# Prosperville Code Documentation

Table of Content

## 1. Codebase Structure

Prosperville.ipynb is the main file of the game. It strings together all GUI elements and backend logic, and provides a unified entry point for the game.

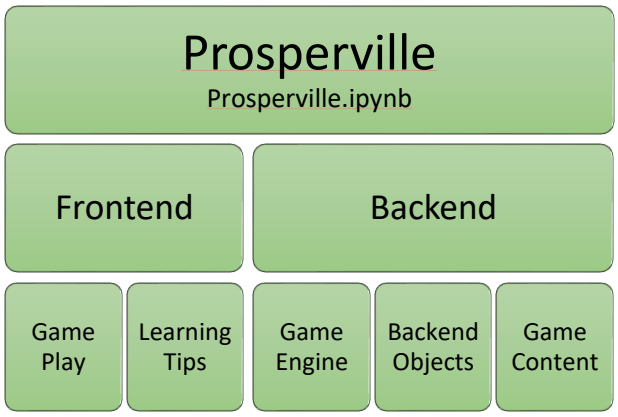


*Figure 1 Game logic structure*

The frontend is mostly implemented with ipywidgets package. Frontend code is organized in various files in the "gui" folder. We designed the front end so that it has two distinct features: 1) the usual game play that interacts between user inputs and the backend logic, and 2) a display area that shows a set of 3 learning tips that explain concepts related to better money habits.

The backend is coded in the "backend" folder. It is designed to be independent of the frontend implementation. The backend can be logically broken down to three components: the game engine that controls game progression and keeps scores, the backend objects that calculate the impact of user choices, and the game content design component that defines the stages and events of the game.

### 1.1. Frontend Files

The main game play interface is divided into four areas: header (gui/header.py), footer (gui/footer.py), side bar (gui/sidebar.py) and playground (gui/playground.py). Figure 2 below illustrates where the first three components are in the game graphic user interface. The area that is not annonated in the figure is the playground.

The playground displays a GridBox widget that has four child UI elements. These child UI elements occupy the same area in the playground. Only one of the four is displayed at any time of the game. These other four elements are:

- lifestage.py: the main game play area
- knowledge.py: knwoledge tip display area
- dashboard.py: dashboard display area. This area shows play chocies and their bi-monthly scores.
- msgbox.py: a message display area. This is area is only rendered right before the game ends to send a message.

Within lifestage.py, we implemented two ipywidget widgets to represent an event display area and each one of the options that come with an event. In Figure 2, each box with a yellow "Select" button is an option. It is implemented in gui/uiOption.py. uiLifeEvent object coded in gui/uiLifeEvent.py creates nesting uiOption objects and houses them within itself.

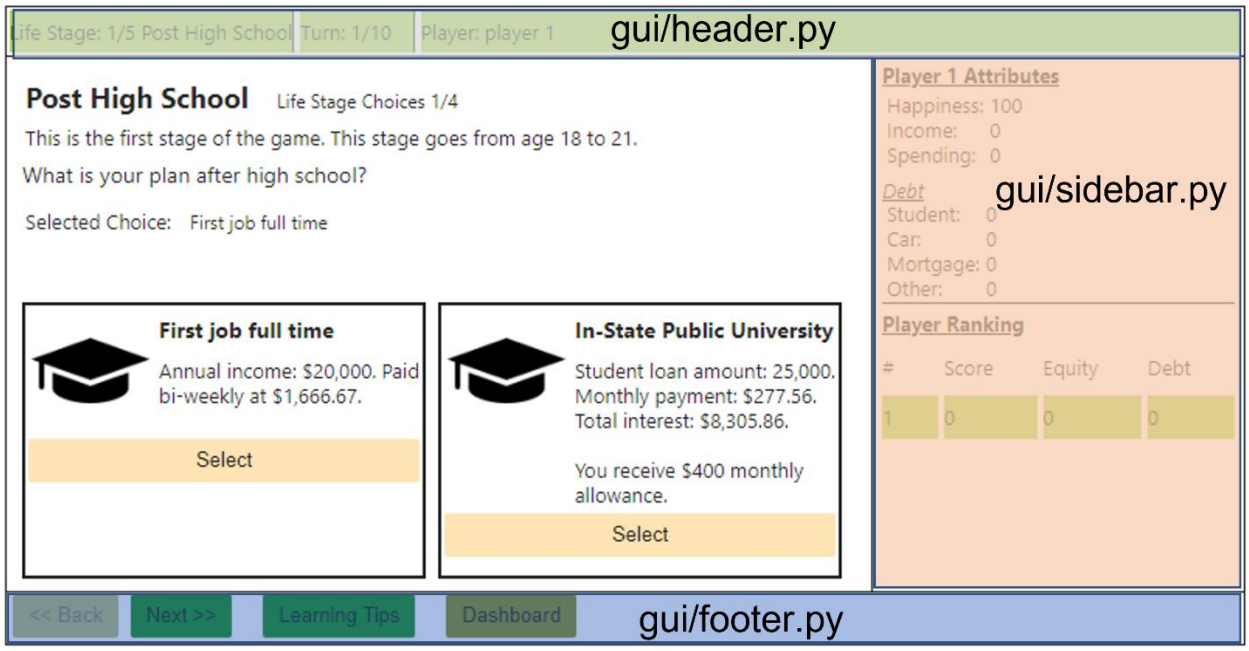


*Figure 2 GUI implementation breakdown*

## 1.2. Backend Files

The backend consists of game engine, backend objects and content design logic.

### *1.2.1. Content design logic*

The content design logic is the backbone of the game. It defines the events and stages, creates master game progression table, and instantiates backend objects. The game progression table defines how the game should move from one event to another.

The content design logic subcomponent consists of the following files:

- backend/gameitems.py: this is the main file of the content design logic and the entry point of the subcomponent's content for the game engine. The file replies on stage and event definitions to create game progression table and various mapping tables for easy access of the game. This file also contains light sanity checks to ascertain stages and events are designed appropriately.
- backend/design/stagedef.py: defines life stages of the game. Events in each stage are referenced by event names, not the event definition objects themselves.
- backend/design/eventdef.py: defines the events. Some events may have options for a player to choose. The options are also defined in this file.

### *1.2.2. Backend objects*

The impact each event or option has on a player's finance is carried out through backend objects. These objects are created by backend/gameitems.py based on the event definitions in backend/design/eventdef.py. Most of the backend objects simulate financial impacts and have easy to understand representations of financial object such as loans and assets.

The following files define the backend objects:

- expense.py: defines the Expense class

- income.py: defines both Salary and Asset classes.
- loan.py: defines the Loan class
- othrbkendobj.py: defines 1) the base class for all backend objects 2) the HappinessAdjustmentRatio class. This is the class that represents a direct change on happiness value outside of the relationship happiness has with wealth and spending. For more information on happiness calculation, please refer to section 2.3.

#### *1.2.3. Game Engine*

Game engine mostly concerns itself with game progression and score keeping. It has two files:

- prosperville.py: this file defines the main game logic in Prosperville class. This is the main backend class. All other elements in the backend can be accessed via this class. This is also the interface via which the front end communicates with the backend. Prosperville class mainly implements game progression and random event drawing logic.
- player.py: this file defines the Player class. This class keeps score and synthesizes the impact of backend objects / event options into player finances and happiness.

An event option may have multiple backend objects attached to it. When a player selects an option, all its backend objects are attached to that player's corresponding Player class instance. The simulation goes through all these backend objects and assesses the relevant finance information from the backend object's schedule table one object at a time. The backend object has all its schedule simulated upon the creation of the object during the first loading of backend.gameitems.py file. The simulation then combines all finances from each object's schedule table to arrive at a player's finance attributes. These attributes are then made into the calculation of happiness value.

## 2. Backend Logic

### 2.1. Game Progression Units

From a player's perspective, the game consists of stages and turns. Within each turn, there are events. Each random event is its own turn. A player is allowed to make modifications to their choices for all events within a single turn. For more details on stages and turns, please refer to the game rules document.

The game is designed that each player makes one choice for one event at a time. Thus, a more granular unit of game progression is needed to properly describe the game progression. A step represents the game progression specifically tied to an event. This is the single most granular unit of game progression for a single player game.

For multiplayer games, the game progression is controlled by both step and which player is at play. Within each turn, one player makes all their event choices before the next player can do so.

We implemented a game progression table, referred to as step table in the source code, to clearly describe the game progression. To view this table, in the Prosperville.ipynb, a user can execute the following code in a new cell at the end of the file:

```
import pandas as pd
pd.DataFrame(pv.crntGame.step_table)
```

To demonstrate the relationship between steps and turns, below is the first few rows of the step table.

*Table 1 First few rows of the step table with some columns omitted*

| | stage | turn | stage_name | event_name | period_first | period_sim_last | period_last | is_random_event_step |
|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 0 | post_hi_schl | stg1_college | 0 | 46 | 95 | False |
| **1** | 0 | 0 | post_hi_schl | stg1_car | 0 | 46 | 95 | False |
| **2** | 0 | 0 | post_hi_schl | stg1_lodging | 0 | 46 | 95 | False |
| **3** | 0 | 0 | post_hi_schl | stg1_part_time_job | 0 | 46 | 95 | False |
| **4** | 0 | 1 | post_hi_schl | | 47 | 95 | 95 | True |
| **5** | 1 | 2 | young_adlt | stg2_firstjob | 96 | 274 | 455 | False |
| **6** | 1 | 2 | young_adlt | stg2_firsthouse | 96 | 274 | 455 | False |

The table is implemented as a dictionary of lists with each key-list pair representing the column name and column values. In Table 1 above, the first untitled column is the step index. It is also the index of each list in the dictionary. The stage_name and event_name columns refer to the names of the stages and events that are defined in the content design logic (section 1.2.1).

Table 1 shows that stage 0 has 5 steps (rows 0-4) and 2 turns (turn = 0 and 1). All players can go back and forth on their selections of these events in turn 0. Once the game progresses to turn 1, all choices made for turn 0 are no longer editable.

Step 4 in Stage 0 and Turn 1 does not have an event_name value in Table 1. This is because the table is pulled at the beginning of the game when stage=0, turn=0 and step=0. Because Step 4 is a random event step (see the last column in Table 1), the event simply has not been drawn by the game engine. Thus the event_name value for the step is empty. This table is updated when the random event is revealed.

### *2.1.1. Simulation Period*

Although Step concerns with user interaction aspects of game progression, the most granular game progression unit is Simulation Period. We may refer to it as simply "period" for short at times. A simulation period is a unit of time over which the simulation calculates player finances. The default is set to bi-monthly in backend/design/__init__.py:

```
# defines number of simulation periods per month
NPeriodsPerMonth = 2
```

In Table 1, there are three columns that record various periods for a step. period_first defines the starting period of a step. period_sim_last shows the last period of a step.

Each row in the schedule table of a backend object corresponds to a simulation period.

### 2.2. Option Backend Definition Manifest

This section details how the key-value pairs saved in the backend field of backend.design.eventdef.stctOption are used to define various behaviors of the backend objects.

The backend field for stctOption is designed to be a dictionary although no safety check is implemented to enforce this rule in backend/gameitems.py. For all backend objects, the backend's BackendObjectBase class implementation dictates that all backend field dictionaries should have the following keys:

- type: type of the backend object. This is used to quickly check what type of object it is. Each class has its own value. FFor simplicity, we decided to not use isinstance function.
- amount: the main amount of an object. For Loan, this is the initial principal. For Expense, this is the expense amount. For Salary, this is the total annual pay. For Asset, this is the initial asset value. For HanppinessAdjustmentRatio, this is the adjustment ratio itself.
- title: this is mostly used in debugger to differentiate one object from another.
- start_period: the starting period at which the backend object takes effect on the player who has this object.
- is_happiness_spending: a Boolean that indicates if the object should be considered as the spending that affects happiness. This field is only relevant to expense and loan objects.
- category: this unutilized value is designed for the aggregation of backend objects' impact by categories such as car, house, etc.
- pay_freq_n_periods: defines the payment frequency. For every pay_freq_n_periods periods, a payment of loan or expense, a paycheck of Salary, one time appreciation of asset is applied, depending on the backend object type.
- term, term_unit: these two define how long the object should take effect on a player. The term defines the length while the term unit defines the time unit of the term value. Term unit accepts these values: "yr", "mth", "prd". For relevant info, see implementation of term_2_period function in backend/_shared.py.

Additionally, different backend objects also require unique key-value pairs in the backend field of the option definition.

For Expense objects or backend['type']=='expense':

- annual_rate: this is the annual rate at which the expense amount increases. This allows us to model inflation. This is converted to the proper periodic value per the following two values.
- rate_freq: once every rate_freq number of periods, the expense increases.
- rate_freq_unit: the unit at which rate_freq is quoted: "yr", "mth", "prd"

For Salary objects or backend['type']=='salary':

- amt_quote_term: this defines over what period is the amount quoted: "annual", "one-time". This allows us to use the Salary class to define one-time payment events such as winning lottery.

For relevant info, please see the implementation of periodic_amount function in backend/_shared.py file.

For Asset objects or backend['type']=='asset':

- value_cap: this is the total asset value that the asset's value cannot exceed. This is implemented to contain the compounding growth applied to house values.
- annual_rate: the annual rate at which the asset value increases
- recurring_n_periods: for every recurring_n_periods number of periods, the initial amount value of the object is added to its present value. This is used to implement periodic investment.
- recurring_end_period: at which period is the recurring investment defined by the above stops.

For Loan objects or backend['type']=='loan':

- annual_rate: the annual interest rate of the loan. Note the period interval over which interest rate is applied is defined by pay_freq_n_periods (utilized by the base class BackendObjectBase).

### 2.3. Happiness value and Score

The winner of a game is determined by the ranking of player scores. The score is the average happiness value over all simulation periods. The happiness value is based on a formula that takes into account a player's wealth and their qualifying spending. Happiness is affected by wealth (wealth effect) and spending (leisure effect).

**Wealth effect**: We modeled the happiness such that the wealthier a player is, the happier the player becomes, until the wealth reaches a level close to $1.5 million.

**Leisure effect**: In general, the more quality time / leisure a person consumes, the happier they are. In this game, we approximate leisure by qualifying spending. We assume that the more a player spends on qualifying spending items, the more leisure they directly or indirectly consume. This effect is generally maxed out when spending exceeds $4000 per month. The qualifying status of a loan payment or spending item is defined in is_happiness_spending of the option definition (see section 2.2) or correspondingly the backend object's field of the same name.

Note that the more one spends the less they are able to accumulate for wealth. There is a delicate balance between spending and saving (-> increasing wealth) to maximize the total happiness.

**Direct Impact**: A third way happiness value can be influenced is a direct percentage adjustment of the raw happiness value. This allows us to model the adverse effect high stress job has on happiness even though the job may pay above average.

The following equation defines the happiness value:

$$f^H\left(W_{i,t}, A^S_{i,t}\right) = \frac{1}{2}\left(A^{HW}_{i,t} + A^{HI}_{i,t}\right) H_{i,t}$$

where $A^S_{i,t}$ is monthly qualifying spending amount for player $i$ at simulation period $t$; $W_{i,t}$ represents the wealth of a player; $H_{i,t}$ is an adjustment factor. HappinessAdjustmentRatio backend objects can influence $H_{i,t}$ while all other backend objects change $W_{i,t}$ and / or $A^S_{i,t}$.

The wealth effect is formulated below:

$$A_{i,t}^{HW} = \left(1 + e^{-W_{i,t}/W_{i,t}^{M}+1}\right)^{-1}$$

where $W_{i,t}^{M} = 167000$ is the median wealth. Figure 3 below plots $A_{i,t}^{HW}$ as a function of wealth.

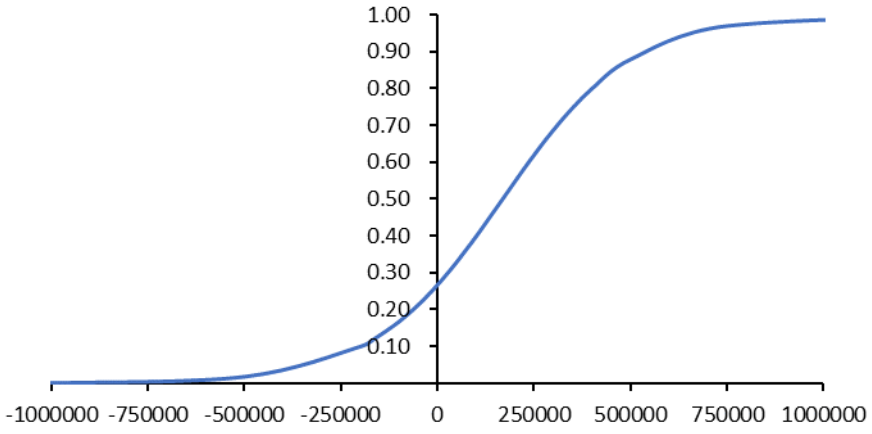


*Figure 3 Happines as a function of wealth*

The leisure effect is modeled as:

$$A_{i,t}^{HI} = \tanh\left(A_{i,t}^{S}/2A_{M}^{S}\right)$$

where $A_{M}^{S} = 2133$ represents the monthly median spending. Figure 4 below plots $A_{i,t}^{HI}$ as a function of spending.

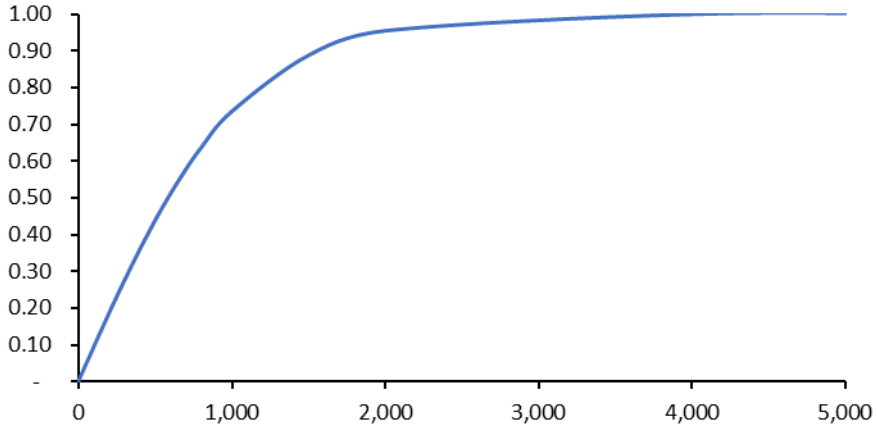


*Figure 4 Happiness as a function of spending*

### *2.3.1. Score initialization*

Because we allow players to set their initial cash levels at the start of the game, players may have different starting wealth from one game to another. To remove this difference, we normalize all starting game happiness to 100 by the initial happiness at the start of the game via the following calculation:

$$A_{i,t}^{H} = f^{H}\left(W_{i,t}, A_{i,t}^{S}\right)\frac{100}{f^{H}\left(W_{i,t=0}, A_{i,t=0}^{S}\right)}$$

Finally, player $i$'s score at simulation period $t$ is:

$$S_{i,t} = \frac{1}{t}\sum_{s=0}^{t} A_{i,s}^{H}$$

## 3. Code Improvement Suggestions

Now you have a general idea of how the code works, it is time to read the source code itself. To guide you through the source code further, we have the following improvement suggestions for you to keep in mind when you read the code. This would help you be more focused on your code reading.

These suggestions are progressively harder. So we suggest you start from the very first one.

### *3.1.1. Second house purchase*

We, cough(!) cough(!), intentionally left out a second house purchase event out of stage 3, even though the event is already defined in the event definition file. Can you put it back?

- Where do you think the stage definition is at in the code?
- Once you think you put the event back into the stage, can you verify the event is indeed added back by inspecting the step table?
- What do you think the house value would do to your overall wealth? Can you change different house values to see what the impact might be?

### *3.1.2. College lodging cost*

The game currently assumes the off-campus rent is too cheap compared to reality. Can you increase the amount?

- The Lodging choices show up in the third step of the first stage. You can locate the relevant information in the step table in Table 1 on page 5.
- Which part of the backend logic component should have this information?
- Can you verify your change by inspecting _score_table field content from the Player class?

### *3.1.3. Consecutive random events*

Although random events are randomly drawn, we do occasionally notice the same event may be drawn back to back in the Peak Career stage. Can you write new logic to avoid that?

- First identify where the logic of drawing random events is implemented.
- Then come up with a method. The simplest method is the rejection method: if a second random event in the same stage is the same as the previous event, we reject the second event and redraw for the second event.
- How do you verify your new logic is working given all results are supposed to be random?

### *3.1.4. Missing stage in the score table*

If you click the "Dashboard" button on the footer of the game, have you notice the score table has an empty "Stage" column? Can you figure out a way to add that information to the table?

- The game is guided through the step table implemented in the field crntGame.step_table. The table also has the information you needed to map a period to its stage.
- Can you identify where in the frontend logic the score table display is implemented? How about the backend equivalent?

- There are many ways you can implement this and there is no right or wrong way of doing it. But if we are doing it, we would look into a mapping table and perhaps the apply method in pandas package.

#### *3.1.5. Initial survey user interface*

Right now, the initial setting of the game is hard coded in a cell in Prosperville.ipynb file. Can you design a frontend that accepts user inputs of player names and gives a place to enter initial cash?

## 4. Appendix A: Source Code File Table

The following table summarizes the main function of each code file.

| File / Folder | Description and main components |
|---|---|
| prosperville folder | |
| > Prospervillel.ipynb | Main entry point file |
| > backend folder | Backend logic subfolder |
| > > __init_.py | Initialization file for the backend submodule |
| > > _shared.py | Contains shared objects across the backend module<br>➢ term_2_period: function that converts a term form event definition to number of periods.<br>➢ periodic_amount: function that c onverts an amount to its a value suitable for a single simulation period |
| > > design | Design subfolder |
| > > > __init__.py | Game design constant<br>➢ NPeriodsPerMonth defines the number of simulation period per month |
| > > > eventdef.py | Event definition<br>➢ stctEvent: namedtuple that defines the structure of event definition<br>➢ stctOption: namedtuple that defines the structure of option definition<br>➢ pvEvents: a list of event definitions |
| > > > stagedef.py | Stage definition<br>➢ stctStage: namedtuple that defines the structure of stage definitions |
| > > gameitems.py | Main file for the Game Design logic component |
| > > expense.py | Expense backend object<br>➢ Expense: class that represents an Expense item from an event |
| > > income.py | Income type backend objects<br>➢ Salary class that represents Salary provided by an event<br>➢ Asset class that represents Asset provided by an event |
| > > loan.py | Loan backend object<br>➢ Loan: class that represents a loan item from an event |
| > > othrbkendobj.py | Other backend objects<br>➢ BackendObjectBase: a base class for all backend objects. This class defines the common routines that are shared among all backend objects.<br>➢ HappinessAdjustmentRatio: class that represents a direct impact on happiness outside of the influence of wealth and spending. |
| > > player.py | Game engine player logic<br>➢ Player: class that reprsents a player (human or AI), stores player attributes and scores the player based on the attribute values |
| > > prosperville.py | Main game engine logic<br>➢ Prosperville: class that represents the backend logic of the game Prosperville<br>➢ crntGame variable: globally accessed among the Jupyter notebook, the backend modules, and GUI modules to represents the current running game |
| > gui | Frontend user interface folder |
| > > __init_.py | Empty, simply placed to make the gui folder a module for easy access of subfolder .py files |
| > > _shared.py | Common elements that are shared across files in the gui folder |

| File / Folder | Description and main components |
|---|---|
| > > dashboard.py | Dashboard UI display area<br>➢ tabDashBoard: Tab widget that displays the dashboard UI |
| > > footer.py | Footer UI display area<br>➢ tblFooterArea: GridBox widget that displays the footer content<br>➢ refresh_gui: function that refreshes the entire GUI content based on game progression |
| > > header.py | Header UI display area<br>➢ tblHeaderArea: GridBox widget that displays the header content<br>➢ refresh_header: function that refreshes the header content based on game progression |
| > > knowledge.py | Knowledge UI display area<br>➢ tabKnowlege: Tab widget that displays the content of knowledge tips |
| > > lifestage.py | Life Stage display area: main game play area<br>➢ tblLifeStagePage: GridBox widget that displays the content of the game<br>➢ refresh_lifestage: function that refreshes the game content based on game progression |
| > > msgbox.py | Displays the message area of the game<br>➢ show_message function that displays a message in the playground area and pauses the game. |
| > > playground.py | Defines GUI elements in the playground<br>➢ tblHeaderArea: GridBox widget that displays the playground content<br>➢ refresh_ refresh_playground: function that renders the playground based on all game attributes |
| > > sidebar.py | Defines GUI elements in the side bar area of the game<br>➢ tblSideBarArea: GridBox widget that displays the content of the side bar<br>➢ refresh_sidebar: function that refreshes the sidebar content based on game progression |
| > > uiLifeEvent.py | Implements uiLifeEvent class that represents a UI element for an event |
| > > uiOption.py | Implements uiOption class that represents a UI element for an event option |
| > css | Folder that contains a css file used to define GUI look |
| > res | Resource folder for images |