Faculty of Arts and Sciences
Department of Computer Science
CMPS 200 – Introduction to Programming
Assignment 10 – Due Nov 20, 2017

Reading material: chapter 18, sections 16.5 (invariants and assertions)

1. Squares and Triangles.
Implement classes `Square` and `Triangle` as subclasses of class `Polygon` from the previous assignment. Each will overload the constructor method `__init__()` so it takes only one argument l (the side length), and each will override method `area()` that computes the area using a simpler implementation. The method `__init__` should make use of the superclass `__init__` method, so no instance variables (l and n) are defined in subclasses. Note: The area of an equilateral triangle of side length s is $s^2 \cdot \sqrt{3}/4$.

2. Bank Account, version 2
The `BankAccount` class of the previous homework had a number of problems: (1) a bank account with a negative balance can be created, (2) the withdrawal amount may be greater than the balance, and (3) the deposit amount may be negative. Modify the implementation so that a `ValueError` exception is raised for any of these violations, together with an appropriate message: 'Illegal balance', 'Overdraft', or 'Negative deposit'.

3. Savings and Checking Accounts
Implement two classes `SavingsAccount` and `CheckingAccount` as subclasses of class `BankAccount`.

(a) Start by modifying the constructor of `BankAccount` so it initializes: the account balance to the value of one input argument (same as asst9), the account type (a string field) to the value of one input argument, and the open date (a datetime field) to the value of the current date and time.

(b) Modify the `__str__` method of `BankAccount` so it prints the account type, the date the account was opened, along with the current balance

(c) Implement the `__lt__` method for `BankAccount` so it compares two accounts by their respective balance.

(d) Add a constructor to `SavingsAccount` so it takes an additional input argument `rate` which initializes the interest rate field

(e) There are no special requirements for the `CheckingAccount` other than making the instantiation of a CheckingAccount object possible.

(f) Add a method to `SavingsAccount` named `compute_interest()` which takes an integer argument `n` indicating the number of years and uses it along with the interest rate to compute and return the balance of the account after `n` years of compounding interest using this formula: $A = P(1 + r/n)^{nt}$ where:

- A = the future value of the investment/loan, including interest
- P = the principal investment amount (the initial deposit or loan amount)
- r = the annual interest rate (decimal)
- n = the number of times that interest is compounded per year (set it as a constant 12)
- t = the number of years the money is invested or borrowed for

(g) Add a method to `SavingsAccount` named `earn_interest()` which adds to the balance the amount of interest earned in one month.*Hint: Calls on compute_interest() and passes it* 1/12

(h) Test your code by creating a `SavingsAccount` with an opening balance of $5,000 and an interest rate of 5% then using a loop print the balance at the end of each month for a total period of 36 months. Your output should match `interest.txt`

(i) Test your code by creating a list and populating it with 10 instances of `SavingsAccount` each with a random initial balance (50-5000), then sort the list by account balance and print all the accounts. Your output should match `accounts.txt`

4. Periodic Table.
Create a class `Element` for representing elements in the periodic table of elements. The constructor should define attributes to store element name, atomic number, symbol, and atomic weight, and accessor methods for each of these values. Create appropriate tests to verify the correctness of the implementation.

Create a class `PeriodicTable`. The data type should have:

- a constructor that reads values from a file to create a dictionary whose keys are strings representing element symbols and whose values are `Element` objects.
  The file `elements.txt` on moodle contains the data pertaining to elements, one element per line. You will need to loop over the lines of the file and for each line use the string `split()` method to obtain the first four values in it (name, number, symbol, and weight).
- a method `weight()` that takes in a string representing a molecule name, such as `'H2 O'`, and returns its molecular weight.
  The string `split()` method will again be useful here for separating the contribution of each atom. The methods `isdigit()` or `isalpha()` may be used to separate the atomic symbol (e.g., `'H'`) from the number of times it occurs.

5. Deck of Cards.
To model a deck of playing cards we create a class to model a single card, and one to model a group of cards. Given `card.py` complete the code for the `Card` and `Card_group` classes wherever you see a *'your code here'* comment. You are not allowed to modify existing code, just add to it.

(a) Test your code by creating a single card, then creating an instance of `Card_group` with any a few cards. Try printing the instance, you should get an image that shows all the cards in the group similar to the image shown below.

(b) implement a class `Standard_deck` that inherits from `Card_group` and add a constructor that initializes the deck with all 52 cards (13 of each suit).

(c) Use the inherited `nextCard()`, and `hasCard()` on the instance of `Standard_deck` to print all cards. Verify that you have printed all 52 unique cards.

(d) Add a method `deal()` to `Standard_deck` which takes in a parameter indicating the number of cards to deal.

(e) Use the `shuffle()` method inherited from `Card_group` to shuffle the cards and print cards until you get an Ace of Hearts.

(f) Create a class `Deck_of_hearts` that has only the thirteen cards of hearts