Contents

Project Details

The following class descriptions may not be final or not the best way to implement the projects. You can modify the following class descriptions if you have a good justification.

Some projects are inherently persistent. Some has weaker relation with unique names. For first two phases, you can assume you know all unique names for object. In later phases you need to implement some basic flat directory based listing of objects so user can choose the object to work with. Also some basic authentication and user ownership can be added in last two phases optionally.

1. A Social Network for Book and Media Sharing

Item class

`Item` class have the following methods

Method	Description
`constructor(owner, type, title, uniqid, artist, genre, year)`	create a new collectible. If `uniqid` is not `None` the information for collectible is retrieved from Internet (from ISBN for example)
`borrowedreq(user)`	Add a borrow request for user. Requests are queued. The order in the queue is returned

Method	Description
`borrowedby(user, returndate='+2 weeks')`	user object borrows the item until `returndate`
`returned(location=None)`	borrowed item is returned
`comment(user, commenttext)`	a friend commented on the item
`listcomments()`	list of userid/comment pairs for the item
`rate(user, rating)`	a friend rated the item
`getrating()`	return average rating of the item
`locate(location)`	change/set location of the item (room/board/shelf) as text
`setstate(statetype, state)`	Change accessibility state of the item. `statetype` is either of `view, detail, borrow, comment, search` and `state` is either of `closed, everyone, friends, closfriends`. Users can make calls to only viewable items. `detail` implies `view`.
`search(user, searchtext, genre, year, forborrow=False)`	class method for searching an item. All matching item/user pairs are returned as a list. `searchtext` is searched in title and artist. If text has multiple words, all words should match independently. Matches are case insensitive, year can be a range. If `forborrow` is `True` only items that can be borrowed by user is returned.
`watch(user, watchmethod)`	User wants to watch the item for comments or borrow status change for immediate notification. Watching comments requre `detail` access, borrow status requires `borrow` access.
`view(user)`	The summary information for the item is listed.
`detail(user)`	The detailed information for the item is listed. If `user` is the owner, location is also listed, otherwise skipped
`announce(type, msg)`	Announce/promote the item to type (`friend, closfriend`) of owner, along with message. Notification is sent to all relevant users even they do not watch.
`delete()`	Item is deleted, all watchers will be notified, all borrow requests are dropped.

User class

Method	Description
`constructor(email, namesurname, password)`	A new user is created. User is in `notverified` state. User is send an email with verification number.
`verify(email, verification number)`	Static method. Verification number is checked agains stored one and user is enabled.
`changepassword(newpassword, oldpassword = None)`	User password is changed. If forgotten, oldpassword is set as `None` and a reminder with a uniq temporary password is sent. Next call will be with temporary password as `oldpassword` for changing the password.
`lookup(emaillist)`	A list of emails are given and looked in the user database, matching emails are returned.
`friend(email)`	Send a friendship request to the user with given email.
`setfriend(user, state)`	Change friend status with user, either `nofriend, friend, closefriend`
`listitems(user)`	Get item list of the user given as parameter. Only items with `view` permissions are listed.
`watch(user, watchmethod)`	Watch user for new items. `user` should be friend of current user object. One of `detail` or `borrow` permission is required for the items.

Other Issues

You can skip 5 methods in first phase, but you should complete in second phase. You need to implement at least one `watch()`. When a watch event occurs `watchmethod()` parameter is called with the user and item as parameters. All `whatch` calls can be followed by the same call with `watchmethod == None` to cancel them.

2. Online Bidding System

User class

Method	Description
`constructor(email, namesurname, password)`	A new user is created. User is in `notverified` state. User is send an email with verification number.
`verify(email, verification number)`	Static method. Verification number is checked agains stored one and user is enabled.

Method	Description
`changepassword(newpassword, oldpassword = None)`	User password is changed. If forgotten, oldpassword is set as `None` and a reminder with a uniq temporary password is sent. Next call will be with temporary password as `oldpassword` for changing the password.
`listitems(user, itemtype = None, state='all')`	Get item list of the user given as parameter. The `state` is either of `all, onhold, active, sold`
`watch(itemtype = None, watchmethod)`	Static method watch for new items. User will be notified by calling `watchmethod` for newly added items and staring bids for given item types. `None` stands for all types
`addbalance(amount)`	Users virtual balance in system is incremented/decremented by amount
`report()`	Get financial report for user including items sold, on sale, all expenses and income

SellItem class

Method	Description
`constructor(owner, title, itemtype, description, bidtype, starting, minbid = 1.0, image=None)`	create a new item for sale. `bidtype` is either of `increment, decrement, instantincrement`. `minbid` is the minimum bid unit.
`startauction(stopbid = None )`	Start the bidding for auctions. Auction will stop when `stopbid` is reached, if there is a bidder at the moment, he/she will win the auction automatically.
`bid(user, amount)`	User bids the `amount` for the item. Auction should be active and users should have a corressponding balance. Users balance is reserved by this amount. He/she cannot spend it until bid is complete. If there is a previous bid, it is updated.
`sell()`	Only owner can call this. Item is sold to the last bidder. Auction is closed
`view()`	Give the complete state of the item. Show auction data if available as well.
`watch(user, watchmethod)`	User is notified by calling `watchmethod` when the auction state of the item changes
`history()`	Detailed acivity log for item with creation, auction start, bids and final value

3. Online Configurable Board Game

Game configuration

Game configuration can be specified as a JSON file and it contains all information to execute the game logic.

Cells

Game consists of cells. A cell is either empty, or contains an action or contains an artifact. If it has an action, player arriving at the cell has to perform the action. If there is an artifact, player is given a chance to acquire the artifact. Artifacts also associated with some game logic explained below.

A cell information has the following fields:

`'cellno'`: integer id for the cell
`'description'`: A string content for cell
`'action | artifact'` (optional): actionval or artifactval .

Cell Actions

Cell actions are defined as the following JSON values:

Action	Description
`{'jump':relpos|abspos}`	player jumps to a relative or absolute position on board. Relative position is given as an integer as +2, -3. Absolute position is given as a string `'=34'`
`{'skip':turns}`	Player has to wait for the given turn times to play again
`{'drop':value}`	Players credit (i.e. money, health) is decremented by given value. If credit fails below 0, game is over for the player
`'drawcard'`	Draw a card from the deck of actions
`{'add':value}`	Players credit (i.e. money, health) is incremented by given value.
`{'pay':(uid, value)}`	Player pays the given credit to the `uid`

Artifacts

Game artifacts start at game board and can be picked by the player arriving the cell. They either disappear when picked by user or owned by user during the game. Picking an artifact may have a 0 or positive price. In addition to the artifact price, each artifact is associated with an optional cell action. When picked, the cell is updated to contain this new action.

In summary, an artifact value may have the following fields: `'name'`: name of

the artifact
'owned': boolean, True if owned, False when it disappears
'price': integer value for cost of picking up
'action' (optional): Cell action that replaces the content in the cell when artifact is picked.

Game setup

Game setup consists of following fields: 'name' : name of the game
dice' : max dice number at each turn.
'cells' : list of cell information s described above
'cycles' : False if board has no cycles. 'termination': 'finish' | 'firstbroke' | {'firstcollect': n} | {'round': n}
'cards': List of actions for cards (useful only if 'drawcard' action exists in the cells.
'credit': Default credit assigned to each user on start.

In 'finish' first one reaching the last cell wins (implies 'cycles':False. In 'firstbroke' the maximum credit user wins when any user gets below 0. In 'firstcollect', first player collecting n different artifacts (with different names) win. In 'round' first player cycled n times win (implies 'cycles':True). If cycles is a positive integer amount, each player completing a round earns this amount.

Sample simple game

```
{'name': 'Python Master',
 'dice' :6,
 'cycles': True,
 'termination': {'round': 5},
 'cells' : [ {'cellno' : 0, 'description' : 'Start'},
    {'cellno' : 1, 'description': ''}
    {'cellno' : 2, 'description' : 'Bug', 'action': {'skip': 1} },
    {'cellno' : 3, 'description' : 'New version', 'action': {'jump': 3} },
    {'cellno' : 4, 'description' : 'Runtime error', 'action': {'jump': '=0'} },
    {'cellno' : 5, 'description' : ''},
    {'cellno' : 6, 'description' : 'New IDE',
        'artifact': {'name': 'ide', 'owned': False,
                     'price': 0, 'action': {'skip': 1}}},
    {'cellno' : 7, 'description' : ''}],
 'cards': []
}
```

Game Class

Method	Description
`constructor(config)`	create a new game instance from given JSON config
`join(player)`	Player id joins game
`ready(player)`	Player sets its state ready to start the game. When all joined users so far ready game starts. When first player marks himself/herself ready, no more players can join the game
`listgames()`	Static method for getting list of games (name, state) served by the server. Player may call the `join()` method on joinable games later.
`state()`	Return full game state, the players position, cell descriptions, next turn etc.
`next(player)`	Player calls this for rolling the dice, and drawing a card. Returns the state change for the user with its description
`pick(player, pickbool)`	Player calls this for responding a `pick` choice.

Player Class

Method	Description
`constructor(id, nickname)`	create a player for new user
`join(game)`	calls `game.join(self)` if user is no in a game
`ready()`	calls `game.ready(self)`
`turn(type)`	notification method when next turn belongs to the player. `type` is `roll` at first notification. Player responds with a `next()` call. If `type` is `drawcard`, user responds again with a `next()` call. If it is `choice` with artifact description, user responds with a `pick(self, True)` or `pick(self, False)`.

Other Issues

Game starts as soon as all joined players acknowledge ready. Game sends notifications to players with `turn()` calls and expects them to respond. As they respond, it takes the action and send user the new state back.

For the first phase, you are expected to implement the game engine without artifacts and card picking. Full functionality is expected at the end of phase two.

4. Chain Reaction Machine Editor/Simulator

This project is highly dependent on **pymunk** library and most functionality limited by it. A simple editor with limited capacity for subset of **pymunk** is expected. Note that the objects defined below may need interaction of multiple **pymunk** objects.

Shape class

Following shape types are supported. Shapes are given realistic weight and size values in real life: * Balls (circles): `Bowling`, `Tennis`, `Marble` * Blocks (rectangles) : `Domino`, `Book` * Segments (segments) : `Fixed` , `Rotating` * Triggers (complex objects)

Segments are used to form paths for moving objects for sliding and rotating over. Usually they are `Fixed`, static segments. For creating chain reactions, `Rotating` segments are fixed at a middle point only and make rotating movement over it (like seesaws in kindergardens).

Triggers are complex objects consisting of multiple shapes to trigger reactions. For example a mechanism that releases a spring when touched from side so that spring pushes another segment when it is released. Also you can implement it programatically, when a subobject in trigger moves (get this information from simulation), you change velocity of another object. You can design your own trigger or triggers. Triggers are usually one shot. The rest is up to your imagination.

`Shape` is a base class of all such shapes and the concrete classes derived from it. Each shape will have a different set of calls and corresspondance in `pymunk`.

Connector class

Shapes can be connected (constrained) by using a set of connectors (joints in `pymunk`:

`Line`: A fixed length line connecting objects. (`PinJoint`) `Spring`: A spring connecting two bodies.(`DampedString`) `Rope`: A simple rope connecting object.

`Rope` does not exists in `pymunk` but you can have 4-5 bodies along the line connecting the objects and constraint them pairwise with `PinJoint`s in sequence.

`Connector` class is a base class of all connectors. It gets two shapes in its constructor.

Board class

Each game board will contain a `pymunk space` for 2D simulation. All objects are added/removed/edited on the board.

Method	Description
`constructor(x,y, name)`	A new board with x y dimensions are created
`addShape(shape, offset)`	Add new shape to board
`removeShape(shape, offset)`	Rmove shape from board
`moveShape(shape, offset)`	Move shape to a new location on board
`connect(connectorcls, shape1, shape2)`	Create a new connector with given class, return the `Connector` object.
`disconnect(connector)`	Remove the `Connector` from the board
`pick(x, y)`	get the list of objects containing point `(x, y)`. Axis aligned bounding box intersection is sufficient.
`attach(user)`	User attaches to the board. User is notified by all changes made by any user. When simulation started by any user, other attached users will observe it
`detach(user)`	User detaches from the board
`list()`	Static method to list all boards in the system (with their names). Users can attach to any board he/she likes
`start()`	Starts the simulation
`save(file)`	Saves the current board on given file name
`load(file, name)`	Static method to load a file into a newly constructed board
`state(update)`	Get the list of shapes with their current location and orientation in a simulation step. If boolean `update` is `True` only items changed since the last simulation step are returned

User class

Method	Description
`constructor(name)`	Constructor for user
`notify(state)`	User is sent the current state of the editor or simulation throught this method

Other issues

In first phase you are expected to implement couple of objects (a ball, a box, segment and a connector) and establish a connection with `pymunk`. Full functionality is expected at phase 2.