Overview

Cookie Clicker is a game built around a simulation in which your goal is to bake as many cookies as fast as possible. The main strategy component of the game is choosing how to allocate the cookies that you have produced to upgrade your ability to produce even more cookies faster. You can play Cookie Clicker here. Before you start work on this mini-project, we strongly recommend that you complete the Practice Activity, "The Case of the Greedy Boss", which is designed to walk you through the steps of building a simulation similar to Cookie Clicker.

In Cookie Clicker, you have many options for upgrading your ability to produce cookies. Originally, you can only produce cookies by clicking your mouse. However, you can use the cookies you earn to buy other methods of producing cookies (Grandmas, farms, factories, etc.). Each production method increases the number of "cookies per second" (CPS) you produce. Further, each time you buy one of the production methods, its price goes up. So, you must carefully consider the cost and benefits of purchasing a production method, and the trade-offs change as the game goes on.

For this assignment, you will implement a simplified simulation of the Cookie Clicker game. You will implement different strategies and see how they fare over a given period of time. In our version of the game, there is no graphical interface and therefore no actual "clicking". Instead, you will start with a CPS of 1.0 and may start purchasing automatic production methods once you have enough cookies to do so. You will implement both the simulation engine for the game and your own strategies for selecting what production methods to buy.

We have provided the following template that contains an outline of the code you will write, including a `ClickerState` class, which will keep track of the state of the simulation, and a `simulate_clicker` function, which will run the simulation. The signature (name and parameters) of the functions, classes, and methods in this file must remain unchanged, but you may add any additional functions, methods, or other code that you need to.

Provided Code

We have provided a `BuildInfo` class for you to use. This class keeps track of the cost (in cookies) and value (in CPS) of each item (production method) that you can buy. When you create a new `BuildInfo` object, it is initialized by default with the default parameters for our game. Keep in mind that if you pass a `BuildInfo` object around in your program and it is modified anywhere, then you will see the changes everywhere. If you do not want that, we have provided a `clone` method that will create an identical copy of the object. You can see a description of the class and its methods here. It has methods to allow you to find out what all of the items' names are, the cost per CPS of each item, and to update the cost of a particular item appropriately.

We have also provided a `run` function to run your simulator. Note that the `run` function simply calls `run_strategy`, which runs the simulator once with a given strategy. You can add more calls to `run_strategy` inside of `run` once you develop your own strategies. The `run_strategy` function runs the simulation, prints out the final state of the game after the given time and then plots the total number of cookies over time. You can replace our `SimplePlot` plotting code with your own plotting code if you wish to use IDLE or another Python IDE. If you do leave our plots in, make sure that your web browser is configured to allow popup windows from CodeSkulptor. The plots will help you understand how the number of cookies grow over time, which is the point of this week in the class, so we recommend you do look at the plots, even if you comment them out while you are debugging.

We have provided a simple strategy, called `strategy_cursor_broken`. Note the signature of the function: `strategy_cursor_broken(cookies, cps, history, time_left, build_info)`. All strategy functions take the current number of cookies, the current CPS, the history of purchases in the simulation, the amount of time left in the simulation, and a `BuildInfo` object (even if they don't use these parameters). You'll note that this simple strategy just always picks `"Cursor"` no matter what the state of the game is. This is obviously not a good strategy (and it violates the requirements of a strategy function given below), but rather is a placeholder so you can see the signature of a strategy function looks like and use it while you are debugging other parts of your code.

Testing your mini-project

As always, testing is a critical part of the process of building your mini-project. Remember you should be testing each method as you write it. Don't try to implement all of the methods and then test. You will have lots of errors that all interact in strange ways that make your program very hard to debug.

- As always, we suggest that you build your own collection of tests using the `poc_simpletest` module as you implement the `ClickerState` class. We also strongly suggest that you add a `print_history` method that nicely prints the history of the game. Then, you can run short simulations, print the history, and confirm that things are working as you would expect.
- Finally, submit your code (with the call to **`run`** commented out) to this Owltest page. This page will automatically test your mini-project. It will run faster if you comment out the call to **`run`** before submitting. Note that trying to debug your mini-project using the tests in OwlTest can be very tedious since they are slow and give limited feedback. Instead, we *strongly*suggest that you first test your program using your own test suite. Programs that pass these tests are much more likely to pass the OwlTest tests.**NOTE:** OwlTest will test your code in isolation. For example, your strategy functions will be tested with *our* simulation functions. Your simulation function will be tested using *our* **`ClickerState`** class. And so on. You therefore must respect the interfaces and specifications of each element of this project. This both enables us to test the pieces of your project in isolation and is good development practice to follow specifications and keep your code independent.

Remember that OwlTest uses Pylint to check that you have followed the coding style guidelines for this class. Deviations from these style guidelines will result in deductions from your final score. Please read the feedback from Pylint closely. If you have questions, feel free to consult this page and the class forums.

When you are ready to submit your code to be graded formally, submit your code to the CourseraTest page for this mini-project that is linked on the main assignment page.

Phase One

You should first implement the **`ClickerState`** class. This class will keep track of the state of the game during a simulation. This various fields in this class should roughly correspond to the local variables used in implementing the function **`greedy_boss`** in the Practice Activity. (Cookies correspond to money and upgrades to CPS correspond to bribes.) By encapsulating the game state in this class, the logic for running a simulation of the game will be greatly simplified. The **`ClickerState`** class must keep track of four things:

1. The total number of cookies produced throughout the entire game (this should be initialized to `0.0`).
2. The current number of cookies you have (this should be initialized to `0.0`).
3. The current time (in seconds) of the game (this should be initialized to `0.0`).
4. The current CPS (this should be initialized to `1.0`).

Note that you should use **`float`**s to keep track of all state properties. You will have fractional values for cookies and CPS throughout.

During a simulation, upgrades are only allowed at an integral number of seconds as required in Cookie Clicker. However, the CPS value is a floating point number. In addition to this information, your **`ClickerState`** class must also keep track of the history of the game. We will track the history as a list of tuples. Each tuple in the list will contain 4 values: a time, an item that was bought at that time (or **`None`**), the cost of the item, and the total number of cookies produced by that time. This history list should therefore be initialized as `[(0.0, None, 0.0, 0.0)]`.

The methods of the **`ClickerState`** class interact with this state as follows:

- **`__str__`**: This method should return the state (possibly without the history list) as a string in a human readable format. This is primarily to help you develop and debug your program. It will also be used by OwlTest in error messages to show you the state of your **`ClickerState`** object after you fail a test.

- **`get_cookies`**, **`get_cps`**, **`get_time`**, **`get_history`**: These methods should simply return the current number of cookies, the current CPS, the current time, and the history, respectively. Note that **`get_history`** should return a copy of the history list so that you are not returning a reference to an internal data structure. This will prevent a broken strategy function from inadvertently messing up the history, for instance.
- **`time_until`**: This method should return the number of seconds you must wait until you will have the given number of cookies. Remember that you cannot wait for fractional seconds, so while you should return a **`float`** it should not have a fractional part.
- **`wait`**: This method should "wait" for the given amount of time. This means you should appropriately increase the time, the current number of cookies, and the total number of cookies.
- **`buy_item`**: This method should "buy" the given item. This means you should appropriately adjust the current number of cookies, the CPS, and add an entry into the history.

If a method is passed an argument that is invalid (such as an attempt to buy an item for which you do not have enough cookies), you should just return from the method without doing anything.

Phase Two

Once you have a complete **`ClickerState`** class, you are ready to implement **`simulate_clicker`**. The **`simulate_clicker`** function should take a **`BuildInfo`** class, the number of seconds to run the simulation for, and a strategy function. Note that **`simulate_clicker`** is a higher-order function: it takes a strategy function as an argument!

The first thing you should do in this function is to make a clone of the **`build_info`** object and create a new **`ClickerState`** object. The function should then loop (in the same manner in the function **`greedy_boss`** from the Practice Activity) until the time in the **`ClickerState`** object reaches the duration of the simulation.

For each iteration of the loop, your **`simulate_clicker`** function should do the following things:

1. Check the current time and break out of the loop if the duration has been passed.
2. Call the strategy function with the appropriate arguments to determine which item to purchase next. If the strategy function returns **`None`**, you should break out of the loop, as that means no more items will be purchased.
3. Determine how much time must elapse until it is possible to purchase the item. If you would have to wait past the duration of the simulation to purchase the item, you should end the simulation.
4. Wait until that time.
5. Buy the item.
6. Update the build information.

Note that the **`ClickerState`** class already implements methods that will greatly simplify steps 3–6. Use them!

Also note that the time is only incremented when you wait until you can purchase the item selected by the strategy function. You should not try to create a loop that ticks through each second of the simulation. This will not work effectively because it will be incredibly slow. Most times during the simulation you are just waiting until you have accrued enough cookies (this is the boring part of the actual Cookie Clicker game), so the process described above just skips over those times.

For correctness, you should not allow the simulation to run past the duration. This means that you should not allow an item to be purchased if you would have to wait until after the duration of the simulation to have enough cookies. Further, after you have exited the loop, if there is time left, you should allow cookies to accumulate for the remainder of the time left. Note that you *should* allow the purchase of items at the final duration time. Also, if you have enough cookies, it is possible to purchase multiple items at the same time step. (Note that this differs from the actual Cookie Clicker game, where it is not possible to buy multiple items at the same time.) This is most likely to happen exactly at the final duration time, when a strategy might choose to buy as many items as it can, given that there is no more time left.

Finally, you should return the **`ClickerState`** object that contains the state of the game.

If you have implemented things correctly, with the provided `strategy_cursor_broken` function, the given `SIM_TIME`, and default `BuildInfo`, the final state of the game should be:

- Time: 10000000000.0
- Current Cookies: 6965195661.5
- CPS: 16.1
- Total Cookies: 153308849166.0

Phase Three

Finally, you should implement some strategies to select items for you game. You are required to implement the following strategies:

1. `strategy_cheap`: this strategy should always select the cheapest item that you can afford in the time left.
2. `strategy_expensive`: this strategy should always select the most expensive item you can afford in the time left.
3. `strategy_best`: this is the best strategy that you can come up with.

If there is not enough time left for you to buy any more items (or your strategy chooses not to), your strategy function should return `None`, otherwise your strategy functions should return a valid name of an item as a string. As described above, if your strategy function returns `None` this should cause your `simulate_clicker` function to exit the loop and finish the simulation.

For `strategy_best`, you will be graded on how many total cookies you are able to earn with the default `SIM_TIME` and `BuildInfo`. To receive full credit, you must get at least 1.30×10^{18} total cookies. In addition, you may implement as many other strategies as you like. We will have a forum thread dedicated to showing off your favorite/best strategies!

Mark as completed