CS 1110 Spring 2018, Assignment 5*

1 Motivation: Uno

For this assignment we will exercise a few different concepts: while loops, classes, and sub-classes. We do this by implementing a simplified version of the game of Uno.

In Uno, players take turns attempting to play a card from their hand and add it to the top of a common pile of cards. Cards can only be added to the pile if they match at least either the suit or the rank of the top card on the pile or if the card being added is a wild card. The first player who has gotten rid of all their cards wins!

Because we can't afford to license Uno for this class, we're going to implement a slightly simplified version of the game based on standard playing cards. We'll treat certain cards as the special cards in a game of Uno based on their rank, following this guide:

- `10`s are Reverse cards, which reverse the order of play.
- `Jack`s are Skip cards, which skip over the next player.
- `Queen`s are Draw Two cards, which force the next player to draw 2 cards before the start of their turn. Unlike the traditional Uno rules, we'll allow that player to continue to play their turn like normal after taking their extra cards.
- `King`s are Wild cards, which can be played on top of any card. Unlike traditional Uno rules, the person who played the card does not get to choose the next suit. In our game, the Wild will set the suit to whatever suit the King card has.
- `Ace`s are Wild Draw Four cards, which has the same features of a Wild card while also forcing the next player to draw 4 extra cards before the start of their turn.

We have provided a skeleton of this game with 6 files. Search through all files, and you will find comments with the string `TODO`. These are areas that we have intentionally left blank and expect you to implement. To get a better idea of what these files do, it is highly recommended that you read the Structure of the Code section below.

Contents

*Authors: Anthony Poon, Victoria Litvinova.

2 Instructions

Download the A5 files from the course Assignments page. Complete the parts marked as `TODO` in the `a5_uno.py`, `a5_player.py`, and `a5_unocard.py` files. Please make sure you address every `TODO`. Turn in these three files. Your changes to these three files should not rely on modifications to the other files to work. There is an additional file that you may turn in named `optional_extensions.py`, but this isn't required, and you don't need to turn this in.

3 Structure of the Code

Because this assignment uses classes more deeply than previous assignments, it has a few more files than normal. Classes are one way to give your code a lot more organization, but that means that functionality is often split up into different places. To help you do your assignment, it is a good idea to understand the structure of the code so you know how all the pieces connect.

3.1 Major Parts and How They Interact

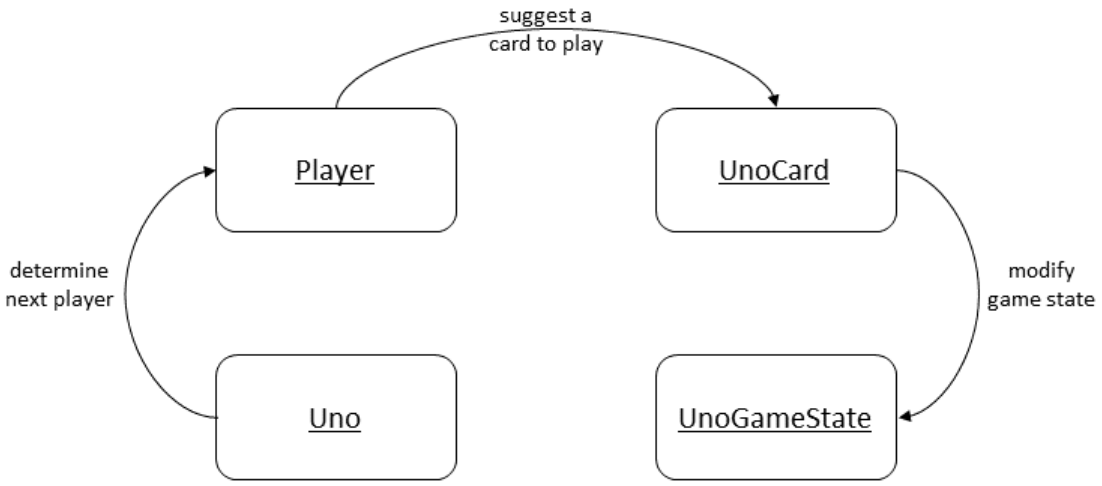
There are four major parts to the Uno game, represented in the diagram below.

The first and most important of these is the `Uno` class. This class is responsible for the flow of the game. In its `init` method, `Uno` will set up the game for a given list of players, create and shuffle the deck, and deal a starting hand to each player. When the game is being played, the `Uno` class will determine who the next player is, represented by an instance of the `Player` class.

The `Player` class encapsulates the player state, such as the hand the player currently holds. It also is responsible for implementing the player's actions. When the game is being played, the player must choose a card from its hand and suggest to play it. The player is only "suggesting" a card because the card may not be playable according to the rules of Uno. In our implementation, we give the player a limited number of attempts to suggest a playable card before they are forced to draw a card and skip their turn.

The `UnoCard` class has the responsibility of determining whether instances of this class can be played on top of another card, according to the rules of Uno. In addition, as certain cards have special actions or abilities, this class is responsible for performing those special actions by modifying the game state.

Lastly, the `UnoGameState` keeps track of the state of the game beyond individual players and cards. Imagine it as a sort of ledger that keeps track of who the next player is, whether that player needs to draw extra cards, and whether the play order has been reversed or not.

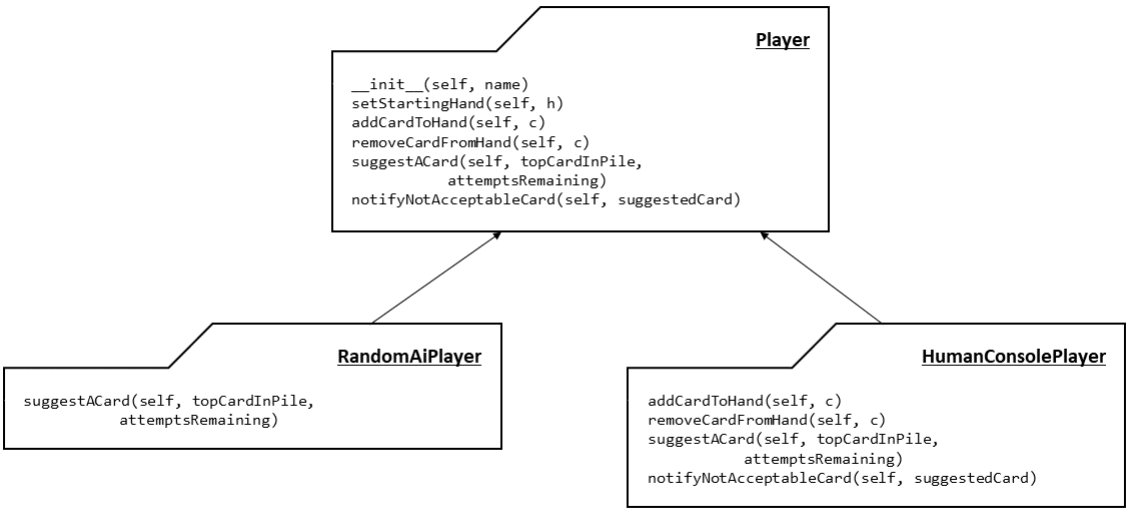


This cycle will continue until `Uno` decides that a player has won.

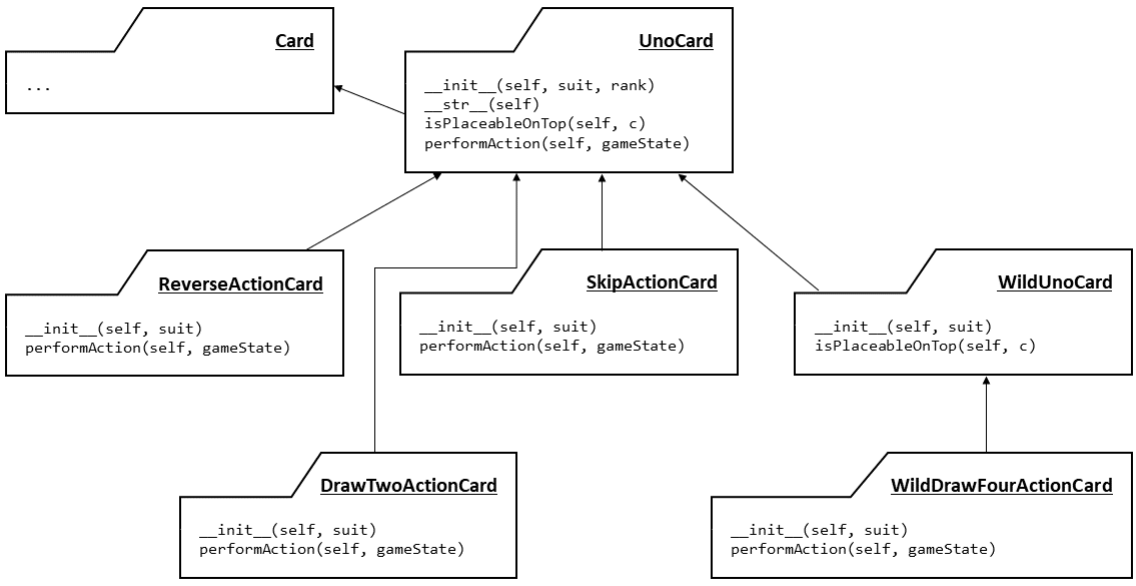
3.2 Class Structure of Player and UnoCard

To provide their functionality, `Player` and `UnoCard` both have several subclasses, some of which you'll help implement. Their layout is described in more detail below.

The following is a class diagram for `Player`. You'll notice that some methods are implemented mostly in the parent class, namely the methods that allow the Player to accept a new hand and manage it by adding and removing cards. There are two pre-made subclasses. The `RandomAiPlayer` and `HumanConsolePlayer` have different ways to determine what card to suggest, so both of these subclasses override the `suggestACard` method. If you look through the code, you'll notice that `HumanConsolePlayer` also overrides some other methods, mostly to add some nice `print` statements to give the human user some more insight into their player state.



The following is a class diagram for `UnoCard`.



`UnoCard` extends from the `Card` class that you've seen previously. This allows `UnoCard` to reuse some of `Card`'s functionality. Relevant to the game of Uno, `UnoCard` defines two additional methods: `isPlaceableOnTop` implements restrictions on which card can be played on top of another card. `performAction` allows special cards to modify

the game state. There are several subclasses of `UnoCard` to implement the functionality of the special cards. You'll notice `WildUnoCard`s also form their own subclass hierarchy under `UnoCard`. Subclasses allow us to share functionality between class definitions, but also change functionality where appropriate as classes become increasingly specialized.

3.3 Other Advice

We've provided `a5_unochecks.py` which provides some tests to help you check that your code is working successfully. This file uses a different framework than the Cornell Asserts package that we've been using before, but the underlying concepts are the same. You can run these tests by running `python a5_unochecks.py` from your command prompt or terminal.

Do not assume that your assignment is perfect simply because it passes all the tests provided in `a5_unochecks.py`! Pay close attention to the specifications of the methods you are implementing. We also encourage you to write additional tests as you think necessary to help you correctly implement your assignment. If you run into issues, adding debug print statements of your players' hands, the first card in the deck, the last card added to the pile, and the game state object in each player's turn can help you better understand how your game is progressing.

We also hope you find time to gain some useful insight into how to write and test object-oriented code by reading the other parts of the assignment.

4 Rules (all the same as for previous assignment)

There is no revise-and-resubmit for this or any subsequent assignment unless otherwise noted.

4.1 You need to re-group (so to speak) on CMS, regardless of prior groupings

You may work alone or with just one other person, who can be someone you've grouped with before in CS1110, or a different person.

If you are partnering, regardless of whether you were grouped in a previous assignment, the two of you must still form a new group *specifically for this assignment* on CMS before submitting.[1]

If your partnership dissolves, see the course Academic Integrity description about "group divorce" for what to do.

4.2 (Dis-)allowed collaborations and documentation requirements

Our policies are laid out in full on the course Academic Integrity page, but we re-state here **the main rules**: where "you" means you and, if there is one, your one CMS-registered group partner,

1. Never look at, access or possess any portion of another group's work in any form.
2. Never show or share any portion of your work in any form to anyone except a member of the course staff.
3. Never request solutions from outside sources; for example, on online services like StackOverflow.
4. DO specifically acknowledge by name all help you received, whether or not it was "legal" according to (1)-(3).

5 Code cleanup and submission checklist (all the same as before except the due date)

Before submitting, ensure your code obeys the following.[2]

- Lines are short enough (~80 characters) that horizontal scrolling is not necessary. (We've sometimes violated this for readability.)
- You have indented with spaces, not tabs (this is not an issue if using Komodo Edit).

[1] This links your submission "portals".

[2] One reason for these requirements is that they speed up the process of reading hundreds of files.

- You have removed any debugging `print` statements.
- You have removed all `pass` statements that were provided with the initial code.
- You have removed "instruction" comments, typically marked "`# TODO`".
- If you added any helper functions, these have good docstring specifications and you have put sufficient testing code for your functions.

Make sure the following are all true before you submit.

1. You (and your partner) have included your name(s) and NetID(s) in the header of all files.
2. You've also indicated in the file headers the names of any (non-staff) people whose help should be acknowledged.
3. The date in the header comments has been changed to when the files were last edited.
4. You have set your CMS notifications settings to receive email regarding grade changes, and regarding group invitations.

5.1 Due dates

1. If you are partnering: well before submission, follow the instructions in the "How to form a group" instructions on the course Assignments page. Both parties need to act on CMS in order for the grouping to take effect.
2. By 2pm on Wed May 9, submit whatever you have done at that point to CMS, following steps 1-3 in the "Updating, verifying, and documenting assignment submission" section of the course Assignments page. It is OK if you haven't finished working on the files yet.
3. By **11:59pm on Wed May 9**, make your final submission, again following the aforementioned steps 1-3.[3]

6 Optional Extensions

One nice thing about object-oriented programming is that it provides a built-in framework for expanding the functionality of your application. For this assignment, we offer you the option of submitting some extensions to the Uno game. This is entirely optional and no extra credit will be offered. How would you extend the Uno game to add functionality? Are there different types of Players or Cards that could be created? What if you wanted to implement an entirely different card game?

If you would like to submit some extensions for our consideration, please do so a new file named `optional_extensions.py` and submit this file alongside your assignment. If your extensions require modifications to the base `Uno`, `Player`, `UnoCard`, or `UnoGameState` classes in a manner that those classes do not work without your modifications, then please instead create copies of these classes in your `optional_extensions.py`. Please make sure that your implementation of the Uno game still works without the `optional_extensions` to get credit for the assignment.

[3]The 2pm checkpoint on Wed May 9 provides you a chance to alert us during business hours if any problems arise. Since you've been warned to submit early, do not expect that we will accept work that doesn't make it onto CMS on time, for whatever reason. There are no so-called "slipdays" and there is no "you get to submit late at the price of a late penalty" policy. Of course, if some special circumstances arise, contact the instructor(s) immediately.