

Fitness Functions

Evaluation of individuals in PonyGE2 is carried out by the specified fitness function. All fitness functions are located in `src/fitness`. Fitness functions in PonyGE2 must be a class instance contained in its own separate file.

NOTE that fitness function classes in PonyGE2 **must** have the same name as their containing file. For example, a fitness function class titled "my_ff" **must** be contained in a file called "my_ff.py". Doing otherwise will produce an error.

The Base Fitness Function Class

All fitness functions in PonyGE2 inherit from a main base fitness function class, contained in `fitness.base_ff_classes.base_ff`. Using object inheritance allows for various checks and balances to be implemented in the base class, which makes writing new fitness functions a more streamlined and easier process.

NOTE that while all example fitness functions in PonyGE2 inherit from the `fitness.base_ff_classes.base_ff` fitness function class, it is not strictly necessary that any new fitness functions do so. However, it is strongly recommended that new fitness functions follow this convention.

Inheriting from the `fitness.base_ff_classes.base_ff` fitness function class has a number of benefits:

1. The base fitness function class defines the `[[default fitness|Evaluation#default-fitness]]` correctly as `NaN`. Any fitness function that inherits from the base class automatically inherits the correct default fitness value, and as such does not have to implement/define the default fitness.
2. The base fitness function class defines the `[[fitness maximisation/minimisation|Evaluation#fitness-maximisationminimisation]]` attribute as `False`. Any fitness function that inherits from the base class is automatically set up to minimise fitness values, and as such does not have to implement/define the `maximise` attribute if this is the aim of the new fitness function.

The base fitness function class defines the framework for any new fitness function which seeks to inherit from the base class. There are three methods defined in the base class:

1. `__init__(self)`

The `__init__()` method contains all code to be executed during the initialisation of the fitness function. This typically includes the definitions of class attributes or instance variables such as `[[default fitness|Evaluation#default-fitness]]` or `[[fitness maximisation/minimisation|Evaluation#fitness-maximisationminimisation]]`.

2. `__call__(self, ind, **kwargs)`

The `__call__()` method of a fitness function class is called when an individual is evaluated. Fitness functions in PonyGE2 are called automatically from `representation.individual.Individual.evaluate`. In the base fitness function class, the `__call__()` method contains a number of exception handling clauses designed to catch a number of pre-determined errors (see the `[[default fitness|Evaluation#default-fitness]]` section for more

information). The `__call__()` method of the base fitness function class directly calls the `evaluate()` method of the class.

3. `evaluate(self, ind, **kwargs)`

The `evaluate()` method of a fitness function contains the code that is used to directly evaluate the phenotype string of an individual and return an appropriate fitness value.

Inheritance from the Base Class

Through the magic of object inheritance, any new fitness function which inherits from the base fitness function class only needs to define **two** of the above three methods:

1. A new `__init__(self)` method. As with the base class, this method can be used to define class attributes or instance variables, and can also be used to execute potentially computationally expensive code such as the importing/reading of `[[dataset|Evaluation#datasets]]` files.

NOTE that any new fitness function which inherits from the base fitness function class **must** initialise the `super()` class (i.e. the `fitness.base_ff_classes.base_ff` from whence inheritance comes) in the `__init__()` of the new fitness function with the following line of code:

```
super().__init__()
```

See all PonyGE2 fitness functions for examples of this.

2. A new `evaluate(self, ind, **kwargs)` method. A skeleton `evaluate()` method has been made available in the base fitness function class, for the sole explicit function of being over-written by any class which inherits from the base class. Defining a new `evaluate()` method in a new fitness function will over-write the existing `evaluate()` method in the base class, but will still retain the `__call__()` method of the base class. Thus, when the call to evaluate an individual is made (from `representation.individual.Individual.evaluate`), the original `__call__()` method of the base fitness function class is first called, and the new `evaluate()` method of the new fitness function is then called from within the `__call__()` method.

NOTE that any new fitness function which inherits from the base fitness function class and which over-writes the base `evaluate()` method **must** use the same method signature as the original base `evaluate()` method, i.e. the definition of the `evaluate()` method in any fitness function which inherits from the base class **must** be:

```
def evalaute(self, ind, **kwargs):
```

Doing otherwise will produce an error.

NOTE that any new fitness function which **does not** inherit from the base fitness function class **does not** require an `evaluate()` method to be defined.

Any new fitness function which inherits from the base class **does not** need to define a new `__call__()` method. Doing so will over-write the original `__call__()` method of the base class, and will essentially render inheritance from the base class useless, as all exception handling checks from the base class will be over-written. This is why a dedicated skeleton `evaluate()` method has been written for the base class.

The Fitness Function Template

To make the process of writing new fitness functions as easy as possible, a blank fitness function template for new implementations has been included with PonyGE2. This template can be found at `fitness.base_ff_classes.ff_template`. The fitness function template is set up with all necessary code, including an optional overwrite for `[[fitness maximisation/minimisation|Evaluation#fitness-maximisationminimisation]]`, necessary initialisation of the `super()` base fitness function class (from whence the template inherits), and a basic definition of the necessary `evaluate()` function. All that is required to write a new fitness function using this template is to create a copy of the template, name it appropriately (remember to give the file and class the same name), and insert the necessary code for evaluating the phenotype of an individual into the `evaluate()` function. See all PonyGE2 fitness functions for examples of this.

For more information on *how* to evaluate/execute an individual phenotype, see the Grammars section on `[[evaluating -vs- executing phenotypes|Grammars#special-grammars-expressions-vs-code]]`

NOTE that the fitness function template does **NOT** include a definition of a `__call__()` method, as doing so would over-write the same method from the base class.

It is strongly encouraged for users to base new fitness functions from this template.

Default Fitness

The use of a "default fitness" is used in PonyGE2 for a number of reasons:

1. With traditional GE, `[[invalid individuals|Representation#invalid-individuals]]` are individuals which cannot produce a valid phenotype string from their given genome. Since they do not have a valid phenotype, they cannot be executed/evaluated correctly, and as such can not be assigned a fitness value in the regular way. Note that this does not occur with the use of `[[context-aware "intelligent" operators|Representation#context-aware-intelligent-operations]]`.
2. Grammars can often produce large individuals, or individuals with complex behaviour. While valid phenotype strings may be generated, in some cases (such as nested exponential terms, or division by zero), these phenotypes may not evaluate correctly and may raise Exceptions or Errors. Examples of such errors include `FloatingPointError`, `ZeroDivisionError`, `OverflowError`, or `MemoryError`. These individuals are also counted as "runtime_error" individuals in the stats.

Since all individuals require *some* fitness value to be assigned to them, this is accomplished in the above cases through the use of a "default" fitness value. The default "default fitness" value in the base PonyGE2 fitness function class is `NaN`. This allows for easy filtering during the statistical gathering process, to ensure that invalid individuals are not so included.

Since a default fitness value is defined in the `[[base fitness function class|Evaluation#the-base-fitness-function-class]]`, any fitness function which inherits from the base class does not need to define a new default

fitness value. However, any newly implemented fitness function which **does not** inherit from the base class **must** define a `default_fitness` attribute.

NOTE that the use of "default fitness" in PonyGE2 is distinct from the concept of a fitness "penalty", as might be found in e.g. constrained optimisation problems.

Fitness Maximisation/Minimisation

A key concept in any optimisation technique is the notion of a fitness gradient, which allows solutions to be ranked in binary terms of "better" or "worse" on any single given objective. The definition of how one single objective fitness value is considered "better" or "worse" than another fitness value is done through maximisation or minimisation.

Put simply, if the objective of a single given fitness function is to maximise fitness, then given two different individuals with valid (i.e. not `[[default fitness][Evaluation#default-fitness]]`) fitness values, the individual with the greater fitness value will be deemed the "better" solution. Conversely, if *minimisation* is the goal of the fitness function, then the individual with the lowest fitness value will be deemed the "better" of the two.

Fitness maximisation or minimisation is handled in PonyGE2 by the `maximise` attribute of the fitness function. If the `maximise` attribute is set to `True`, then the fitness function will seek to *maximise* fitness. Conversely, if the `maximise` attribute is set to `False`, then the fitness function will seek to *minimise* fitness. Fitness comparison between two individuals is done through the definition of a customised `__lt__(self, other)` function in the `representation.individual.Individual` class.

Since the `maximise` attribute is set to `False` in the `[[base fitness function class][Evaluation#the-base-fitness-function-class]]`, any fitness function which inherits from the base class is automatically set up to **minimise** fitness by default. However, if you are either seeking to **maximise** fitness, or if your newly implemented fitness function **does not** inherit from the base class, you **must** define a `maximise` attribute in the new fitness function. The `[[fitness function template][Evaluation#the-fitness-function-template]]` contains the code to overwrite the base maximisation/minimisation attribute should you so desire to use it.

Additional Functionality

Dynamic ranging of variables within the grammar

As detailed in the section on `[[variable ranges in grammars][Grammars#variable-ranges-in-grammars]]`, it is possible to set a specific `n_vars` attribute in the fitness function, which can then be accessed and used in the grammar through the use of `GE_RANGE:dataset_n_vars`. This can be especially useful for cases where a dataset has a large number of input variables. Traditional grammar parsers would require each variable to be explicitly mentioned in the grammar, e.g.:

```
<vars> ::= x[0] | x[1] | x[2] | x[3] | x[4] | x[5] | ...
```

For datasets with a high number of variables, this can get quite tedious, and can be difficult to read. The use of the `n_vars` attribute in the fitness function allows you to set the number of variables in the dataset. In our fitness function, we would define:

```
self.n_vars = [THE DESIRED NUMBER OF VARIABLES]
```

The `<vars>` line from the grammar above could then be replaced with:

```
<vars>    ::= x[<varidx>]
<varidx> ::= GE_RANGE:dataset_n_vars
```

See `fitness.supervised_learning.supervised_learning` and the `supervised_learning/supervised_learning.bnf` grammar for a working usage example.

Evaluating on Training and Test Data

For fitness functions which evaluate solutions on either training or test [[datasets|Evaluation#datasets]] (e.g. supervised learning problems), PonyGE2 can automatically evaluate the best solution at the end of the evolutionary run on unseen "test" data. This is done through the use of a `training_test` attribute in the fitness function, which PonyGE2 checks for during the stats collecting process in `stats.stats.get_soo_stats` or `stats.stats.get_moo_stats`.

PonyGE2 switches between evaluation on training or test data with the use of an optional input argument of `"dist"`. By default, the `dist` argument is set to `"training"`. However, for test evaluation, the input argument is changed to `"test"`. Optional input arguments for PonyGE2 fitness functions are handled through the `**kwargs` input. In order to set a default input flag for `"dist"` to `"training"`, the following line of code is required in the `evaluate()` method of the fitness function:

```
dist = kwargs.get('dist', 'training')
```

This line defines a `dist` attribute for the fitness function, which can then be used in the `evaluate()` method to automatically set the training or test dataset upon which evaluation is to take place. An example of how this can be achieved is provided below:

```
if dist == "training":
    # Set training datasets to be used.
    x = self.training_in
    y = self.training_exp

elif dist == "test":
    # Set test datasets to be used.
    x = self.test_in
    y = self.test_exp
```

Note that the above example is contingent on `self.training_in`, `self.training_exp`, `self.test_in`, and `self.test_exp` variables being defined and set in the `__init__()` method of the fitness function. See

`fitness.supervised_learning.supervised_learning` for a working usage example.

Error Metrics

Some supervised learning fitness functions require an error metric (e.g. mean-squared error, or `mse`) to be specified. While the default `regression` and `classification` fitness functions provided in PonyGE2 have their error metrics set to `rnse` and `f1_score` respectively by default, it is possible to specify new error metrics with the argument:

```
--error_metric [ERROR_METRIC_NAME]
```

or by setting the parameter `ERROR_METRIC` to `[ERROR_METRIC_NAME]` in either a parameters file or in the params dictionary, where `[ERROR_METRIC_NAME]` is a string specifying the name of the desired error metric. A list of currently implemented error metrics is available in `utilities.fitness.error_metric`.

NOTE that for some supervised learning problems that use specified error metrics (e.g. mean-squared error), these error metrics have their own `maximise` attributes. This `maximise` attribute is automatically used by the specified supervised learning fitness function in place of the standard `[[maximise|Evaluation#fitness-maximisationminimisation]]` attribute.

Datasets

Some supervised learning fitness functions may require a dataset across which to be evaluated. Datasets for PonyGE2 are saved in the `datasets` folder. Most supervised learning problems require two datasets: training and test data. These are specified independently in order to allow for problems to be run only on training data (i.e. if no test dataset is specified, the best individual at the end of a run will *not* be evaluated on unseen test data).

Training datasets can be specified with the argument:

```
--dataset_train [DATASET_NAME]
```

or by setting the parameter `DATASET_TRAIN` to `[DATASET_NAME]` in either a parameters file or in the params dictionary, where `[DATASET_NAME]` is a string specifying the full file name including file extension of the desired training dataset.

Testing datasets can be specified with the argument:

```
--dataset_test [DATASET_NAME]
```

or by setting the parameter `DATASET_TEST` to `[DATASET_NAME]` in either a parameters file or in the params dictionary, where `[DATASET_NAME]` is a string specifying the full file name including file extension of the desired testing dataset.

If using any of the tree example supervised learning fitness functions (`supervised_learning.supervised_learning`, `supervised_learning.regression`, or `supervised_learning.classification`), datasets are automatically loaded by the `supervised_learning.supervised_learning` fitness function class using the `utilities.fitness.get_data.get_data` function. If implementing a new fitness function, then this `get_data` function should be called during the `__init__()` method of the fitness function:

```
self.training_in, self.training_exp, self.test_in, self.test_exp = \
    get_data(params['DATASET_TRAIN'], params['DATASET_TEST'])
```

NOTE that when using training and test datasets for evaluation, you **must** include a `self.training_test = True` attribute in the fitness function, as detailed [\[\[above|Evaluation#evaluating-on-training-and-test-data\]\]](#).

NOTE that you **must** specify the file extension when specifying the dataset names.

While it is recommended that supervised learning problems implement training *and* unseen testing data, it is not necessary to use testing data with these problems. If you wish to run PonyGE2 with no test dataset, you can simply use the argument:

```
--dataset_test None
```

NOTE that if no testing dataset is being used, there is no need to set a `training_test` attribute in the fitness function. This can be done automatically in the `__init__()` method of the fitness function with the following check:

```
if params['DATASET_TEST']:
    self.training_test = True
```

Dataset Delimiters

By default, PonyGE2 will try to automatically parse specified datasets using the following set of data delimiters in the following order:

1. "\t" [tab]
2. "," [comma]
3. ";" [semi-colon]
4. ":" [colon]

If the data cannot be parsed using the above separators, then PonyGE2 will default to using whitespace as the delimiter for separating data (a warning will be printed in this case). However, it is possible to directly specify the desired dataset delimiter with the argument:

```
--dataset_delimiter [DELIMITER]
```

or by setting the parameter `DATASET_DELIMITER` to `[DELIMITER]` in either a parameters file or in the params dictionary, where `[DELIMITER]` is a string specifying the desired dataset delimiter.

NOTE that you **do not** need to escape special characters such as `"\t"` with `"\\t"` when passing in the `--dataset_delimiter` argument from the command line. Simply specify the raw desired string.

Targets

Some fitness functions may require a target value. For example, the `string_match` fitness function requires a target string to match. Target strings can be specified with the argument:

```
--target [TARGET_STRING]
```

or by setting the parameter `TARGET` to `[TARGET_STRING]` in either a parameters file or in the params dictionary, where `[TARGET_STRING]` is a string specifying the desired target string. Once set, the `TARGET` string can be accessed throughout PonyGE2 via the `params` dictionary.

NOTE that all target values will be stored in PonyGE2 as a string by default. If a fitness function requires any data structure other than a string, the target string itself must be parsed to the desired data structure within the fitness function itself.

NOTE that if this parsing of the target string is done in the `__init__()` call of the fitness function, it only needs to be done once rather than at every fitness evaluation.

Specifying Fitness Functions

Fitness functions can be specified with the argument:

```
--fitness_function [FITNESS_FUNCTION_NAME]
```

or by setting the parameter `FITNESS_FUNCTION` to `[FITNESS_FUNCTION_NAME]` in either a parameters file or in the params dictionary, where `[FITNESS_FUNCTION_NAME]` is a string specifying the name of the desired fitness function.

Multiple Objective Optimisation

Multiple objective optimisation (MOO) in the form of the Non-Dominated Sorting Genetic Algorithm 2 (NSGA2) [[[Deb et al., 2002|References]]] has been included with PonyGE2. The developers of PonyGE2 have sought to implement multiple objective optimisation in the simplest and most intuitive way possible.

The core concept of how MOO is implemented in PonyGE2 requires a single fitness function for each desired objective fitness. Each individual fitness function is then a standalone fitness function (i.e., PonyGE2 should be capable of being used with these fitness functions in a normal single-objective manner) with all of the structure and attributes of a standard fitness function (as detailed [[above|Evaluation#the-base-fitness-

function-class]]). The user can simply then specify all desired fitness functions to be used, and PonyGE2 will automatically use NSGA2.

Example Problems

Two example problems are provided:

1. Regression with bloat control

This example runs the standard Vladislavleva-4 [\[\[regression|Example-Problems#regression\]\]](#) example, but with a second fitness function that aims to minimise the number of nodes in the derivation tree. To try this problem, specify the following command-line argument:

```
--parameters moo/regression_bloat_control.txt
```

2. An example benchmark multi-objective optimisation problem from [\[\[Zitzler, et al.|References\]\]](#) is provided. To try this problem, specify the following command-line argument:

```
--parameters moo/zdt1.txt
```

Specifying Multiple Fitness Functions

Multiple fitness functions are specified in PonyGE2 in the same manner that single fitness functions are specified.

Command Line

If using the command line, fitness functions can be specified with the argument:

```
--fitness_function [FITNESS_FUNCTION_1_NAME] [FITNESS_FUNCTION_2_NAME] ...
```

where [\[FITNESS_FUNCTION_1_NAME\]](#), [\[FITNESS_FUNCTION_1_NAME\]](#), etc are the names of the desired individual fitness functions.

NOTE that when specifying multiple fitness functions from the command line it is **not** necessary to use commas or any separators between the various fitness functions. Equally, it is **not** necessary to use any form of bracketing when specifying multiple fitness functions from the command line. The command line parser of PonyGE2 handles this automatically.

Parameters File / Params Dictionary

Unlike with the command line, when specifying multiple fitness functions from either the [algorithm.parameters.params](#) dictionary or from within a parameters file, it is necessary to place the names of each desired fitness function into a list and to use a comma to delineate between individual fitness functions. An example of this would be:

```
FITNESS_FUNCTION: [[FITNESS_FUNCTION_1_NAME], [FITNESS_FUNCTION_2_NAME]]
```

where `[FITNESS_FUNCTION_1_NAME]`, `[FITNESS_FUNCTION_1_NAME]`, etc are the names of the desired individual fitness functions.

NOTE that when specifying multiple fitness functions from either the `params` dictionary or in a `parameters` file that it **is** necessary to place the desired fitness functions in a list, i.e. with bracketing and commas.

Example parameters files for multi-objective optimisation problems can be seen in the `parameters/moo` folder.

Multi-Objective Fitness Function Implementation

PonyGE2 handles the use of multiple fitness functions through a dedicated base MOO fitness function, contained in `fitness.base_ff_classes.moo_ff`. When `[[multiple fitness functions are specified|Evaluation#specifying-multiple-fitness-functions]]`, this fitness function class is automatically used as a holding class for each individual fitness function. Fitness functions are handled as a list of individual fitness function class instances. When evaluation of an individual is called, each fitness function within this list is called in turn in order to produce an array of fitness values for the given individual. Since each individual fitness function acts as a standalone fitness function, this means that **any** fitness function in PonyGE2 can be used as a fitness component for multiple objective optimisation.

NOTE that the base MOO fitness function class automatically initialises each of the specified fitness functions during its own initialisation.

Since the base MOO fitness function class generates a list of individual (single objective) fitness functions, the base MOO fitness function does not need to inherit from the `[[base fitness function class|Evaluation#the-base-fitness-function-class]]` as each individual (single objective) fitness function so inherits. Since each individual fitness function inherits from the base class, the only checks that are required in the MOO fitness class pertain to its multiple objective nature and the handling of default fitness values.

Default Fitness

Since the actual implementation of multiple objective optimisation in PonyGE2 is defined as a list of individual fitness functions, the `[[default fitness|Evaluation#default-fitness]]` must also be handled as a list. The base MOO fitness function class defines its own `default_fitness` attribute as a list of all of the individual `default_fitness` values of the individual specified fitness functions.

With regards to the setting of default fitness values, if any single fitness value for any given fitness function returns a `NaN` value (i.e. the default fitness of the base class), then all fitness values for that particular individual are set to their default fitness values (i.e. all fitness are given the `default_fitness` value of their respective fitness functions). Furthermore, if any single fitness function encounters a runtime error, the entire individual is counted by the stats as a "runtime_error" individual (Note that a check is in place to ensure that violations across multiple fitness functions are not recorded more than once for a single individual).

Multi-Objective Operators

Since multiple objective optimisation creates a pareto front of individuals, standard single-objective operators which implement comparisons between individuals (e.g. selection and replacement) are not compatible with MOO problems. As such, NSGA2 has its own implementation of `[[selection|Selection#nsga2]]` and `[[replacement|Replacement#nsga2]]` operators. However, these are not set by default if multiple fitness functions are used. As with normal selection and replacement operators, MOO-compatible operators must be directly specified.

NOTE that MOO-compatible operators require a `multi_objective = True` attribute. Failure to do so will result in an Error.

Multi-Objective Statistics

Aside from `[[selection|Selection#nsga2]]` and `[[replacement|Replacement#nsga2]]` operators, one of the major distinctions between single objective optimisation and multiple-objective optimisation is how statistics are collected and reported on the population. As such, the use of multiple objective optimisation requires a dedicated statistics handler. Such a function is provided in `stats.stats.get_moo_stats`. This MOO stats parser is automatically used by PonyGE2 in the event of multiple fitness functions being used.

The main differences between single objective and multiple objective statistics are:

1. A single "best ever" individual is not tracked by `utilities.stats.trackers.best_ever`. Instead, the first (non-dominated) pareto front of individuals is tracked therein. This is also true for the "best_fitness" statistic, which is replaced with the "first_fronts" statistic. This new statistic tracks the number of individuals on the first (non-dominated) pareto front.
2. Since the single objective "best_fitness" is no longer tracked, normal fitness plotting can not be done with multiple objectives. As such, fitness plotting is replaced with the plotting of all fitness values on the first (non-dominated) pareto front across each generation. Fronts are color-coordinated for visual distinction. Axes are automatically labelled with the respective names of the individual fitness functions.

NOTE that at present fitness plotting is only enabled in PonyGe2 for problems with either one or two objectives. Problems with three or more objectives do not yet have fitness plotting enabled.

3. Since the single "best ever" individual is replaced with a list of all individuals on the first (non-dominated) front, saving of the single best individual with multi-objective optimisation saves a folder containing all individuals on the first front.
4. Fitness metrics across the entire population (such as average fitness) cannot be tracked. The "ave_fitness" statistic is replaced with the "pareto_fronts" statistic. This new statistic tracks the total number of pareto fronts.

Multicore evaluation

Evaluation of a population of individuals can be done in series (single core evaluation) or in parallel (multi core evaluation). Multicore evaluation can be activated with the argument:

```
--multicore
```

or by setting the parameter `MULTICORE` to `True` in either a parameters file or in the params dictionary.

Additionally, the number of processor cores used for multicore evaluation can be controlled with the argument:

```
--cores [INT]
```

or by setting the parameter `CORES` to `[INT]` in either a parameters file or in the params dictionary, where `[INT]` is an integer which specifies the number of cores used for fitness evaluations. The default value is to use all available cores.

NOTE that multicore evaluation may require a slightly longer setup time on Windows operating systems. For more information, see the documentation string of the function `utilities.algorithm.initialise_run.pool_init()`.

NOTE that multicore evaluations may not necessarily improve computational runtime for small problems as a certain overhead is necessary to run the multicore evaluation process.

NOTE that for smaller problems fitness evaluations may not necessarily present a bottleneck in terms of computational run-time. It is advised to use a python profiler to ascertain whether or not fitness evaluations present such a bottleneck. If this is the case, multicore evaluation may improve the run-time of a single evolutionary run.

NOTE that when running batches of multiple experiments, it will **always** be faster to run multiple single-core experiments in parallel, rather than multiple multi-core experiments in series.

Caching

Caching is provided in PonyGE2 to save on fitness evaluations and to track the number of unique solutions encountered during an evolutionary run. Cached individuals have their fitness stored in the `utilities.trackers.cache` dictionary. Dictionary keys are the string of the phenotype. Using the phenotype string as the key for the cache means that duplicate phenotypes will not be evaluated, even if their genetic makeup is distinct (see the `[[many-to-one mapping process]Representation#many-to-one-mapping]]` section).

Caching can be activated with the argument:

```
--cache
```

or by setting the parameter `CACHE` to `True` in either a parameters file or in the params dictionary.

There are currently three optional extras for use with the cache, all of which are activated in `fitness.evaluation.evaluate_fitness`:

Fitness Lookup

This is the default case when caching is activated. Individuals whose phenotypes match one which has already been evaluated have the previous fitness read directly from the cache, thus saving fitness evaluations. Fitness lookup can be *de*-activated with:

```
--dont_lookup_fitness
```

or by setting the parameter `LOOKUP_FITNESS` to `False` in either a parameters file or in the params dictionary.

Fitness Penalty

Individuals which have already been evaluated (i.e. duplicate individuals) are given a default bad fitness. The fitness to be assigned is the default fitness value specified in the fitness function. This parameter can be activated with the argument:

```
--lookup_bad_fitness
```

or by setting the parameter `LOOKUP_BAD_FITNESS` to `True` in either a parameters file or in the params dictionary.

Mutate Duplicates

Individuals which have already been evaluated (i.e. duplicate individuals) are mutated to produce new unique individuals which have not been encountered yet by the search process. This parameter can be activated with the argument:

```
--mutate_duplicates
```

or by setting the parameter `MUTATE_DUPLICATES` to `True` in either a parameters file or in the params dictionary.

NOTE that the various caching options are **mutually exclusive**. For example, you cannot specify `--mutate_duplicates` with `--lookup_bad_fitness`.

NOTE that if a linear mutation operator is used, it is possible for PonyGE2 to get stuck in an infinite loop if `--mutate_duplicates` is specified.