents -= 1
    self.budget += sum

def move(self, board):
    '''Method computing the move of a player on the board.
    step is a random value from the interval [1,6]
    tracklen = length of the track
    newpos = players position on board plus step
    then board position is taken modulo to tracklength.
    this is the new board position which is returned.
```

```

'''
step = self.random.randint(1, 6)
tracklen = len(board.tracks[self.track])
newpos = step + self.board_pos
self.board_pos = newpos % tracklen
return self.board_pos

def __str__(self):
'''Printing method for information about this class.
Prints information about the players data.
'''
playerstr = "Player " + self.name + " with region " \
+ str(self.position + 1) + " has:\n" \
+ "\tgreen cities: " + str(self.green_cities) + "\n" \
+ "\tgray cities: " + str(self.gray_cities) + "\n" \
+ "\tschools: " + str(self.schools) + "\n" \
+ "\tpatents: " + str(self.patents) + "\n" \
+ "\tinsurance: " + str(self.insurance) + "\n" \
+ "\tbudget: " + str(self.budget) + "\n" \
+ "\twelfare: " + str(self.wf) + "\n" \
+ "Location on board: " + str(self.board_pos) + "\n"

return playerstr

class Cell:

def __init__(self, position=0, kind="regular", occupied=-1):
'''Initialisation of class Cell. kind is set to regular,
occupied to 1 and position to 0
'''
self.occupied = occupied
self.kind = kind
self.position = position

def __str__(self):
'''Printing method for class Cell'''
return "(" + str(self.occupied) + ", " + str(self.kind) \
+ ")"

class Board:
'''Class that creates the "game board"'''

def __init__(self, features = Features()):
'''Initialisation of class Board. Sets tracks to be an empty list.
Computes the tracklength of each track.
Creates a list containing information about the events that can
happen on the current track and the tracklength and the cells of
the track.
'''
self.tracks = []

for i in range(len(features.tracks)):
tracklen = features.tracks[i]["length"]
track = []
for j in range(tracklen):
track.append(Cell(j))
for j in features.tracks[i]["climate"]:
track[j].kind = "climate"
for j in features.tracks[i]["policy"]:
track[j].kind = "policy"
self.tracks.append(track)

def __str__(self):
'''Creates information about the game board'''
boardstr = ""
for i in range(len(self.tracks)):
boardstr += "Track " + str(i+1) + ":\n"

```

```

        boardstr += " "
        for cell in self.tracks[i]:
            boardstr += str(cell) + " "
        boardstr += "\n"

    return boardstr

if __name__ == "__main__":

    '''User input for number of players'''
    nr = int(raw_input("Number of players: "))

    '''User input for seed (some indicator for random stuff)'''
    seed = int(raw_input("Seed: "))

    '''assigns methods of class Features to object features'''
    features = Features()

    '''Creates a dictionary with r_probs (some kind of random probabilities).
    Then generates entry for this dictionary with floats depicting
    probabilities
    '''
    r_probs = {}
    r_probs["bid"] = 0.5
    r_probs["ask_bad_guy"] = 0.5
    r_probs["sanction"] = 0.5
    r_probs["ask_pat"] = 0.5
    r_probs["sell_pat"] = 0.5
    r_probs["scrap"] = 0.5
    r_probs["gray"] = 0.5
    r_probs["insurance"] = 0.5
    r_probs["school"] = 0.5
    r_probs["buy_pat"] = 0.5
    r_probs["green"] = 0.5
    r_probs["more_pat"] = 0.5

    '''An empty list players is created and then filled with the number of players
    set by the user when the game starts. Each player is called Randomx, where x is
    an integer. each player gets assigned Prob_Strategy with the given
    random probabilities (they don't change)'''
    players = []
    for i in range(nr - 1):
        players.append(Player("Random"+str(i+1), i, \
            Prob_Strategy(r_probs,features)))

    '''Here, a dictionary probs is created and then filled with entries depicting
    different probablities for different actions. '''
    probs = {}
    probs["bid"] = 0.8
    probs["ask_bad_guy"] = 0.9
    probs["sanction"] = 1.0
    probs["ask_pat"] = 0.8
    probs["sell_pat"] = 0.5
    probs["scrap"] = 0.9
    probs["gray"] = 1.0
    probs["insurance"] = 0.7
    probs["school"] = 0.9
    probs["buy_pat"] = 1.0
    probs["green"] = 1.0
    probs["more_pat"] = 0.2

    '''To the list of players another element, another player, is added. He is called
    LessRandom and his strategy is an Adaptive_Strategy which starts with the given

```

```
probabilities from dictionary probs. (these will change)'''
```

```
players.append(Player("LessRandom", nr - 1, Adaptive_Strategy(probs, \
    features)))
```

```
'''An object game is created by assigning to is the methods from class Game called with
arguments players and features. While over is equal to zero one round of the game is
called.
'''
```

```
'''
```

```
game = Game(players, features)
```

```
over = 0
```

```
while over == 0:
```

```
    over = game.one_round(nr)
```