CS 112 – Project #6 **Due SUNDAY, December 8th, 11:59pm.** **Classes and Exceptions**

Note! No tokens may be used on this last assignment – the deadline is final.

Background:
Classes allow us to define new types for Python. We can first think of a class as defining a new container – instead of a list, tuple, set, or dictionary, we can have our own collection of values, each with a chosen name rather than an index/key. We can then read out a value or update a value, much like reading or replacing the values in a list or dictionary. But we can also put methods in a class definition, giving us a way to specify the exact ways we should interact with values of this new type. Once we have created a class definition, we can create as many objects of the new type as we want and use them in our programs. We can create entirely new types during the design phase of writing a program. This enables us to think in terms of types (and instances of types) that better model what we see in the real world, instead of endlessly relying on lists or dictionaries (and having to remember exactly how we intended to use those things as proxies for the values we actually had in mind).

Exceptions allow us to recover from unusual events – some unavoidable (user types in bad file name or bad input in general), some unexpected (such as `IndexErrors` when stepping through a list). Without exceptions, we tend to program in an "ask permission, then attempt" style of calculation. But using exceptions, we can instead program in a "try it first, ask for forgiveness" approach by writing code that specifies how to recover from exceptions that we allowed to occur.

- Project Basics document: http://cs.gmu.edu/~marks/112/projects/project_basics.pdf
- Project Six tester file: http://cs.gmu.edu/~marks/112/projects/tester6p.py
- Available test batch names: much like per-function in the past, you can run tests based on their names with these identifiers that we've embedded into the tester:
  - init/str/repr/eq tests, per class:
    `Move  PlayerException  Player`
  - added test batches based on method names:
    `change  copy  reset_history  reset  update_points  ever_betrayed`
    `record_opponent_move  copy_with_style  choose_move`
  - test batches by function name:
    `turn_payouts  build_players  composition  run_turn  run_game  run_tournament`

Grading Rubric

There are 105 test cases, each worth one point. (for 105% top score). There's no particular extra credit part.

- if you don't comment your code or submit it correctly, we might deduct up to 10 points.
- if you use globals, we might deduct up to 5 points.
- if you `import` disallowed things, you might lose all points for related test cases.

What's Allowed?

As long as you don't import anything, **you can use anything built-in or that you create**. Now that we're really writing our own datatypes, the built-ins aren't really designed to solve our quite-specific problems! Just as we used to say "add your own helper functions and call them", you can also add your own helper methods and use them.

Task – A Game of Trust!

This project is happily based upon this wonderful discussion of trust and game theory: https://ncase.me/trust/ While you can just read this document, that site gives a wonderful introduction to the ideas we will be implementing and simulating in our last project of the semester. We will implement a tournament of competitions where players may both win or lose, to see how trust can factor into our decisions.

Notes

Two players will face each other. They each decide independently to "cooperate" or "cheat". If they both cooperated, they each win two points. If they both cheated, nobody wins anything. If one cooperates and one cheats, the cheater gets +3 and the cooperator loses a point. That wasn't very kind! ☹

- One **turn** is defined as each player making a choice, and winning or losing some points as a result. Shared history against this player is available to players, and they may use this to guide their decision. It costs one point to play a turn. A player's points may go negative, and yet they can still spend/lose more points.

- One **game** is defined as two players playing multiple turns together. Their increasingly shared history will certainly affect their chosen behavior: will they keep cooperating even if their opponent doesn't? Will they hold grudges? Between games, players will reset their history; perhaps this new person is more trustworthy!

- A **tournament** involves multiple players. Each round, all players play each other once, accumulating or losing points. At the end of the round, some lowest scorers are removed (and any others with negative points), and some highest scorers are duplicated. After all rounds have been played, we see the "evolved" population of who remains, as a summary dictionary of the count of each style of player that survived.

Kinds of Players

There are five kinds of players in our version of the tournament. (they have unique first letters too...)

- `previous` starts by cooperating, then always copies the previous move of their opponent. (this was the "copycat" in the shared link).
- `friend` always cooperates, no matter what. May be vulnerable to abuse! (this was the "cooperator" in the shared link).
- `cheater` never cooperates. (this was the "cheater" in the shared link).
- `grudger` cooperates as long as the opponent does, but after one non-cooperation, never cooperates ever again with this opponent. (this was the "grudger" in the shared link)
- `detective` starts with cooperate, cheat, cooperate, cooperate; then if they've never been betrayed, they cheat. If ever they are betrayed, they behave like `previous` does. (this was the "detective" in the shared link)

Required Classes

Working through the classes in the given order is the simplest path to assignment completion; but many methods can be completed out of order if you get stuck on others. Many of the methods are quite simple, once you get comfortable accessing the fields of the object. Just be sure to complete the `init/str/repr/eq` definitions before moving on to other classes that use them!

Methods `return None` when no other return value is specified.

`class Move:` represents one move, indicating a choice to cooperate or not.

- `def __init__( self, cooperate=True):` constructor of a move: create/initialize the `cooperate` field.
- `def __str__ (self):` return `"."` for a cooperating move, `"x"` for a non-cooperating move
- `def __repr__(self):` create/return a string as in this example: `"Move(True)"`
- `def __eq__ (self, other):` return boolean: whether this move has same `cooperate` value as the `other`
- `def change (self):` change the cooperation status of this move.
- `def copy (self):` create/return a new `Move` object that has the same cooperation status.

`Class PlayerException(Exception):` Represents any exceptional situation for players, with a message stored inside of it. Be sure to include the `(Exception)` portion of the signature, so our `PlayerException` type is an exception type (and can thus be `raise`d and `except`ed).

- `def __init__(self, msg):` create/initialize a `msg` field based on the parameter.
- `def __str__ (self):` return the `msg`.
- `def __repr__(self):` return a string like this: `"PlayerException('message contents')"`

`class Player:` Represents one player, whose `style` string will resolve their behavior. A `Player` object keeps track of the needed internal state: previous moves of the opponent, their style, and their points.

- `def __init__(self, style, points=0, history=None):` create/initialize fields for each parameter.
  - If `history` is `None`, then re-assign an empty list to it. Assume it's a list of moves otherwise.
  - If style isn't one of our five accepted styles, `raise` a `PlayerException` with a message like `"no style 'copykitten'."`
- `def __str__(self):` create/return a string (utilizing `Move`'s str) like: `"friend(12)+.+.+..."`
- `def __repr__(self):` create/return a string like: `"Player('grudge', 12, [Move(True), Move(False)])"`
- `def reset_history(self):` reset `history` to an empty list. Don't change `points`.
- `def reset (self):` reset `history`, and also reset `points` to zero.
- `def update_points(self, amount):` increase points by the parameter. Might go negative.
- `def ever_betrayed(self):` return whether an opponent ever cheated, based on the `history`.
- `def record_opponent_move(self, move):` add the given `Move` to the end of the history.
- `def copy_with_style(self):` create/return a `Player` with no `history`, and matching `style` to this one.

- `def choose_move(self):` determine/return what `Move` to do next, based on the given `style` and opponent's move history. This doesn't perform the move though; that's the job of the `run_turn` function.
  - See the previous pages' description for the five types of players and their behaviors; all of that decision logic does in fact get implemented here!

Required Functions

These functions are **not part of any of the previous classes**, but they can *use* the class definitions to make objects, call their methods, and interact with them. The classes gave us new types, now let's put them to use!

- `def turn_payouts(move_a, move_b):` return a tuple of the payouts players a and b get with those moves.
  - if both cooperate, each gets +2. So you would return `(2,2)`.
  - if both cheat, each gets +`0`.
  - if one cheats and the other cooperates, the cheater gets `+3` and the cooperator gets `-1`.

- `def build_players(initials):` given a string of initials, create and return a list of `Player`s of those styles, with default values for all other parameters (points, history).
  - If some other character is present, `raise` a `PlayerException` with the message like `"no style with initial 'q'."`
  - Example: `build_players("pf")` ⟶ `[Player('friend', 0, []), Player('previous', 0, [])]`

- `def composition(players):` given a list of players, create and return a dictionary of each player type (the keys) and how many of that style were in the list of given players. Don't include styles with zero players.
  - Example: `{"friend":5, "previous":9, "grudger":1}` `# no detectives/cheaters`
  - Hint: you might want a helper function that prints this out nicely to help you debug other stuff.

- `def run_turn(player_a, player_b):` have the two players pay 1point to play; get them to choose moves; record each other's move in `history`; determine their payouts; and adjust their `points`. Returns nothing.
  - If both players have the same `id()`, `raise` a `PlayerException` with the message `"players must be distinct."`

- `def run_game(player_a, player_b, num_turns=5):` given two players, reset their histories first, and then have them play a total of `num_turns` times together. Returns nothing, but we could inspect their points/histories later on to see how it went.
  - If the players were identical, handle the raised exception by returning prematurely (no other effects).

- `def run_tournament(players, num_turns=10, num_rounds=5, starting_points=0, num_replaces=5):` given a list `Player`s, and numbers of turns, rounds, starting points, and replacements, this function will do the following:
  - run `num_rounds` rounds (assume `num_rounds` is positive). Each round:
    - reset all points/history for everyone (before any games in the round, everyone has `starting_points`)
    - everybody plays everybody else exactly once; every game must have `num_turns` turns. Reset history each game but players accumulate/lose points the entire round.
    - find the `num_replaces` lowest scorers; remove them. Then, remove anyone with negative points.
    - find the `num_replaces` highest scorers; create new players that copy their styles, and append them to the end of the players list.
    - if all players are gone, `raise` a `PlayerException` with this message: `"all players died after round 4."` (use the actual round number; the first round is round #1).
  - Return a "composition" dictionary of the results (see `def composition` above).

Going Further

At this point, the project is done. But!

If you were to write a `main()` function that accepts all the arguments of `run_tournament` as command-line arguments (except using a string of initials instead of player objects), printed the composition of players before and after the tournament, you'd have a nice top-level interface to the whole program. You can already just go to your code interactively and write something like

```
>>> run_tournament(build_players("fffcccddd"),10,5,0,5)
```

But now you could just invoke it from the command line, e.g.:

```
os-prompt$ python3 code.py fffcccddd 10 5 0 5
```

Here's what main could look like:

```
import sys
def main():
    # read in all (optional) arguments.
    if len(sys.argv)>1:
        initials = sys.argv[1]
    else: initials = "pfcgd"*5 # five of each
    # the rest are all integers.
    more_args = []
    for arg in sys.argv[2:]:
        more_args.append(int(arg))

    # create the players, show them before the tournament.
    ps = build_players(initials)
    print(composition(ps))

    # try the tournament and print either the result or the issue.
    try:
        ans = run_tournament(ps, *more_args)
        print(ans)
    except PlayerException as e:
        print(e)

# if your file was run from the command line (and not imported from another file),
# then this will call your main function.
if __name__=="__main__":
main()
```