Before you begin

- Create two files, *complex.py* and *gachapon.py*
- Read up on complex numbers if you have forgotten how they work
- Read chapters 15 and 16 in the text book

Learning Objectives:

You should be able to do the following:

- Make a class
- Add class attributes and have them store immutable and mutable data
- Override the base members such as repr, +, str by using magic methods

Thoughts (come back and read these after you have read the problems):

- Python already has support for complex numbers. We are going to ignore that for the purposes of this assignment. Your code should not use any built-in Python functionality for complex numbers (except possibly for testing).
- *__repr__* and *__str__* do similar things. Python internally uses *__repr__* sometimes and *__str__* at other times. You should implement *__repr__* to return the string representation of the complex number class. Then have *__str__* have a single line: return *self.__repr__()*

Grading Rubric: [16 points]

In regards each item, please see associated footnotes for each item [*], [†].

- [1 points] `GachaponSimulator` initializes correctly [*].
- [0.5 points] `_simulate_once` works and produces the expected number [*]. If no footnote is include then the portion will be manually graded.
- [0.5 points] `reset` works [*].
- [1 points] `simulate` works [*].
- [1 points] `get_summary_stats` works as specified [*].
- [1 points] Command line interface which allows user to pass in a number of prizes and iterations works.
- [2 points] Complex initialization works, with values saved correctly [*].
- [1 points] `__repr__` and `__str__` work as described [*].
- [2 points] Addition works and produces the right value for two complex objects, a complex number plus a number, and a number plus a complex.
- [2 points] Multiplication works for all three cases [†].
- [Extra Credit 2 points] Substraction, and division work for all three cases [†].
- [2 points] Code passes PEP8 checks with 10 errors max [*].
- [2 points] Style checks, cleanliness, layout, readability, etc.

[*] Indicates that the autograder will tell you if this problem is correct on our set of test cases and assign you credit.
[†] Indicates the autograder will grade this in the background, but only tell the result after the homework has been published.

Problem 1: Gachapon Simulator Revisited

Recall from Lab 4 we wrote a funciton `simulate_gachapon` that simulated the number of iterations it took to win all N prizes out of a selection of N prizes, assuming each prize was drawn uniformly with replacement. The objective here is to take this code and package it into a compact class which is easy to interact with. Create a new file `gachapon.py` for this assignment.

- Write a class `GachaponSimulator` which initializes with one parameter `prizes_n` which represents the size of the prize pool. You should save it in an appropriately named attribute. You should also create an attribute called results which defaults to an empty list.

  Note: you are not yet running the simulator when you initialize something like `our_sim = GachaponSimulator(10)`; you are simply "laying the groundwork", so to speak.

- Write a method called `_simulate_once` which takes in no additional arguements besides `self` and runs one round of the gachapon game. It should return the number of iterations the game took.

- Write a method called `reset` which takes in no additional arguements which resets the results attribute to an empty list.

- Write a method called simulate which takes in one additional argument `num_games` representing the number of games to simulate. It should then run that number of games and record the number of iterations for each game. It should then take these results and append them to the `results` attribute.

  Note that the `results` list should not be reset if simulate is called multiple times. This means that if we call `our_sim.simulate(30)` and then `our_sim.simulate(40)` the `len(our_sim.results)` should be 70.

- Write a method called `get_summary_stats` which takes in no additional arguements. It should return a dictionary with the number of games which have been currently been run with key `n`, the mean number of iterations across all games with key `mean`, and the population standard deviation with key `stdev` from the current results list.

  if the user has not run any games, both `mean` and `stdev` should be the value `None`. If only one game has been run then `stdev` should be `None`.

- Finally, implement a command line interface at the end of `gachapon.py` which lets you pass in two arguments, a number of prizes and a number of games to play, and that will print out a nicely formatted message telling the user the results of the simulation. The command would look like:

  ```
  python gachapon.py 10 1000
  ```

  A possible output would be:

  ```
  Running 10-prize lottery simulator 1000 times
  Average number of iterations was 32.15 (standard deviation of 5.18)
  ```

  You should take advantage of the interfaces defined in `GachaponSimulator` to reduce the amount of code for this section; the process of actually running the simulation and obtaining the stats should be just a few lines of code.

Problem 2: Complex class setup

In this section, we will write a `class` which represent s a complex number. Note that python already has support for complex numbers built in; for this example, we are expecting you to reimplement the basic functionality from scratch. All work should go in a file called `complex.py`

- Create a `Complex` class that initializes with two optional parameters representing real and imaginary components, defaulting to 0 if they are not specified. Store the real and imaginary components as floats in attributes call `re` and `im`. Therefore, we should be able to do something like:

```
a = Complex(-3, 2)
b = Complex(2)
print(a.re, b.im)  # Should print -3.0 0.0
```

- Implement the `__repr__` and `__str__` magic methods so that when we try to print out your complex number, we get a readable representation of your complex number, exactly following the format shown below:

```
print(Complex(-3, 2))
(-3.0 + 2.0i)

print(Complex())
(0.0 + 0.0i)

print(Complex(3.4,-2.1))
(3.4 - 2.1i)
```

Problem 3: Complex class - Arithmetic

In this section, we will extend your `Complex` class to be able to do basic arithmetic. You should be able to add Complex objects with other complex objects, but you should also be able to add them to other integers and floats.

While we will not grade you on test code for this section, we highly encourage you to write some test code to make sure everything is operating as expected. **The autograder will not show you correctness results until after the assignment is graded.**

- Start by implmenting addition, so that all of the following will work

```
a = Complex(2.0, 3.0)
print(a + Complex(-1.5, 2))  # (0.5 + 5.0i)
print(a + 8)                 # (10.0 + 3.0i)
print(3.5 + a)               # (5.5 + 3.0i)
```

  Note that the result of the addition operation should be another Complex object. You will need to define the `__add__` and `__radd__` magic methods to get these to work.

- Next, implement multiplication, so that all of the following will work:

```
a = Complex(1.0, -3.0)
print(a * Complex(4.0, 5.5))  # (20.5 - 6.5i)
print(a * 3.5)                # (3.5 - 10.5i)
print(-2 * a)                 # (-2.0 + 6.0i)
```

- **Extra Credit:** Implement subtraction and division in the same vein. Note that because subtraction and division are non-commutative, you should pay special attention to how you define the reverse magic methods. Be sure to test you code. Note that the magic method for division is `__truediv__`.