Homework 3

Total Points:

100 (Correctness and Style) + 3 EC (Checkpoint) + 5 EC (Question 7)

Due Dates (SD time):

- **Checkpoint (read below): <u>Sunday, January 26st, 11:59pm</u>**
- **Entire Assignment: <u>Tuesday, January 28th, 11:59pm</u>**

Starter Files

Download hw03.zip. Inside the archive, you will find starter files for the questions of this homework. You cannot import anything to solve the problems.

IMPORTANT: Coding and Docstring Style

This is a reminder that your code style will be graded. Here are a few useful links:

Style Guide Document

Style Guide on Course Website

Style Guide Examples

Submission

By the end of this homework, you should have submitted the homework via Gradescope. You may submit more than once before the deadline; only the final submission will be graded. Refer to Lab00 directions for submission.

Testing

At any point of the homework, use the following command to test your work:

```
>>> python3 -m doctest hw03.py
```

Checkpoint Submission

Due Date: Sunday, January 26th, 11:59pm (SD time)

You can earn up to **3 points extra credit** by submitting the checkpoint by the due date above. In the checkpoint submission, you should complete **Question 1, 2** and submit ***hw03.py*** file to gradescope.

Checkpoint submission is graded by some simple hidden tests (no tests against edge cases). Note that in your final submission, you should still submit these questions, and you may modify your implementation if you notice any errors.

A Guide on Asserts

Starting from HW03, you will be asked to write **assert statements** whenever a question requires them. Assert statements are used to prevent any unexpected input that may corrupt your code, including arguments in invalid type, and arguments that do not fit the logic and the input constraints. Refer to the Style Guide Example for examples. Here are a few important notes to guide you:

1. You should check **each** input argument, so go through them one by one;
2. The requirements of an argument will be provided in the question prompt. For example, if the write-up says: take in a length-5 list of positive integers called `lst`, you should check the following **in order**:
    a. `lst` is a list
    b. `lst` has length 5
    c. The elements of `lst` are all integers
    d. The elements of `lst` are all positive
3. Note that order is important because some checks depend on the correct check result of previous ones. For example, if the length check happens before the list type check, an error will occur when the input is an integer, because you cannot apply the `len()` function on an integer.
4. For type checks, the functions `type()` or `isinstance()` are helpful.
5. For assertion error doctests, please follow examples given in the doctest. In particular, you could simply copy-paste the expected output for all assertion errors.
6. When you need to check everything inside a list, for example, you will need to loop through all elements. Instead of using an explicit for loop, **you must use list comprehension with the `any()/all()` function** to achieve this in this class. For example,

Incorrect	**Correct**
`for idx in index:`	`assert all([`
`    assert isinstance(idx, int)`	`    isinstance(idx, int)`

	` for idx in index`
	`])`

7. The tutorial on `any()`, `all()` function can be found here.
8. Assert statements do not count towards the number of lines in the one-line list comprehension requirement.
9. In Python, boolean values are integers (source). You can assume that in cases where you need to assert for integer types, booleans will NOT be tested.

Example (assert + magic number declaration + list comprehension requirement):

```
def increment(nums):
    """
    Docstring (and doctests) omitted.
    """
    increment_amount = 10
    assert isinstance(nums, list)
    return [num + increment_amount for num in nums]
```

General Notes and Requirements

1. **DO NOT IMPORT ANY PACKAGES.**
2. Please add your own doctests (**at least three**) as the given doctests are not sufficient. **You will be graded on the doctests**.
3. When a question **(Question 1.1, 1.2, 1.3, 2, 3, 4, 5)** requires **list comprehension:** your implementation should have list comprehension(s) and possible assignment lines, including magic number declarations. **No explicit for/while loops** are allowed. **No helper functions** are allowed. **No map/filter/lambda** are allowed (will be covered later).
   For example, the following is allowed:
   ```
   lst_1 = [... for ... in ...]
   lst_2 = [... for ... in ...]
   return ...
   ```
4. When a question **(Question 1.1, 1.2, 1.3, 3, 4, 5, 7)** requires **assert statements**: follow the assert guide to write assert statements to prevent any unexpected inputs. You will be graded on this. If a question does not require assert statements, assume all inputs are valid.

Required Questions

Question 1.1

As you explore more about Pythonia, you realize it is in fact an incredibly animal-friendly world. You are able to see many of them just walking down the streets.
Write a function that takes in two parameters: `animal_dict` and `target`.

- `animal_dict` is a dictionary where each dictionary key is a string of an animal name and its value is a string that states the type of that animal.
- `target` is a string of a type of animal.

The function returns a list of animal names(s) that are of the same type as `target` (case-insensitive).

Requirements: Assert statements, List Comprehension

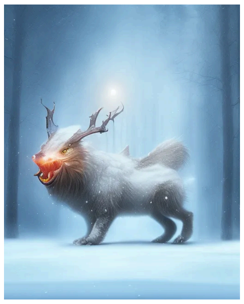
```
def animals(animal_dict, target):
    """
    >>> animals_dict = {"dogs": "mammals", "snakes": "reptiles", \
    "dolphins": "mammals", "sharks": "fish"}
    >>> target = "mammals"
    >>> animals(animals_dict, target)
    ['dogs', 'dolphins']

    >>> target = "amphibians"
    >>> animals(animals_dict, target)
    []

    >>> animals_dict = {True: "mammals"}
    >>> animals(animals_dict, target)
    Traceback (most recent call last):
    ...
    AssertionError

    """
```

Question 1.2



After the Festival, you have so much food left that you've decided to feed them to these animals. But first, you need to decide what exactly you can feed to the animals.

Write a function that takes one parameter, a dictionary, where each dictionary key is a string of an animal name and its value is a list with two elements: `[<current_energy>, <animal_type>]`:

- `<current_energy>` is an integer representing the current energy state of the animal.
- `<animal_type>` is a string representing the type of the animal.

Your function should return a list with the corresponding animal type.

Requirements: Assert statements, List Comprehension

```
def animal_type(animals):
    """
        >>>  animals  =  {"dogs":  [10,  "omnivores"],  "snakes":  [8,
"carnivores"], \
    "lions": [12, "carnivores"], "sheep": [7, "herbivores"]}
    >>> animal_type(animals)
    ['omnivores', 'carnivores', 'carnivores', 'herbivores']

    >>> animals = {"bears": [5, "omnivores"]}
    >>> animal_type(animals)
    ['omnivores']

       >>>  animals  =  {"dogs":  [0.8,  "omnivores"],  "snakes":  [8.1,
"carnivores"], \
    "lions": [12, "carnivores"], "sheep": [7, "herbivores"]}
    >>> animal_type(animals)
    Traceback (most recent call last):
    ...
    AssertionError
    """
```

Question 1.3

Now it is time to have an animal feast! Write a function that takes three parameters: `animals`, `food`, `target_energy`.

- `animals` is a `dictionary` described in Question 1.2
- `food` is a `tuple` with three elements: (`<food>`, `<animal_type>`, `<gained_energy>`):
    - `<food>` is a string of the name of the food that will be given.
    - `<animal_type>` is a list that contains strings of the types of animals that the food would be given to (case sensitive).
    - `<gained_energy>` is a positive integer representing the energy level that the animal would gain if they consume one unit of the food.
- `target_energy` is a positive integer that is the desired energy state for each animal.

Your function should return a list that contains tuple(s) of length 2 and/or string(s):

- If an animal can consume the given food (`animal_type` matches), append a tuple where the first element is the name of the animal and the second element is the number of units the animal has to consume to meet the target energy state.
- If the animal cannot consume the food, it would be mapped to the string `'dislike :('`

For example:

Input	**Output**	**Explanation**

`animals = {`
`"dogs": [10, "omnivores"],`
`"snakes": [8, "carnivores"],`
`"lions": [12, "carnivores"],`
`"sheep": [7, "herbivores"]`
`}`
`food = ("lettuce", ["omnivores", "herbivores"], 2)`
	`[`
	`('dogs', 3),`
	`'dislike :(',`
	`'dislike :(',`
	`('sheep', 4)`
	`]`
		• `'dogs'` need to consume 3 units of `'lettuce'` to meet the target energy level because the consumption of one unit of `'lettuce'` would help `'dogs'` gain 2 units of energy.
		• `'snakes'` and `'lions'` are mapped to `'dislike :('` because they are neither

`target = 15`		omnivores or herbivores • `'sheep'` needs to consume 4 units of `'lettuce'` to meet the target energy level for the same reasoning above for `'dogs'`.

Requirements: Assert statements, List Comprehension

```
def hungry(animals, food, target_energy):
      """
      >>> animals = {"dogs": [10, "omnivores"], "snakes": [8,
"carnivores"], \
      "lions": [12, "carnivores"], "sheep": [7, "herbivores"]}
      >>> target = 15
      >>> food = ("lettuce", ["omnivores", "herbivores"], 2)
      >>> hungry(animals, target, food)
      [('dogs', 3), 'dislike :(', 'dislike :(', ('sheep', 4)]

      >>>  animals  =  {"dogs":  [10,  "omnivores"],  "snakes":  [8,
"carnivores"], \
      "lions": [12, "carnivores"], "bears": [5, "omnivores"]}
      >>> target = 15
      >>> food = ("grass", ["herbivores"], 2)
      >>> hungry(animals, target, food)
      ['dislike :(', 'dislike :(', 'dislike :(', 'dislike :(']

      >>> food = ("berries", "herbivores", 2)
      >>> hungry(animals, target, food)
      Traceback (most recent call last):
      ...
      AssertionError

      """
```

Question 2

Animals are fed! It is time to have some fun. You want to find the best arcade game to purchase that you can afford.

Write a function called `affordable_arcade_game` that takes two parameters:

- `arcade_games` is a dictionary where the keys are the names of the arcade games, and the values are lists of tuples. Each tuple contains two items: the price (an int) of the game and its rating.
    - Note, a key can have more than one tuple in its value. (think of them as different versions with different ratings).
- `budget` is an integer representing your budget for purchasing an arcade game

Your function should return the title of the arcade game with the highest rating that is affordable within the given budget. If there is a tie, return the title with the highest alphabetical order.

Requirements: **List Comprehension**

Note: No assert statements needed. You may assume the input is always valid and there will always be at least one affordable game within budget.

```
def affordable_arcade_game(arcade_games, budget):
    """
    >>> arcade_games = {'Pac-Man': [(5, 4.7), (10, 4.9)],\
    'Galaga': [(8, 4.8), (12, 4.6)],\
    'Street Fighter II': [(15, 4.9)],\
    'Mortal Kombat': [(12, 4.7), (18, 4.8)],\
    'Donkey Kong': [(10, 4.5), (14, 4.6)]}

    >>> affordable_arcade_game(arcade_games, 10)
    'Pac-Man'

    >>> affordable_arcade_game(arcade_games, 15)
    'Street Fighter II'
    """
```

Question 3

Yay, you can finally play the game but you stumble upon a new one. A *gacha game*! In this *gacha game*, there are different types of prizes with corresponding probabilities.

Write a function called `calculate_probability_ratios` that takes three parameters:

- `prizes` is a list of strings of all the prize names.
- `probabilities` is a list of floats, each probability corresponds to the corresponding prize at the same index.
- `target_prize` is a string.

- `prizes` and `probabilities` lists have the same length and `target_prize` exists in `prizes`.

Your function should return the ratio of the probability of winning the target prize to the sum of probabilities of all prizes.

Requirements: Assert statements, List Comprehension

```
def calculate_probability_ratio(prizes, probabilities, target_prize):
    """
    >>> calculate_probability_ratio(["Gold Coin", "Silver Coin",
"Gem", \
        "Key"], [0.1, 0.2, 0.3, 0.4], "Gem")
    0.3

    >>> calculate_probability_ratio(["Sword", "Shield", "Bow",
"Staff"], \
        [0.25, 0.2, 0.3, 0.25], "Shield")
    0.2

    >>> calculate_probability_ratio(["Coin", "Shield", "Gem"], \
        [0.25, 0.2, 0.5], 0)
    Traceback (most recent call last):
    ...
    AssertionError
    """
```

Question 4

There is a weekend in Pythonia and you noticed that all the tutors are playing an arcade. You want to try it out, but they require you to solve a problem first, making sure you are not behind on your assignment.

Write a function that takes in a string that should only have alphanumeric characters and has at least one numeric character as input. Your function should return the string account after transforming the input using the following rules:

- All non-vowel alphabets should be in lowercase
- All vowels should be uppercase
- All digits should be moved to the front of the string in order
- The numeric part of the string should repeat *i* number of times where *i* is the last digit in the input string

For example:

Input	Output	Explanation
`string = “helloDSC20”`	`‘hEllOdsc’`	There is no numeric part in this string because the last digit of the input string is 0
`string = “helloDSC21”`	`‘21hEllOdsc’`	There is only one ‘21’ because the last digit of the input string is 1

Hints:

1. `str.isalnum()` and `str.isnumeric()` or `str.isdigit()` might be helpful.
2. `str.join()` might be helpful. Visit this tutorial for examples.
3. To assert strings as alphanumeric, look up this tutorial.
4. According to the tutorial above, empty strings are **non**-alphanumeric strings.
5. Try to break the input string down into the different cases and apply list comprehension from there.

Requirements: Assert statements, List Comprehension

```
def account(string):
    """
    >>> string = "helloDSC20"
    >>> account(string)
    'hEllOdsc'

    >>> string = "DSC2023python"
    >>> account(string)
    '202320232023dscpythOn'

    >>> string = "hello!! DSC20"
    >>> account(string)
    Traceback (most recent call last):
      ...
      AssertionError
      """
```

Question 5

After exploring the arcade games, it was time for you to go home. But do not forget about the prizes! In the prize booth, there are a variety of prizes, each with its own ticket cost.

Write a function called `prize_combinations` that takes two parameters:

- `prizes` is a list of tuples, each tuple contains the prize name as a string and the ticket cost as an integer.
- `target_tickets` is the total amount of tickets you can spend at the prize booth, which is represented as an integer.

Your function should return all possible **pairs** of prizes that can be redeemed without exceeding the target number of tickets. Make sure your output list is sorted in alphabetical order. (sorted() is a great function to use here)

Requirements: Assert statements, List Comprehension

Note: Each prize may only be redeemed once, i.e. you should not consider pairs of the same prize.

```
def prize_combinations(prizes, target_tickets):
      """
      >>> prizes = [("Plush Toy", 10), ("Board Game", 20), ("Puzzle",
5), ("Remote Control Car", 30)]
      >>> prize_combinations(prizes, 15)
      [('Plush Toy', 'Puzzle'), ('Puzzle', 'Plush Toy')]

      >>> prize_combinations(prizes, 10)
      []

      >>> prize_combinations(prizes, 1.0)
      Traceback (most recent call last):
      ...
      AssertionError
      """
```

Question 6

You were walking out from the prize booth when you noticed that everyone is upset: one of the games is bugged and needs to be fixed: a new inventory system needs to be created.

Write a function called `manage_inventory(inventory, action, item_name, quantity)` that manages the game's inventory.

- `inventory` is represented as a dictionary, where keys are item names, and values are quantities (positive integers).
- `action`: specific operation (represented as a string)
- `item_name`: item to update (represented as a string)
- `quantity`: used in `action` (represented as an integer equal to or more than 0)

The function should work as follows:

- If the `action` is "add", add the specified quantity to the item's quantity in the inventory.
- If the `action` is "remove", subtract the specified quantity from the item's quantity in the inventory.
    - If the quantity is greater than what's available, no action should be taken and the dictionary would not be updated.

Assumptions:

- `item_name` exists in `inventory`.
- There will only be two actions possible ("*add*" and "*remove*").

Your function should return a dictionary with the updated values. Construct the returned dictionary with a *one-line dictionary comprehension* (will be explained in the discussion), which is a dictionary version of list comprehension. The format is:

{key : value for … in …}

Requirements: Assert statements, Dictionary Comprehension

```
def manage_inventory(inventory, action, item_name, quantity):
    """
    >>> inventory = {'coins': 100, 'health potions': 5}
    >>> manage_inventory(inventory, 'add', 'coins', 50)
    {'coins': 150, 'health potions': 5}

    >>> manage_inventory(inventory, 'remove', 'health potions', 10)
    {'coins': 100, 'health potions': 5}

    >>> manage_inventory(inventory, 'remove', 'health potions', 3)
    {'coins': 100, 'health potions': 2}

    >>> manage_inventory([], 'remove', 'health potions', 3)
    Traceback (most recent call last):
    ...
    AssertionError
    """
```

Question 7 (Extra Credit, 5 points)

After you got home, you realized that you had spent more money than you anticipated. So before your next visit you decide to do some research to help determine which game you should play that would likely let you gain a positive number of points with the amount of money you have. To do so, you look into the game history of each machine.

Write a function that takes in two parameters: `games` and `money`.

- `games` is a dictionary where the keys are the names of the game machines and values are lists of tuples that represent each round of a game from the game history of the corresponding machine. Each tuple has two elements: the first element represents the points gained/lost and the second element represents the number of *game coins* used for that round.
- `money` is a non-negative integer representing the MNT (Monty, Pythonia currency) amount you have to use to play the games. **Each MNT you have can be changed into 4 game coins**.

To choose a profitable game, you take two factors into account:

1. The sum of points gained/lost in previous games should be strictly positive.
2. The **MNT** money you have, converted to game coins, should be greater than or equal to the sum of game coins used in previous games.

Your function should return a string of a game's name. If multiple games meet the requirements, return the first game that appears in the given `games` dictionary. Assume that at least one game will be considered profitable.

For example:

Input	Output	Explanation

`games = {`
`"Pac-Man": [(20, 5), (15, 8), (-25, 12)],`
`"Space Invaders": [(-10, 3), (3, 5)],`
`"Street Fighter II": [(10, 2), (5, 8), (10, 7)]`
`}`
 	
`money = 5`
	`'Street Fighter II'`	• Since you have 5 MNT, which converts to 20 game coins.
		• The sum of points gained/lost is negative in `'Space Invaders'`, hence it does not satisfy the first condition.
		• The three rounds of `'Pac-Man'` used a total of 25 coins, hence it does not satisfy the second condition (25 > 20).
		• The function returns `'Street Fighter II'` because it satisfies both requirements.

Requirements: Assert statements, List Comprehension and/or dictionary comprehension only

```
def arcade(games, money):
    """
    >>> games = {"Pac-Man": [(20, 5), (15, 8), (-25, 12)], \
    "Space Invaders": [(-10, 3), (3, 5)], \
    "Street Fighter II": [(10, 2), (5, 8), (10, 7)]}
    >>> money = 5
    >>> arcade(games, money)
    'Street Fighter II'

    >>> money = 10
    >>> arcade(games, money)
    'Pac-Man'
    """
```

Submission

Please submit the homework via Gradescope. You may submit more than once before the deadline, and only the final submission will be graded. Please refer to the Gradescope FAQ if you have any issues with your submission.