# CS116 Winter 2011 Assignment 8
## Due: Tuesday, March 22 at 10:00AM

The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.

**Important Information:**

1. Read the course Style Guide for Python (Appendix C in the course notes) for information on assignment policies and how to organize and submit your work. In particular, your solutions should be placed in files a8qY.py, where Y is a value from 1 to 5.
2. Be sure to download the interface file to get started.
3. You may use helper functions in your solutions as needed. Simply include them in the same file with your solution, but do not declare them locally. You should follow the full design recipe for your helper functions (which includes testing).
4. If you use print statements for debugging purposes, be sure to remove them (or comment them out) before you submit your solutions.
5. Do not copy the purpose directly from the assignment description – it should be in your own words.
6. You may assume that all consumed data will satisfy any stated assumptions, unless indicated otherwise.

**Language level:** Any Python in the range 2.5.0 to 2.7.1. (Do **not** use Python 3.0 or higher)

**Coverage**: Module 8

**Note:** Once you complete the assignment, you may wish to go to http://hci.uwaterloo.ca/faculty/mterry/cs116_test_songs to download a utility file that can be used to apply your functions to real songs. However, downloading and using the utility file is *completely optional*.

**Overview**
Digital media (e.g., digital audio or digital images) represent real-world, analog data using *samples*. Samples are simply numbers representing a measurement at a point in time. For example, CD audio is created by sampling sound levels 44,100 times per second.

In this assignment, we will be working with samples representing songs. In particular, you will write a series of functions that will allow you to reconstruct one song from the samples (sounds) found in another song. The way we will accomplish this goal is as follows:

- We will have two songs, a *target song*, and a *replacement song*. We will recreate the target song using samples (sound snippets) from the replacement song
- We will first chop both songs up into small pieces, into what we call *SampleBlocks*
- For each SampleBlock in the target song, we will find the closest SampleBlock from the replacement song
- We will then put all of the chosen SampleBlocks from the replacement song together to create a new version of the target song

What is remarkable about this process is that we can create a very close approximation of the target song using only snippets of the replacement song, if we chop both songs into very small pieces (such as 10/44100 of a second). What is even more remarkable is that the replacement song can be nothing like the target song. In fact, it could even be a recording of you and your friends talking.

On the other hand, if you try to recreate the target song using relatively large snippets from the replacement song (say, 5000/44100 of a second), the result will sound almost nothing like the original (though you may hear similar rhythms).

Once you finish the assignment, you can apply your functions to real songs, if you wish (*this is entirely optional*). See http://hci.uwaterloo.ca/faculty/mterry/cs116_test_songs for more information.

**Data Structure Used For This Assignment**
For all questions, you will be working with the following class (defined in the interface file):

```
class SampleBlock:
    'Fields: samples, average_amplitude'
```

The `samples` field is a list of floats, while the `average_amplitude` field is a float (a single number) representing the average value found in `samples`.

Recall from lecture that we can create a new SampleBlock object and assign it values as such:

```
sample_block = SampleBlock()
sample_block.samples = []
sample_block.average_amplitude = 0.0
```

To assist with your debugging, we have included methods for SampleBlock that will convert its values to a string, so you can print them out as you test your code.

**You should copy and paste the class definition that we provide in each assignment file**: *do not import a8interface.py.*

1. In this question, you will write a function

   `chop_samples(float_list, block_size)`

   that consumes a list of floats and a *positive* integer, and produces a list of SampleBlock objects.

   Your function will chop `float_list` into smaller lists, each of length `block_size`. You will assign each of these smaller lists to the `samples` field of a new SampleBlock object.

   The number of values in the `samples` list should be *exactly* `block_size` long. If the number of values in the `float_list` argument is not a multiple of `block_size`, you should not create a SampleBlock to hold the leftover values of `float_list`.

   As you create each new SampleBlock object, you should also calculate the average value of the floats found in the `samples` field and assign the result to the `average_amplitude` field of the SampleBlock.

   You should return a list of these SampleBlock objects in the same order as encountered in `float_list`. If `float_list` is less than `block_size` long, you should return an empty list.

   As an example, calling

   `chop_samples([0.0, 1.0, 2.0, 6.0, 7.0, 8.0, 1.0, 4.0, 1.0, 9.0], 3)`

   will yield a list of 3 SampleBlock objects:

   - The first SampleBlock object will have `[0.0, 1.0, 2.0]` in its `samples` field and `1.0` in its `average_amplitude` field
   - The second SampleBlock will have `[6.0, 7.0, 8.0]` in its `samples` field and `7.0` in its `average_amplitude` field
   - The final SampleBlock object will have `[1.0, 4.0, 1.0]` in its `samples` field and `2.0` in its `average_amplitude` field
   - The final value in `float_list`, `9.0`, will be ignored and not present in any SampleBlock.

   **For this question, *you must use iteration and list slicing*. You *may not* use recursion.**

2. Python lists include a method called `sort`. By default, `sort` uses an equivalent of the built-in function `cmp`, short for “compare” (or “comparator,” or “comparison”). However, you can pass in your own function to `sort` to perform a custom comparison when sorting. The compare function passed to `sort` takes two arguments and should behave as follows:

   - If the first argument is less than the second, it should return -1
   - If the second argument is less than the first, it should return 1
   - Otherwise, it should return 0

   For this question, write a function

   ```
   sort_sample_blocks(sample_block_list)
   ```

   that consumes a list of SampleBlock objects and returns nothing. Your function will have the side effect of sorting `sample_block_list` in ascending order according to the `average_amplitude` values of the SampleObjects. You should mutate this list in-place by using the built-in `sort` method, using a compare function that you write and pass to the method. You may use `lambda` for the compare function, if you wish.

   Example:
   Using the list of SampleBlocks produced in the last question's example (with average amplitudes of 1.0, 7.0, and 2.0), the sorted list of SampleBlocks would then be in the order of 1.0, 2.0, and 7.0. In code:

   ```
   block_1 = SampleBlock()
   block_1.samples = [0.0, 1.0, 2.0]
   block_1.average_amplitude = 1.0

   block_2 = SampleBlock()
   block_2.samples = [6.0, 7.0, 8.0]
   block_2.average_amplitude = 7.0

   block_3 = SampleBlock()
   block_3.samples = [1.0, 4.0, 1.0]
   block_3.average_amplitude = 2.0

   blocks = [block_1, block_2, block_3]
   sort_sample_blocks(blocks)

   # => sorts “blocks” in place, with the new order of the blocks being
   # block_1, block_3, and block_2 in the list “blocks”
   ```

3. For this question, you will use a variation of binary search (as discussed in Module 8). Specifically, write a function

   ```
   find_closest_sample_block_index(target_sample_block,
                                   sorted_list_of_sample_blocks)
   ```

   that consumes a SampleBlock and a non-empty sorted list of SampleBlocks, and produces an integer.

   Your function should return the *index* of the SampleBlock object in the `sorted_list_of_sample_blocks` with an `average_amplitude` that is *closest* to the `average_amplitude` of the `target_sample_block` passed in. Importantly, we define "closest" as the smallest absolute difference between their `average_amplitude` values. As the name implies, `sorted_list_of_sample_blocks` will be sorted in ascending order according to the `average_amplitude` values of the SampleBlocks.

   Note that there may not be a SampleBlock in `sorted_list_of_sample_blocks` with an `average_amplitude` exactly the same as the one in `target_sample_block` (but if there is, your function should return that SampleBlock's index when it is found, rather than continuing the search). If an exact match is not found, you should return the index of the SampleBlock closest to the final index considered by the binary search, with the closest `average_amplitude`. If there are two SampleBlocks around the final index examined that have `average_amplitude`s equally close to the `target_sample_block`, you should choose the one with the lower index. For example, if you were searching for a `target_sample_block` with `average_amplitude` of 1.5, but there were only two SampleBlocks in the list with `average_amplitude`s of 1.0 and 2.0, you would return the index of the SampleBlock with the `average_amplitude` of 1.0.

   For this question, you may use either recursion or iteration to solve the problem, but you *may not use linear search* to find the closest match – you must use binary search. Consequently, solutions that iterate through the entire list or use abstract list functions such as map or filter will receive no marks.

   As an example, given a sorted list of three SampleBlocks, with `average_amplitude` values of 1.0, 3.0, and 5.0, your function would return:

   - 0 if the `target_sample_block` passed in had an `average_amplitude` of 0.9
   - 0 if the `target_sample_block` passed in had an `average_amplitude` of 2.0
   - 1 if the `target_sample_block` passed in had an `average_amplitude` of 2.5

   *Hint*: Make sure you write test cases for the various boundary cases to ensure your function works correctly for all input.

4. Write a function

   ```
   calculate_sum_square_difference(left_sample_block, right_sample_block)
   ```

   that consumes two SampleBlock objects (with equal-length `samples` fields) as input and produces a float as output. The function should calculate the sum of the pairwise squared differences between each value in the left and right SampleBlocks' `samples` fields.

   For example, given a `left_sample_block` with `samples` values `[1.0, 2.0, 3.0]`, and a `right_sample_block` with `samples` values `[9.0, 5.0, 1.0]`, your function will produce $(1.0-9.0)^2 + (2.0-5.0)^2 + (3.0-1.0)^2$, or 77.0.

5. For the final piece, write a function

   ```
   find_best_match(target_sample_block,
                   list_of_replacement_sample_blocks,
                   similarity_function,
                   start_index,
                   num_neighbours)
   ```

   that consumes a SampleBlock, a (non-empty) list of SampleBlock objects, a similarity function (described below), and two integers (`start_index` and `num_neighbours`). The function will produce a SampleBlock object. `start_index` will satisfy the following constraint:

   ```
   0 <= start_index < len(list_of_replacement_sample_blocks)
   ```

   Your function will examine all SampleBlocks from (`start_index - num_neighbours`) to (`start_index + num_neighbours`), *inclusive,* without going off either end of the list, to find the SampleBlock closest to `target_sample_block`, as measured by `similarity_function`. As an example, if `start_index` were 3 and `num_neighbours` was 2, indices 1, 2, 3, 4, and 5 would be examined.

   Across this range, your function should return the SampleBlock closest to `target_sample_block` (rather than the index of the block). If there are multiple SampleBlocks equally close to `target_sample_block` as measured by the `similarity_function`, you should return the SampleBlock found earliest in the list (i.e., the one with the smallest index).

   `similarity_function` is a function with the following contract:

   ```
   SampleBlock, SampleBlock → float[>=0]
   ```

   The `similarity_function` is guaranteed to return a non-negative number, and the lower the number it returns, the more similar the SampleBlocks.

   For this question, you *must* use iteration; you may not use recursion.

As an example:

```
target_block = SampleBlock()
target_block.samples = [1.0, 2.0, 3.0]
target_block.average_amplitude = 2.0

replacement_block_1 = SampleBlock()
replacement_block_1.samples = [0.0, 0.0, 1.0]
replacement_block_1.average_amplitude = 1.0/3

replacement_block_2 = SampleBlock()
replacement_block_2.samples = [1.0, 1.0, 2.0]
replacement_block_2.average_amplitude = 4.0/3

replacement_block_3 = SampleBlock()
replacement_block_3.samples = [2.0, 3.0, 4.0]
replacement_block_3.average_amplitude = 9.0/3

replacement_blocks = [replacement_block_1,
                      replacement_block_2,
                      replacement_block_3]

def block_similarity(left_sample_block, right_sample_block):
    diff_in_amplitude = left_sample_block.average_amplitude - \
                        right_sample_block.average_amplitude
    return diff_in_amplitude * diff_in_amplitude


find_best_match(target_block, replacement_blocks, block_similarity, 1, 2)
# => Will return the object representing replacement_block_2 from above
```

In the example above, we define a function, `block_similarity`, to illustrate how you could use and test `find_best_match`. In practice, you would use `calculate_sum_square_difference` (the function created in question 4) to get better results in finding a best match for a given SampleBlock.