Interfacing With The Genetic Algorithm Main File

General Overview of main.py and it's various sub-sections, each subsection in the code is denoted by a comment of the same name:

1. First time setup
    a. Set the random seed
    b. Generate the list of initial phenotypes
    c. The original code also clears necessary output files in this section; however, these steps are optional
2. Saving and exiting
    a. Save the pheno list to a file
    b. Exit the program
3. Post grading setup
    a. Load list of graded phenotypes
    b. Assign grades to phenotypes
    c. Order according to fitness
    d. Assign crossover probabilities
4. Calculation and output of vital statistics
    a. This section of code is largely implementation specific
    b. First samples are taken from the population
    c. Fitness info is logged
    d. And more statistics are outputted to a file, from which they will be graphed
5. Pre-crossover setup
    a. Establish a new list for new phenotypes, generated during crossover
    b. Save the best phenotype in a generation (optional, but recommended)
    c. Convert the pheno list into a Biased Random Sequence, which is used to randomly draw phenotypes, with a bias towards their crossover probability
6. Crossover/Mutation loop
    a. This loop runs for population - 1 interactions (because of the fact that the best song is always saved)
    b. Phenotype selection and crossover
        i. Draw to phenotypes from the previous generation randomly
        ii. Determine the point at which they will crossover
        iii. Perform crossover
        iv. Log the crossover
    c. Mutation phase
        i. Determine if Mutation will occur at all
        ii. Determine what kind of mutation
        iii. Mutate, and log mutation
7. Saving and exiting (see above)

Introduction:

The main.py file is a simple yet effective way to run genetic computations on a user defined data structure in python 3. Interfacing with it is mostly a matter of defining proper functions, all of which are defined in the main.py.

Required Functions:

Get_crossover_prob
Get_gen_num
Read_input_file_line
Get_pheno_list
Save_pheno_list_to_file
Log_cross_over
Delete_mutation
Add_mutation
mutation
Log_delete_mutation
Log_add_mutation
Log_mutation
Find_min_pheno
Get_pheno_count
Load_pheno_list
Meta_gen

Get_crossover_prob:

> Typical Inputs: None.
> Outputs: The probability that a phenotype will propagate.
>
> More Info: The inputs will vary depending how the crossover probability is computed.

Get_gen_num:

Typical Inputs: A phenotype and the config file object.
Outputs: The generation number.

More Info: The inputs can vary depending on the user defined data structure, in this case a phenotype song_id property, contains the generation number.

Read_input_file_line:

Typical inputs: A file object.
Outputs: An integer value, which is usually the score of a specific phenotype.

More information: This function is usually used to accept scores generated by another program.

Get_pheno_list:

Typical inputs: The config file object, as defined in ConfigFile.py
Outputs: A list of phenotypes.

More information: This function is usually used to generate a new list of randomly generated phenotypes for initialization.

Save_pheno_list_to_file:

Typical inputs: A list of phenotypes and the config file object.
Outputs: A file, containing the string representation of a list of phenotypes.

More information: In the first implementation, this function also notifies the parent program, which called it, if a pid has been provided in the command line arguments.

Log_cross_over:

Typical inputs: The two phenotypes involved and the crossover point.
Outputs: A record of the crossover for logging purposes.

Delete_mutation:

Typical inputs: A phenotype, and a randomizer object as defined in GeneticRandomizer.py
Outputs: A phenotype, with a random portion deleted.

More information: This function can be omitted, and is best used when the size of your data structure can become a problem for fitness.

Add_mutation:

Typical inputs: A phenotype and the config file object.
Outputs: Modified phenotype with an element added.

More Info: The config file is used to determine how the phenotype is expanded.

mutation:

Typical inputs: A phenotype and the config file object.
Outputs: A modified phenotype.

More Info: The config file is used to determine how the phenotype is modified.

Log_delete_mutation:

Typical Inputs: A phenotype.
Outputs: A record of the mutation for logging purposes

More Info: See Delete_mutation for more info

Log_add_mutation:

Typical Inputs: A phenotype.
Outputs: A record of the mutation for logging purposes.

More Info: See Add_mutation for more info.

Log_mutation:

Typical Inputs: A phenotype.
Outputs: A record of the mutation for logging purposes.

More Info: See mutation for more info.

Find_min_pheno:

Typical Inputs: A phenotype list and the config file object.
Outputs: The phenotype with the smallest score attached.

More Info: In the first implementation of this function, a higher score means a lower fitness. This function ensures that at least one phenotype, with the lowest score, will be passed on to the next generation.

Find_max_pheno:

Typical Inputs: A phenotype list and the config file object.
Outputs: The phenotype with the largest score attached.

More Info: In the first implementation of this function, a higher score means a lower fitness. This function ensures that at least one phenotype, with the largest score, will be passed on to the next generation.

Get_pheno_count:

Typical Inputs: The config file object.
Outputs: The number of phenotypes in a generation.

More Info: The count is based on the parameters set in the configuration file.

Load_pheno_list:

Typical Inputs: The name of the file being read, and the config file object
Outputs: A list of phenotypes.

More Info: This function is the counterpart to save_pheno_list_to_file.

Meta_gen:

Typical Inputs: The config file object.
Outputs: Metadata (data note directly accessible in the user defined data structure, which still participates in mutation)

More Info: This function is optional, but will have to be removed from main.py

See main.py for more implemenation details.

GeneticRandomizer:

GeneticRandomizer.py containes useful methods for randomization of the Genetic Computation process, see the code for more details.

Config File:

To instantiate an instance of the config file object, you need to first import the config file constructor from ConfigFile.py:

From ConfigFile import ConfigFile

The config file is written in plain text and has the following format:

<field name, each word separated by spaces>: <integer or string value>

The config file parser which is used by default, can only parse integers and strings, and nothing else

An example of a config file is:

Min score: 4
Max score: 15
File name: my_file.txt

These fields can then be accessed using the following code:

Con_file = ConfigFile(“your config filename here”)

Then max score can be access using this code:

Con_file.Max_score

Where the result is 15. Note that every space in the .config file is replaced by an underscore in the config file object. As such the rest of the methods can be accessed in the following way:

Con_file.Min_score
Con_file.File_name

This file is a convenient way to pass vital statistics to functions and objects, which are subject to frequent change.
The provided config file (pyth_main.config), contains the following fields by default:

Save file
Input file
Song count (aka pheno count)
Chromo delete prob (see delete_mutate)
Chromo add prob (see add mutate)
Max step size
Mutation chance
Div scale factor (used if diversity is part of your grading scheme)
Score scale factor (used to counterbalance the diversity score)

See ConfigFile.py for more information.

User-Defined Phenotype Interface:

List of required methods:

- __eq__
- __repr__
- __str__
- __copy__
- __getitem__
- __len__
- Crossover
- Mutate
- Chromosome_delete
- Chromosome_add
- flatten

__eq__:

Inputs: Another object .
Outputs: A boolean value.

More Info: The method that determines equality between different instances of the phenotype.

__repr__:

Inputs: None.
Outputs: A string.

More Info: The method that defines how an instance of the phenotype is displayed in the python interpreter.

__str__:

Inputs: None.
Outputs: A string.

More Info: The method that defines how an instance of the phenotype is converted into a string.

__copy__:

Inputs: None
Outputs: A new instance of the phenotype with a copy of all of the original's data.

More Info: The method that defines how a copy of an instance of the phenotype is created.

__getitem__:

Inputs: The desired index.
Outputs: The requested element.

More Info: The method that defines how an element of the phenotype is accessed in the code.

__len__:

Inputs: None.
Outputs: The length of this instance of the phenotype.

More Info: The method that returns the length of the phenotype.

crossover:

Inputs: Another instance of the phenotype, and the index at which the structures will be crossed-over.
Outputs: A new instance of the phenotype.

More Info: The crossover produces a new phenotype which is comprised of the first portion of the original phenotype(up to the cross-over index) and the latter portion of the second, passed-in phenotype(after the crossover index).

mutate:

Inputs: A “Delta mask”
Outputs: The mutated phenotype

More Info: This method defines the mutation process based on a “Delta Mask” input, which contains the mutation information for each element of the flattened phenotype.

chromosome_delete:

Inputs: The index of the chromosome to be deleted.
Outputs: The phenotype without the deleted chromosome.

More Info: How a chromosome is defined depends on the user’s definition of the phenotype.

chromosome_add:

Inputs: The chromosome to be added.
Outputs: The phenotype with the added chromosome.

More Info: The location the chromosome added, and the definition of a chromosome, is determined in the implementation.

flatten:

Inputs: None
Outputs: The flattened version of the phenotype.

More Info: Should walk through the user's definitions of chromosomes and genes to produce the flat version. This method is not mandatory, but it is useful.

Demo:

To successfully operate the demo version, download the GA_Code directory. Open the config file (which resides in GA_Code, and is referred to as pyth_main.config), ignore most of the other parameters except for “song count” and two others. Change “song count” to something manageable, like 3. Next, change “score scale factor” to 1 and “div scale factor” to 0, if they have not been set to those values already. After that, run one of the following commands, in the working directory, with GA_Code in it:

Python3 GA_Code/main.py -n

Or

Pythone GA_Code/main.py -n -s 1999

The following files should show up in your current working directory:

Main_py_output
Main_py_output_genetic
Main_log.txt

The first file contains a list of songs in a format, which in the first implementation, was used to share phenos created by main.py, with code written in c. Of main_py_output is as follows:

<song count for entire file>
<number of tracks>
<tempo of the song>
<song id>
<number of notes in the next track>
<track id> <volume>
<a list of notes>

Main_py_output_genetic, is used for song persistence within main.py, it has the following format:

<song id>
<tempo>
<track id>
<volume>
<a list of notes>

Note that songs are separated by '*' in this implementation, and tracks are separated by a space.

Save main_py_output_genetic as something else (ex. main_py_output_iteration_1).

Next, create a file called main_py_input, and populate it with pairs of 0 and scores of your choice, for each "song count" number of songs.

Ex.

1 0
2 0
3 0

The left number is diversity and the right number is score, where a high score is bad and a low score is good.

Following that, execute this command in the same directory as GA_Code:

Python3 GA_Code/main.py

Your songs have now been scored and have replicated according to the their scores. Check main_log.txt to verify that the GA has run successfully, if so, the log will be populated with entries detailing the operation of the GA, which should something like the example below:

INFO:root:avg fitness: 2, max fitness: 3, min fitness: 1
INFO:root:phenotypes 1 and 3 with fitnesses 2 1, crossed over, at 14
INFO:root:phenotypes 1 and 3 with fitnesses 2 1, crossed over, at 10
INFO:root:phenotype 5, was mutated

Note that the GA will sample songs from each generation, you can ignore these files as they will mostly largely be a duplicate of main_py_output. Another file which will show up is ave_fitness_graph, which contains some of the same information that is found in the log file.

If you examine the previous song file (main_py_output_genetic_iteration_1), you may notice small changes in the songs due to crossover and mutation.