ICS-33: In-Lab Programming Exam #3

1: Details of zip_skip:

The **zip_skip** generator function (a decorator for iterables) has 2 required arguments followed by any number of **iterable** arguments. The parameter names are:

- **predicate**: a function of one argument returning a **bool** value.
- **exception_bool**: a **bool** value to be used for a **predicate** call that raises an exception (see below for details).
- ***iterables**: any number (zero or more) of iterables.

When called, **zip_skip** returns a result that is **iterable** such that

- the **zip** part: produces a **tuple** with the first values produced by each of the iterables, followed by a **tuple** with the second values produced by each of the iterables, followed by a **tuple** with the third values produced by each of the iterables, etc. ...until one of the iterables cannot produce another value.
- the **skip** part: it skips (doesn't produce) any **tuple** if **predicate** returns **False** on **any** (one or more) value in the **tuple**; **IMPORTANT**: if **predicate** raises an exception, it should be treated as if it returned the value of the **exception_bool** parameter.

Using this defintion, executing

```
for i in zip_skip( (lambda x : is_even(x)), False, [0,0,1,1,0], [0,1,0,1,'X'])
   print(i)
```

prints only (note **0** is even, **1** is not even, **'X'** raises an exception)

```
(0, 0)
```

- The first two values in the **list**s are **0** and **0**: the **predicate** is **True** on both, so it produces **(0, 0)**.
- The second two values in the **list**s are **0** and **1**: the **predicate** is **False** on the second, so it skips **(0, 1)**.
- The third two values in the **list**s are **1** and **0**: the **predicate** is **False** on the first, so it skips **(1, 0)**.
- The fourth two values in the **list**s are **1** and **1**: the **predicate** is **False** on both, so it skips **(1, 1)**.
- The fifth two values in the **list**s are **0** and **'X'**: the **predicate** raises an exception on the second (**is_even** doesn't work on strings); since **exception_bool** is **False** we consider this exception to be treated like **False**, so it skips **(0, 'X')**.

If we instead specified **True** for the second argument matching the **exception_bool** parameter

```
for i in zip_skip( (lambda x : is_even(x)), True, [0,0,1,1,0], [0,1,0,1,'X'])
   print(i)
```

Python would print

```
(0, 0)
(0, 'X')
```

because now when the **predicate** is called on **'X'** and raises an exception, the call is treated as if it returned **True**.

Your code must work for any number of iterables. Do not assume anything about the **iterable** arguments, other than they are **iterable**; the testing code uses the **hide** function to "disguise" a simple **iterable** (like a **list**); don't even assume that any **iterable** is finite: so, don't try iterating all the way through any **iterable**.

Like the standard **zip** decorator, this one stops producing results when some **iterable** has no more values to produce (so as long as one **iterable** is finite, this generator function will be finite too). **Hint**: you may use Python's standard **zip** function in your solution.

Finally, **You may NOT import any functions from functools** to help you solve this problem. Solve it with the standard Python tools that you know.

2: Details of closest_sums:

The **closest_sums** function computes how to divide a **list** of values (**int**s) into two **list**s of values, such that the **sums** of values in each **list** are as close as possible. For example, if the original **list** were **[1, 10, 8, 11, 1, 9, 10]** (notice that the values may repeat), then we can divide it into the **list [1, 10, 11, 1]** which sums to **23** and the **list [8, 9, 10]** which sums to **27**. The **sums** are **4** apart: computed as the absolute value of the difference of their **sums**. Note that any permutation of **[8, 9 10]** is allowed in the answer: the **list** could appear as **[8, 9, 10]**, **[8, 10, 9]**, **[9, 8, 10]**, **[9, 10, 9]**, **[10, 8, 9]**, or **[10, 9,8]**; the same for any permutation of **[1, 10, 11, 1]**.

No other division of values from the original **list** yields a smaller difference in **sums**.

The function **closest_sums (ns : [int], l1 : [int], l2 : [int]) : [[int], [int]]** takes three arguments:

- **ns**: a **list** of values (**int**s), which can contain duplicates.
- **l1**: the **list** of values that are already in the first **list**.
- **l2**: the **list** of values that are already in the second **list**.

This function returns a **2-list**: a **list** storing **2 list**s of **int**s. all values in **ns** appear in **one list or the other**, and the sum of the values in each **list** are as close as possible.

Specification: **closest_sums** returns the best (closest sums) division of values in **ns** given the values already in **list**s **l1** and **l2**. We compute the **closest_sums list**s by examining every way to divide values from **ns** into either **l1** or **l2** and then determine which resulting **2-list** leads to the closest sums.

If we use the example above, we will call this function like

```
  closest_sums ( [1, 10, 8, 11, 1, 9, 10],  [],  [] )
```

passing **[]** to **l1** and **l2**, because initially no values from **ns** are in either **list**. In this case, the function call returns **[[1, 10, 11, 1], [8, 9, 10]]** or some other permutation of each **list**, such that the difference in the **sums** is **4**.

Here are a few simpler examples.

1. **closest_sums([8], [], [])** returns **[[8], []]** or **[[], [8]]**,
   regardless of which **list 8** goes into, the difference of the sums would be **8**. An empty **list** sums to **0**.

2. **closest_sums([5,8], [], [])** returns **[[8], [5]]** or **[[5], [8]]**,
   each value must be put in a different **list** so the difference of the sums would be **3**; for any other divisions, **[[], [5,8]]** or **[[5,8], []]**, the difference of the sums would be larger: **13**.

3. **closest_sums([2,5,8], [], [])** returns **[[2,5], [8]]** or **[[8], [2,5]]** or **[[5,2], [8]]** or **[[8], [5,2]]**,
   2 and **5** must be in the same **list** and **8** in the other **list** so the difference of the sums would be **1**; for any other divisions, e.g., **[[2,8], [5]]** or **[[5,8],[2]]**, etc. the difference of the sums would be larger: here **5** and **11**.

For small arguments, like the ones I will test, such a function can compute its result in under a few seconds. **Do not**

mutate any parameters: construct new data structures in recursive calls and/or when combining the results computed by recursive calls.

You must write the **closest_sums** function **recursively**. Do not import/use any other functions from any other modules. Think carefully about how to write this function. Using its specification above (its arguments, result, and their types), think about how to write the base case and how to break down the problem into similar/strictly smaller subproblems and successfully combine the solutions of these recursively solved subproblems. "Its elephants all the way down." See the pictures below, in **Hint 3**,for more detailed information.

Hint 1: For what case can you trivially compute the solution without recurring? Recognize and return the correct value for that case.

Hint 2: For any first value in **ns**, compute the answer based on whether it is better to put that value into the first or second **list**: the only two possibilities. Use the **min** function, similarly to how we have used the **sorted** function, by supplying it with a **key** argument that computes which recursive call is better.

Hint 3: The recursive calls, going downward in the call tree, create every possible distribution of the **ns** values into two **list**s. For example, if **ns = [2, 5, 8]** then we have the following: all ways to distribute **2** at depth 1; all ways to distribute **2** and **5** at depth 2; all ways to distribute **2**, **5**, and **8** at depth 3.

```
                                     []/[]
                     /                                    \
                 [2],[]                                 [],[2]
            /                \                     /              \
      [2,5],[]             [2],[5]             [5],[2]          [],[2,5]
     /        \           /       \           /       \         /        \
[2,5,8],[] [2,5],[8] [2,8],[5] [2],[5,8] [5,8],[2] [5],[2,8] [8],[2,5] [],[2,5,8]]
```

Given the values at the bottom of these call trees (each representing a base case: some distribution of all the values in **ns**) the results going upward select the better distribution of values (combining the results of the solved subproblems).

```
                             [2,5],[8] or [8],[2,5]
                         /                            \
                [2,5],[8]                              [8],[2,5]
              /           \                           /          \
      [2,5],[8]           [2,8],[5]           [5],[2,8]          [8],[2,5]
     /        \            /      \            /      \          /       \
[2,5,8],[] [2,5],[8] [2,8],[5] [2],[5,8] [5,8],[2] [5],[2,8] [8],[2,5] [],[2,5,8]
```

3: Details of shared_dict:

Study the following code and its description to better understand when two names refer to **two objects that are equal** (when **a == b** is **True**) and when two names refer to the **same/unique object** (when **a is b** is **True**).

```
  a = tuple([1,2,3])      # construct a tuple object storing 1, 2, and 3
  b = tuple([1,2,3])      # construct a different tuple object also storing 1, 2, and 3
  c = a                   # make c refer to the same tuple object a refers to
  print(a == b , a is b)  # True (tuples are ==) False (tuples are not the same object)
  print(c == a , c is a)  # True (tuples are ==) True  (tuples are the same object)
```

In this problem you will define a class named **shared_dict**, derived from the **dict** class. **It will operate only on dictionaries whose keys and values are both immutable, so both can be used as keys in other dictionaries**. A **shared_dict** will store its keys and values uniquely: if two different keys are associated to **equal** values (the == comparison is **True**), then they will instead be associated to the **same/unique** object (the **is** comparison is **True**). Writing a special **__setitem__** will ensure this property.

Here is an illustration of the difference between a **dict** and **shared_dict**.

```
sd = shared_dict()
sd['a'] = tuple([1,2,3])    # associate 'a' with one tuple object
sd['b'] = tuple([1,2,3])    # associate 'b' with a different tuple object
print(sd['a'] is sd['b'])  # prints True (would print False for a standard dict)
```

Because **sd** is a **shared_dict**, both keys will be associated with the same/unique **tuple** object; in a **dict** the keys would be associated with different **tuple** objects that are ==.

IMPORTANT: Define **shared_dict** by inheritance, so that it operates as described below, producing the same results as in the examples shown below. By using inheritance, other methods like **__len__** or **__contains__** or any iterations should be inherited and work correctly, without you having to write any code for them. Some calls to these (and other) methods may appear in the testing code for this derived class.

You will write only three methods: **__init__**, **__setitem__**, and **__delitem__**. You should debug the **__setitem__** method before writing **__delitem__**. I have written a **__str__** method in the class, which converts all three dictionaries into a big string; printing the string is useful for debugging (and is actually used in the examples shown below).

1. The **__init__** method should initialize the **shared_dict** and the **_counts** and **_stored_kvs** dictionary attributes it defines.

   When a **shared_dict** is constructed, it is passed no arguments. Besides inheriting from **dict**, it stores two attributes that must be named **_stored_kvs** (a **dict**) and **_counts** (a **dict** or **defaultdict**; I used the latter). For each key and value stored in the **shared_dict**:

   - **_counts** stores a count of how often an object is used as either a key or value (both are counted) in the **shared_dict**.
   - **_stored_kvs** stores as values the unique objects to use in the **shared_dict**: whenever we want to put an object into a **shared_dict** (as either a key or a value), we look up the object (use it as a key) in **_stored_kvs** and use the resulting object found there.

2. The **__setitem__** method should ultimately associate the key argument with the value argument in the **shared_dict**, but which object should be used for the key and which for the value to ensure sharing? Here is the algorithm.

   a. Start with the key object: if it is not already stored as a key in the **_stored_kvs dict**, store it there (using that object as both the key and its associated value).

   b. Use the object in the **_stored_kvs dict** to update the **_counts** dictionary: it associated value is set to **1** if new to **_counts**, or increased by **1** if already there.

   c. Now redo steps a-b for the the value object, possibly also adding it as key and associated value in the **_stored_kvs dict**.

   d. Finally, associate the key object and value object from **_stored_kvs** in the **shared_dict**.

   I have not discussed what must happen in **__setitem__** if the key is already present in the **shared_dict**; don't worry about that case for now: it is not tested here, but is part of the extra credit problem at the end of this exam.

   The following operations mostly print the **shared_dict** information using the **__str__** method written in the class.

```
1. sd = shared_dict()
  # _counts     = {}
  # _stored_kvs = {}
  # shared_dict = {}

2. sd['a'] = tuple([1,2,3])
  # _counts     = {'a': 1, (1, 2, 3): 1}
  # _stored_kvs = {'a': 'a', (1, 2, 3): (1, 2, 3)}
  # shared_dict = {'a': (1, 2, 3)}

3. sd['b'] = tuple([1,2,3])
  # _counts     = {'a': 1, (1, 2, 3): 2, 'b': 1}
  # _stored_kvs = {'a': 'a', (1, 2, 3): (1, 2, 3), 'b': 'b'}
  # shared_dict = {'a': (1, 2, 3), 'b': (1, 2, 3)}

4. print(sd['c'] is sd['b'])
  True

5. sd['c'] = 'c'
  # _counts     = {'a': 1, (1, 2, 3): 2, 'b': 1, 'c': 2}
  # _stored_kvs = {'a': 'a', (1, 2, 3): (1, 2, 3), 'b': 'b', 'c': 'c'}
  # shared_dict = {'a': (1, 2, 3), 'b': (1, 2, 3), 'c': 'c'}

6. sd[tuple([1,2,3])] = 'd'
  # _counts     = {'a': 1, (1, 2, 3): 3, 'b': 1, 'c': 2, 'd': 1}
  # _stored_kvs = {'a': 'a', (1, 2, 3): (1, 2, 3), 'b': 'b', 'c': 'c', 'd': 'd'}
  # shared_dict = {'a': (1, 2, 3), 'b': (1, 2, 3), 'c': 'c', (1, 2, 3): 'd'}

7. sd.verify_shared(trace_all = False) # Set trace_all = True for details of failures
  Verifying equal (==) objects are shared; set trace_all = True for more details of failu
  ---All are correctly shared
```

In English

1. The construction of the **shared_dict** sets attribute dictionaries to be empty.

2. The dictionaries are updated: (1) both the key and value are put into **_stored_kvs** (each is put in both as a key and its associated value), (2) the **_counts** for the key and its value are both set to **1**, and (3) the association appears in the **shared_dict**: both the key and value come from **_stored_kvs**.

3. The dictionaries are updated: (1) the new key is put into **_stored_kvs** (but not the already-present value), (2) the **_counts** for the new key is set to **1** and the **_counts** for the already-present value is incremented to **2** (it is now used a second time), and (3) the association appears in the **shared_dict**: both the key and value come from **_stored_kvs**.

4. The **print**, if you have written your code correctly, the inherited method **__getitem__** should work correctly for all these associations. It should print **True**, because the same/unique **tuple** is associated in the **shared_dict** with the key **'a'** and the key **'b'**.

5. The dictionaries are updated: (1) the key (which is also its value) is put into **_stored_kvs**, (2) the **_counts** for the new key (which is the same as its associated value) is set to **2** (one for its use as key, one for its use as a value), and (3) the association appears in the **shared_dict**: both the key and value come from **_stored_kvs**.

6. The dictionaries are updated: (1) the new value is put into **_stored_kvs** (but not the already-present key), (2) the **_counts** for the already-present key is incremented to **3** (it is now used a third time), and the **_counts** for the new value is set to **1**. (3) the association appears in the **shared_dict**: both the key and value come from **_stored_kvs**.

7. Calling **sd.verify_shared** will indicate whether all values are shared in all the dictionaries, printing an error message identifying any objects that are not correctly shared. For every object not correctly

shared, you will see an error printed, identifying (a) the dictionary, (b) whether the problem is with a key or value, and c) what object is incorrectly stored.

```
ERROR: in _counts key (1, 2, 3) does not refer to the correct object
```

3. The **__delitem__** method should ultimately remove the key argument from the **shared_dict**. But first, it should decrement **_counts** for the key, and if the count goes to 0 (it should never go negative) remove the key from both **_counts** and **_stored_kvs**. It should do these same operations for the value associated with the key being removed.

   The following operations all print the **shared_dict** information using the **__str__** method written in the class.

```
The shared_dict starts as
  # _counts     = {'a': 1, (1, 2, 3): 3, 'b': 1, 'c': 2, 'd': 1}
  # _stored_kvs = {'a': 'a', (1, 2, 3): (1, 2, 3), 'b': 'b', 'c': 'c', 'd': 'd'}
  # shared_dict = {'a': (1, 2, 3), 'b': (1, 2, 3), 'c': 'c', (1, 2, 3): 'd'}

1. del sd['a']
  # _counts     = {(1, 2, 3): 2, 'b': 1, 'c': 2, 'd': 1}
  # _stored_kvs = {(1, 2, 3): (1, 2, 3), 'b': 'b', 'c': 'c', 'd': 'd'}
  # shared_dict = {'b': (1, 2, 3), 'c': 'c', (1, 2, 3): 'd'}

2. del sd['b']
  # _counts     = {(1, 2, 3): 1, 'c': 2, 'd': 1}
  # _stored_kvs = {(1, 2, 3): (1, 2, 3), 'c': 'c', 'd': 'd'}
  # shared_dict = {'c': 'c', (1, 2, 3): 'd'}

3. del sd['c']
  # _counts     = {(1, 2, 3): 1, 'd': 1}
  # _stored_kvs = {(1, 2, 3): (1, 2, 3), 'd': 'd'}
  # shared_dict = {(1, 2, 3): 'd'}

4. del sd[tuple([1,2,3])
  # _counts     = {}
  # _stored_kvs = {}
  # shared_dict = {}
```

   In English

   1. (1) the **_counts** of key **'a'** and its associate value **(1, 2, 3)** are decremented by **1**: the **_counts** of **'a'** becomes **0**, so (2) the key **'a'** is removed from **_counts** and **_stored_kvs**, but not the value **(1, 2, 3)**, because its **_counts** is still **2**: it is still being used as the value associated with key **'b'** and as a key. (3) **'a'** and its association is removed from the **shared_dict**.

   2. (1) the **_counts** of key **'b'** and its associated value **(1, 2, 3)** are decremented by **1**: the **_counts** of **'b'** becomes **0**, so (2) the key **'b'** is removed from **_counts** and **_stored_kvs**, but not the value **(1, 2, 3)**, because its **_counts** is still **1**: it is still being used as a key. (3) **'b'** and its association is removed from the **shared_dict**.

   3. (1) the **_counts** of **'c'** is decremented by **1** twice: once for its use as a key, once as a value; the **_counts** becomes **0**, so (2) **'c'** is removed from **_counts** and **_stored_kvs**, (3) **'c'** and its association is removed from the **shared_dict**.

   4. (1) the **_counts** of key **(1,2,3)** and its associated value **'d'** are decremented by **1**: the **_counts** of both become **0**, so (2) both are removed from **_counts** and **_stored_kvs**, (3) **(1,2,3)** and its association is removed from the **shared_dict**.

Extra Credit: 1 point

The algorithm for the **__setitem__** method described above does **not work** correctly if we are **resetting** an association for a key already in the **shared_dict**; it works only for creating associations with **new** keys. Here is an example that illustrates the problem with this algorithm:

```
sd = shared_dict()
  # _counts     = {}
  # _stored_kvs = {}
  # shared_dict = {}

sd['a'] = tuple([1,2,3])
  # _counts     = {'a': 1, (1, 2, 3): 1}
  # _stored_kvs = {'a': 'a', (1, 2, 3): (1, 2, 3)}
  # shared_dict = {'a': (1, 2, 3)}

sd['a'] = frozenset([1,2])
  # _counts     = {'a': 2, (1, 2, 3): 1, frozenset({1, 2}): 1}
  # _stored_kvs = {'a': 'a', (1, 2, 3): (1, 2, 3), frozenset({1, 2}): frozenset({1, 2})}
  # shared_dict = {'a': frozenset({1, 2})}
```

Note that after the second assignment **_counts** thinks **'a'** appears twice in the **shared_dict** (when it occurs once; it shouldn't be incremented with an old association is updated) and it thinks **(1, 2, 3)** appears once (when it is not in the **shared_dict**; and should not be in the **_stored_kvs dict** either.

You must **modify __setitem__** to correctly produce the following result **without calling __delitem__ or doing the equivalent**. Doing so is overkill: we can fix **__setitem__** to be more efficient. The example above should result in the dictionaries

```
  # _counts     = {'a': 1, frozenset({1, 2}): 1}
  # _stored_kvs = {'a': 'a', frozenset({1, 2}): frozenset({1, 2})}
  # shared_dict = {'a': frozenset({1, 2})}
```