
Pyevolve Documentation

Release 0.6rc1

Christian S. Perone

April 25, 2010

CONTENTS

1	Get Involved !	3
2	Contents	5
2.1	What's new ?	5
2.2	Introduction	8
2.3	Other platforms and performance	13
2.4	Get Started - Tutorial	15
2.5	Modules	30
2.6	Graphical Analysis - Plots	106
2.7	Examples	118
2.8	F.A.Q.	139
2.9	Contributors	139
2.10	License	139
2.11	Credits	140
2.12	Contact the author	140
2.13	Donate	140
3	Index	141
	Index	143

“We can allow satellites, planets, suns, universe, nay whole systems of universe, to be governed by laws, but the smallest insect, we wish to be created at once by special act.”

- **Charles R. Darwin, 1838**

Pyevolve was developed to be a *complete genetic algorithm framework written in pure python*, but since the version 0.6, the framework is also supporting Genetic Programming, so in the near future, the framework will be more an Evolutionary Computation framework than a simple GA framework.

See the changes in the *What's new ?* section of this documentation.

This is the documentation for the release v.0.6rc1.

See some plot screenshots on the *Graph Types and Screenshots* section.

You can download this manual also in other formats:

Pyevolve PDF Manual v.0.6rc1 (PDF) *This is a PDF file version with this manual*

Pyevolve CHM Manual v.0.6rc1 (CHM - Windows Help) *This is the CHM (Windows Help) version of this manual*

GET INVOLVED !

Join with us in [Pyevolve mail-list](#).

Development information and bug reports are in the [Trac](#), and please, feel free to create new tickets with critics or suggestions.

Visit the project [blog site](#) and leave your comment.

CONTENTS

2.1 What's new ?

What's new on the release 0.6rc1:

Optimizations and bug-fixes

Added many general optimizations and bug-fixes. The code is more *pythonic* and stable now.

Documentation, documentation and documentation

Added documentation about the new GP core, new features, changes were done to reflect API changes here and there, etc...

Function Slots - Functions now have weights

Added a new *weight* parameter to the *add* method of the `FunctionSlot.FunctionSlot` class. This parameter is used when you enable the *random apply* of the slot. See the class for more information.

Multiprocessing - the use of multiprocessing module

Added a new method to the `GSimpleGA.GSimpleGA` class, the `GSimpleGA.GSimpleGA.setMultiProcessing()` method. With this method you can enable the use of **multiprocessing** python module. When you enable this option, Pyevolve will check if you have more than one CPU core and if there is support to the multiprocessing use. You **must** see the warning on the `GSimpleGA.GSimpleGA.setMultiProcessing()` method.

Scaling Scheme - the Boltzmann scaling

Added the Boltzmann scaling scheme, this scheme uses a temperature which is reduced each generation by a small amount. As the temperature decreases, the difference spread between the high and low fitnesses increases. See the description on the `Scaling.BoltzmannScaling()` function.

Scaling Scheme - Exponential and Saturated scaling

Added the Exponential and Saturated scaling schemes, using the exponential function to calculate the fitness values. See more in `Scaling.ExponentialScaling()` and `Scaling.SaturatedScaling()`.

Selectors - the alternative Tournament Selection

Added an alternative Tournament selection method, the `Selectors.GTournamentAlternative()`. This new Tournament Selector **don't** uses the Roulette Wheel method to pick individuals.

Statistics - two new statistical measures

Added the **fitTot** and the **rawTot** parameters to the `Statistics.Statistics` class. See the class documentation for more information.

Elitism - replacement option

Added the method `GSimpleGA.GSimpleGA.setElitismReplacement()`. This method is used to set the number of individuals cloned on the elitism.

String representation - resumeString

Added the method `resumeString` to all native chromosomes. This method returns a small as possible string representation of the chromosome.

DB Adapter - XML RPC

Added a new DB Adapter to send Pyevolve statistics, the XML RPC, to see more information, access the docs of the `DBAdapters.DBXMLRPC`.

DB Adapters - OO redesigned

The DB Adapters were redesigned and now there is a super class for all DB Adapters, you can create your own DB Adapters subclassing the `DBAdapters.DBBaseAdapter` class.

The Network module - lan/wan networking

Added the `Network` module, this module is used to keep all the networking related classes, currently it contains the threaded UDP client/server.

The Migration module - distributed GA

Added the `Migration` module, this module is used to control the migration of the distributed GA.

The G2DBinaryString module - the 2D Binary String

Added the `G2DBinaryString` module. This module contains the 2D Binary String chromosome representation.

1D chromosomes - new base class

All the 1D chromosomes representation is now extending the `GenomeBase.G1DBase` base class.

Tree chromosome - new Tree representation chromosome

Added the module `GTree`, this module contains the new `GTree.GTree` chromosome representation and all tree related functions and the `GTree.GTreeGP` chromosome used by Genetic Programming.

VPython DB Adapter - real-time graph statistics

Added the new `DBAdapters.DBVPythonGraph` class, this DB Adapter uses the VPython to create real-time statistics graphs.

MySQL DB Adapter - dump statistics to MySQL

Added the new `DBAdapters.DBMySQLAdapter` class, this DB Adapter will dump statistics to a local or remote MySQL database.

Genetic Programming - Pyevolve now supports GP

Added new support for the Genetic Programming, you can check the examples with symbolic regression. The `GTreeGP` chromosome representation is used for the GP main tree.

Interactive mode - no more platform independent code

Code that was platform independent from the Interactive Mode was removed, so if you are unable to enter in the Interactive Mode using the ESC key, try using the method call to enter in the mode at a defined generation.

Mutators

Added the Simple Inversion Mutation (`Mutators.G1DListMutatorSIM()`) for G1DList genome.

Added the Integer Range Mutation (`Mutators.G2DListMutatorIntegerRange()`) for the G2DList genome.

Added the Binary String Swap Mutator (`Mutators.G2DListMutatorIntegerRange()`) for the G2DBinaryString genome.

Added the Binary String Flip Mutator (`Mutators.G2DBinaryStringMutatorFlip()`) for the G2DBinaryString genome.

Added the GTree Swap Mutator (`Mutators.GTreeMutatorSwap()`) for the GTree genome.

Added the GTree Integer Range Mutator (`Mutators.GTreeMutatorIntegerRange()`) for the GTree genome.

Added the GTree Integer Gaussian Mutator (`Mutators.GTreeMutatorIntegerGaussian()`) for the GTree genome.

Added the GTree Real Range Mutator (`Mutators.GTreeMutatorRealRange()`) for the GTree genome.

Added the GTree Real Gaussian Mutator (`Mutators.GTreeMutatorRealGaussian()`) for the GTree genome.

Added the GTreeGP Operation Mutator (`Mutators.GTreeGPMutatorOperation()`) for the GTreeGP genome.

Added the GTreeGP Subtree Mutator (`Mutators.GTreeGPMutatorSubtree()`) for the GTreeGP genome.

Crossovers

Added the Cut and Crossfill Crossover (`Crossovers.G1DListCrossoverCutCrossfill()`), used for permutations, for the G1DList genome.

Added the Uniform Crossover (`Crossovers.G2DBinaryStringXUniform()`) for the G2DBinaryString genome.

Added the Single Vert. Point Crossover (`Crossovers.G2DBinaryStringXSingleVPoint()`) for the G2DBinaryString genome.

Added the Single Horiz. Point Crossover (`Crossovers.G2DBinaryStringXSingleHPoint()`) for the G2DBinaryString genome.

Added the Single Point Crossover (`Crossovers.GTreeCrossoverSinglePoint()`) for the GTree genome.

Added the Single Point Strict Crossover (`Crossovers.GTreeCrossoverSinglePointStrict()`) for the GTree genome.

Added the Single Point Crossover (`Crossovers.GTreeGPCrossoverSinglePoint()`) for the GTreeGP genome.

Added the SBX Crossover (`Crossovers.G1DListCrossoverRealSBX()`) for G1DList genome, thanks to Amit Saha.

Added the Edge Recombination (`Crossovers.G1DListCrossoverEdge()`) for G1DList genome.

Initializers

Added the Integer Initializer (`Initializers.G2DBinaryStringInitializer()`) for the G2DBinaryString genome.

Added the Integer Initializer (`Initializers.GTreeInitializerInteger()`) for the GTree genome.

Added the Allele Initializer (`Initializers.GTreeInitializerAllele()`) for the GTree genome.

Added the GTreeGP (Genetic Programming genome) Initializer (`Initializers.GTreeGPInitializer()`). It accept the methods: `grow`, `full` and `ramped`.

2.2 Introduction

This is the documentation of the Pyevolve release 0.6rc1. Since the version 0.5, Pyevolve has changed in many aspects, many new features was added and **many** bugs was fixed, this documentation describes those changes, the new API and new features.

Pyevolve was developed to be a *complete genetic algorithm framework written in pure python*, the main objectives of Pyevolve is:

- **written in pure python**, to maximize the cross-platform issue;
- **easy to use API**, the API must be easy for end-user;
- **see the evolution**, the user can and must see and *interact* with the evolution statistics, *graphs* and etc;
- **extensible**, the API must be extensible, the user can create new representations, genetic operators like crossover, mutation and etc;
- **fast**, the design must be optimized for performance;
- **common features**, the framework must implement the most common features: selectors like roulette wheel, tournament, ranking, uniform. Scaling schemes like linear scaling, etc;
- **default parameters**, we must have default operators, settings, etc in all options;
- **open-source**, the source is for everyone, not for only one.

2.2.1 Requirements

Pyevolve can be executed on **Windows**, **Linux** and **Mac** platforms.

Note: On the Mac platform, it's reported that *Pyevolve 0.5* can't enter on the *Interactive Mode*.

Pyevolve can be executed under Jython 2.5b1+, but with some restrictions:

- You can't use some features like the *SQLite3* adapter to dump statistics and *graphs* (unless you install Matplotlib on Jython, but I think that still is not possible).

Pyevolve can be executed under IronPython 2.x, but with some restrictions:

- You can't use some features like the *SQLite3* adapter to dump statistics and *graphs* (unless you install Matplotlib on Jython, but I think that still is not possible).
- You must install a **zlib module** for IronPython.

Pyevolve requires the follow modules:

- **Python 2.5+**, v.2.6 is recommended
- **Optional, for graph plotting: Matplotlib 0.98.4+** The matplotlib ¹ is required to plot the graphs.

¹ Matplotlib is Copyright (c) 2002-2008 John D. Hunter; All Rights Reserved

- **Optional, for real-time statistics visualization: VPython** The VPython ² is required to see real-time statistics visualization.
- **Optional, for drawing GP Trees: Pydot 1.0.2+** The Pydot ³ is used to plot the Genetic Programming Trees.
- **Optional, for MySQL DB Adapter: MySQL for Python** The MySQL ⁴ is used by the MySQL DB Adapter.

2.2.2 Downloads

Windows

Installers for Microsoft Windows platform:

Pyevolve v.0.6rc1 (installer) for Python 2.5 *This is an .exe installer for Microsoft Windows XP/Vista*

Pyevolve v.0.6rc1 (installer) for Python 2.6 *This is an .exe installer for Microsoft Windows XP/Vista*

Pyevolve v.0.6rc1 (source code) for Python 2.x *This is the source code*

Linux

Installation package for Linux platform:

Pyevolve v.0.6rc1 (source code) for Python 2.x *This is the source code*

Examples package

Examples package for Pyevolve v.0.6rc1:

Pyevolve examples v.0.6rc1 (examples) *This is an package with the Pyevolve source code*

2.2.3 Installation

You can download the specific Pyevolve from the *Downloads* section, or using *easy_install*.

The installation can be easy done by using the *easy_install*:

```
easy_install pyevolve
```

You can upgrade your older version too:

```
easy_install --upgrade pyevolve
```

or install a downloaded *egg package*:

```
easy_install /downloads/downloaded_package.egg
```

or install from an URL:

```
easy_install http://site/package.egg
```

² VPython was originated by David Scherer in 2000.

³ Pydot was developed by Ero Carrera.

⁴ MySQLdb was developed by Andy Dustman and contributors.

This command will automatic search, download and install a suitable version of pyevolve, once you have installed, you can test:

```
>>> import pyevolve
>>> print pyevolve.__version__
'v.0.6rc1'
```

easy_install utility is part of *setuptools*. Once you have installed *setuptools*, you will find the *easy_install.exe* program in your Python Scripts subdirectory.

2.2.4 Genetic Algorithm Features

Chromosomes / Representations 1D List, 2D List, 1D Binary String, 2D Binary String and Tree

Note: it is important to note, that the 1D List, 2D List and Tree can carry any type of python objects or primitives.

Crossover Methods

1D Binary String Single Point Crossover, Two Point Crossover, Uniform Crossover

1D List Single Point Crossover, Two Point Crossover, Uniform Crossover, OX Crossover, Edge Recombination Crossover, Cut and Crossfill Crossover, Real SBX Crossover

2D List Uniform Crossover, Single Vertical Point Crossover, Single Horizontal Point Crossover

2D Binary String Uniform Crossover, Single Vertical Point Crossover, Single Horizontal Point Crossover

Tree Single Point Crossover, Strict Single Point Crossover

Mutator Methods

1D Binary String Swap Mutator, Flip Mutator

2D Binary String Swap Mutator, Flip Mutator

1D List Swap Mutator, Integer Range Mutator, Real Range Mutator, Integer Gaussian Mutator, Real Gaussian Mutator, Integer Binary Mutator, Allele Mutator, Simple Inversion Mutator

2D List Swap Mutator, Integer Gaussian Mutator, Real Gaussian Mutator, Allele Mutator, Integer Range Mutator

Tree Swap Mutator, Integer Range Mutator, Real Range Mutator, Integer Gaussian Mutator, Real Gaussian Mutator

Initializers

1D Binary String Binary String Initializer

2D Binary String Binary String Initializer

1D List Allele Initializer, Integer Initializer, Real Initializer

2D List Allele Initializer, Integer Initializer, Real Initializer

Tree Integer Initializer, Allele Initializer

Scaling Methods

Linear Scaling, Sigma Truncation Scaling and Power Law Scaling, Raw Scaling, Boltzmann Scaling, Exponential Scaling, Saturated Scaling

Selection Methods

Rank Selection, Uniform Selection, Tournament Selection, Tournament Selection Alternative (doesn't uses the Roulette Wheel), Roulette Wheel Selection

2.2.5 Genetic Programming Features

Chromosomes / Representations

Tree

Warning: the Tree of Genetic Programming is the class `GTree.GTreeGP` and not the `GTree.GTree` class of the Genetic Algorithm representation.

Crossover Methods

Tree Single Point Crossover

Mutator Methods

Tree Operation Mutator, Subtree mutator

Initializers

Tree Grow Initializer, Full Initializer, Ramped Half-n-Half

Scaling Methods

Linear Scaling, Sigma Truncation Scaling and Power Law Scaling, Raw Scaling, Boltzmann Scaling, Exponential Scaling, Saturated Scaling

Selection Methods

Rank Selection, Uniform Selection, Tournament Selection, Tournament Selection Alternative (doesn't uses the Roulette Wheel), Roulette Wheel Selection

2.2.6 Genetic Algorithms Literature

In this section, you will find study material to learn more about Genetic Algorithms.

Books

Goldberg, David E (1989), *Genetic Algorithms in Search, Optimization and Machine Learning*, Kluwer Academic Publishers, Boston, MA.

Goldberg, David E (2002), *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*, Addison-Wesley, Reading, MA.

Fogel, David B (2006), *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, IEEE Press, Piscataway, NJ. Third Edition

Holland, John H (1975), *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor

Michalewicz, Zbigniew (1999), *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag.

See Also:

Wikipedia: Genetic Algorithms The Wikipedia article about Genetic Algorithms.

Sites

Introduction to Genetic Algorithms A nice introduction by Marek Obitko.

A Field Guide to Genetic Programming A book, freely downloadable under a Creative Commons license.

A Genetic Algorithm Tutorial by Darrell Whitley Computer Science Department Colorado State University

An excellent tutorial with lots of theory

2.2.7 Genetic Programming Literature

In this section, you will find study material to learn more about Genetic Programming.

Books

Poli, Riccardo; Langdon, William B.; McPhee, Nicholas F., *A Field Guide to Genetic Programming*, this book is also available online (a GREAT initiative from authors) in [Book Site](#)

Koza, John R., *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

See Also:

Wikipedia: Genetic Programming The Wikipedia article about Genetic Programming.

Sites

Introduction to Genetic Programming A nice collection of GP related content !

A Field Guide to Genetic Programming A book, freely downloadable under a Creative Commons license.

The Genetic Programming Bibliography A very interesting initiative maintained by William Langdon, Steven Gustafson, and John Koza. Over than 6000 GP references !

2.2.8 Glossary / Concepts

Raw score The raw score represents the score returned by the *Evaluation function*, this score is not scaled.

Fitness score The fitness score is the scaled raw score, for example, if you use the Linear Scaling (`Scaling.LinearScaling()`), the fitness score will be the raw score scaled with the Linear Scaling method. The fitness score represents how good is the individual relative to our population.

Evaluation function Also called *Fitness Function* or *Objective Function*, the evaluation function is the function which evaluates the genome, giving it a raw score. The objective of this function is to quantify the solutions (individuals, chromosomes)

See Also:

Wikipedia: Fitness Function An article talking about the Evaluation function, or the “Fitness Function”.

Sample genome The sample genome is the genome which are used as configuration base for all the new replicated genomes.

Interactive mode Pyevolve have an interactive mode, you can enter in this mode by pressing ESC key before the end of the evolution. When you press ESC, a python environment will be load. In this environment, you have some analysis functions and you can interact with the population of individuals at the specific generation.

See Also:

Module Interaction The Interaction module.

Step callback function This function, when attached to the GA Engine (`GSimpleGA.GSimpleGA`), will be called every generation. It receives one parameter, the GA Engine by itself.

Data Type Independent When a genetic operator is data type independent, it will operate on different data types but not with different chromosome representation, for example, the `Mutators.G1DListMutatorSwap()` mutator will operate on Real, Allele or Integer `G1DList.G1DList` chromosome, but not on `G2DList.G2DList` chromosome.

Standardized Fitness The standardized fitness restates the raw score so that a lower numerical value is always a better value.

See Also:

Genetic Programming: On the Programming of Computers by Means of Natural Selection A book from John R. Koza about Genetic Programming.

Adjusted Fitness The adjusted fitness is a measure computed from the Standardized Fitness, the Adjusted Fitness is always between 0 and 1 and it's always bigger for better individuals.

See Also:

Genetic Programming: On the Programming of Computers by Means of Natural Selection A book from John R. Koza about Genetic Programming.

Non-terminal node The non-terminal node or non-terminal function is a function in a parse tree which is either a root or a branch in that tree, in the GP we call non-terminal nodes as "functions", the opposite of terminal nodes, which are the variables of the GP.

See Also:

Wikipedia: Genetic Algorithm An article talking about Genetic Algorithms.

Wikipedia: Genetic Programming The Wikipedia article about Genetic Programming.

2.3 Other platforms and performance

2.3.1 Running Pyevolve on Symbian OS (PyS60)

Pyevolve is compatible with PyS60 2.0 (but older versions of the 1.9.x trunk should work fine too); PyS60 2.0 is a port of Python 2.5.4 core to the S60 smartphones, it was made by Nokia and it's Open Source. All smartphones based on the [S60 2nd and 3rd editions](#) should run PyS60, you can download it from the [Maemo garage project home](#).

To install Pyevolve in PyS60 you simple need to copy the "pyevolve" package (you can use the sources of Pyevolve or even the "pyevolve" of your Python installation to the smartphone in a place that PyS60 can find it, usually in `c:\resource\Python25`, for more information read the PyS60 documentation. The Genetic Algorithms and the Genetic Programming cores of Pyevolve was tested with PyS60 2.0, but to use Genetic Programming, you must define explicitly the funtions of the GP, like in [How to manually add non-terminal functions to Genetic Programming core](#).

Of course not all features of Pyevolve are supported in PyS60, like for example some DBAdapters and the graphical plotting tool, since no matplotlib port is available to PyS60 at the moment. Pyevolve was tested with PyS60 2.0 in a Nokia N78 and in a Nokia N73 smartphones.

See Also:

Croozeus.com - home to PyS60 developers A lot of information and tutorials about PyS60, very recommended.

Python for S60 - OpenSource The PyS60 project wiki.

2.3.2 Running Pyevolve on Jython

Jython is an implementation of Python language and it's modules (not all unfortunately) which is designed to run over the Java platform. Pyevolve was tested against Jython 2.5.x and worked well, except for the Genetic Programming core which is taking a lot of memory, maybe a Jython issue with the Java JVM.

You're highly encouraged to run Jython with the JVM "-server" option; this option will enable another VM JIT which is optimal for applications where the fast startup times isn't important, and the overall performance is what matters. This JIT of the "Server mode" has different policies to compile your code into native code, and it's well designed for long running applications, where the VM can profile and optimize better than the JIT of "Client mode".

Pyevolve was tested against Jython 2.5.1 in Java v.1.6.0_18 Java(TM) SE Runtime Environment (build 1.6.0_18-b07) Java HotSpot(TM) Client VM (build 16.0-b13, mixed mode, sharing)

See Also:

Jython [Official Jython project home](#).

Java HotSpot [The Java HotSpot Performance Engine Architecture](#).

2.3.3 Running Pyevolve on IronPython

IronPython is an open-source implementation of the Python programming language targeting the .NET Framework and Mono, written entirely in C# and created by Jim Hugunin. IronPython is currently language-compatible with Python 2.6.

Pyevolve was tested against the IronPython 2.6 (2.6.10920.0) in a Windows XP SP3 with .NET 2.0.50727.3603.

See Also:

Official IronPython project home [Official IronPython project home](#).

Differences between IronPython and CPython [Documents with differences between IronPython and CPython \(the official Python interpreter\)](#).

IronPython performance benchmarks [A lot of benchmarks and comparisons between IronPython and CPython](#).

2.3.4 Running Pyevolve on iPod/iPhone

The Genetic Algorithm core of Pyevolve was tested on iPod Touch 2G with the firmware v.3.1.2. To use it, you first must install the port of Python 2.5+ to the OS of iPod. You just need to put the Pyevolve package inside the directory where you'll call your application or just put it inside another place where the Python from iPod/iPhone can be found in path.

See Also:

Miniguide to install Python on iPhone [Miniguide on how to install Python on iPhone](#)

2.3.5 Improving Pyevolve performance

Pyevolve, at least for the versions ≤ 0.6 , have all modules written in pure Python, which enables some very useful features and portability, but sometimes weights in performance. Here are some ways users and developers use to increase the performance of Pyevolve:

Psyco Psyco is the well known Python specializing compiler, created by Armin Rigo. Psyco is very easy to use and can give you a lot of speed up.

Cython Cython is a specific language used to create C/C++ extensions for Python, it is based on the Python language itself, so if you think Psyco is not enough or aren't giving too much optimizations, you can use Cython to create your own C/C++ extensions; the best approach is to use Cython to build your *Evaluation function*, which is usually the most consuming part of Genetic Algorithms.

See Also:

Psyco at Sourceforge.net The official site of Psyco at Sourceforge.net

Psyco 2.0 binaries for Windows Development of psyco was recently done by Christian Tismer. Here you'll find the binaries of Psyco 2.0 (Python 2.4, 2.5 and 2.6) for Windows.

Cython - C-Extensions for Python Official Cython project home.

2.4 Get Started - Tutorial

Pyevolve combined with Python language can be a powerful tool. The best way to show you how Pyevolve can be used, is beginning with simple examples, later we'll show some snippets and etc. So you'll can walk by yourself.

2.4.1 First Example

To make the API easy to use, we have created default parameters for almost every parameter in Pyevolve, for example, when you will use the `G1DList.G1DList` genome without specifying the Mutator, Crossover and Initializer, you will use the default ones: Swap Mutator, One Point Crossover and the Integer Initializer. All those default parameters are specified in the `Consts` module (and you are highly encouraged to take a look at source code).

Let's begin with the first simple example (Ex. 1). First of all, you must know your problem, in this case, our problem is to find a simple 1D list of integers of n -size with zero in all positions. At the first look, we know by intuition that the representation needed to this problem is a 1D List, which you can found in Pyevolve by the name of `G1DList.G1DList`, which means Genome 1D List. This representation is based on a python list as you will see, and is very easy to manipulate. The next step is to define the our *evaluation function* to our Genetic Algorithm. We want all the n list positions with value of '0', so we can propose the evaluation function:

$$f(x) = \sum_{i=0}^n (x[i] == 0) ? 1 : 0$$

As you can see in the above equation, with the x variable representing our genome list of integers, the $f(x)$ shows our evaluation function, which is the sum of '0' values in the list. For example, if we have a list with 10 elements like this:

```
x = [1, 2, 3, 8, 0, 2, 0, 4, 1, 0]
```

we will got the raw score⁵ value of 3, or $f(x) = 3$. It's very simple to understand. Now, let's code this.

We will define our *evaluation function* "eval_func" as:

```
# This function is the evaluation function, we want
# to give high score to more zero'ed chromosomes
def eval_func(chromosome):
    score = 0.0
```

⁵ It is important to note that in Pyevolve, we have *raw score* and *fitness score*, the raw score is the return of the evaluation function and the fitness score is the scaled score or the raw score in absence of a scaling scheme.

```
# iterate over the chromosome elements (items)
for value in chromosome:
    if value==0:
        score += 1.0

return score
```

As you can see, this evaluation function verify each element in the list which is equal to ‘0’ and return the proportional score value. The `G1DList.G1DList` chromosome is not a python list by itself but it encapsulates one and exposes the methods for this list, like the iterator used in the above loop. The next step is the creation of an one *sample genome*⁶ for the Genetic Algorithm. We can define our genome as this:

```
# Genome instance
genome = G1DList.G1DList(20)

# The evaluator function (objective function)
genome.evaluator.set(eval_func)
```

This will create an instance of the `G1DList.G1DList` class (which resides in the `G1DList` module) with the list *n*-size of 20 and sets the evaluation function of the genome to the evaluation function “**eval_func**” that we have created before.

But wait, where is the range of integers that will be used in the list ? Where is the mutator, crossover and initialization functions ? They are all in the default parameters, as you see, this parameters keep things simple.

By default (and you have the **documentation** to find this defaults), the range of the integers in the `G1DList.G1DList` is between the interval [`Consts.CDefRangeMin`, `Consts.CDefRangeMax`] inclusive, and genetic operators is the same I have cited before: Swap Mutator `Mutators.G1DListMutatorSwap()`, One Point Crossover `Crossovers.G1DListCrossoverSinglePoint()` and the Integer Initializer `Initializers.G1DListInitializerInteger()`. You can change everything with the API, for example, you can pass the ranges to the genome, like this:

```
genome.setParams(rangemin=0, rangemax=10)
```

Right, now we have our evaluation function and our first genome ready, the next step is to create our Genetic Algorithm Engine, the GA Core which will do the evolution, control statistics, etc... The GA Engine which we will use is the `GSimpleGA.GSimpleGA` which resides in the `GSimpleGA` module, this GA Engine is the genetic algorithm⁷ described by Goldberg. So, let’s create the engine:

```
ga = GSimpleGA.GSimpleGA(genome)
```

Ready ! Simple not ? We simple create our GA Engine with the created genome. You can ask: “*Where is the selector method ? The number of generations ? Mutation rate ?*“. Again: we have defaults. By default, the GA will evolve for 100 generations with a population size of 80 individuals, it will use the mutation rate of 2% and a crossover rate of 80%, the default selector is the Ranking Selection (`Selectors.GRankSelector()`) method. Those default parameters was not random picked, they are all based on the commom used properties.

Now, all we need to do is to evolve !

```
# Do the evolution, with stats dump
# frequency of 10 generations
ga.evolve(freq_stats=10)
```

⁶ The term *sample genome* means one genome which provides the main configuration for all individuals.

⁷ This GA uses non-overlapping populations.

```
# Best individual
print ga.bestIndividual()
```

Note: Pyevolve have the `__repr__()` function implemented for almost all objects, this means that you can use syntax like ‘print object’ and the object information will be show in an pretty format.

Ready, now we have our first Genetic Algorithm, it looks more like a “Hello GA !” application. The code above shows the call of the `GSimpleGA.GSimpleGA.evolve()` method, with the parameter `freq_stats=10`, this method will do the evolution and will show the statistics every 10th generation; the next method called is the `GSimpleGA.GSimpleGA.bestIndividual()`, this method will return the best individual after the end of the evolution, and the with the `print` python command, we will show the genome on the screen.

This is what this example will results:

```
Gen. 1 (1.00%): Max/Min/Avg Fitness(Raw) [2.40(3.00) / 1.60(1.00) / 2.00(2.00)]
Gen. 10 (10.00%): Max/Min/Avg Fitness(Raw) [10.80(10.00) / 7.20(8.00) / 9.00(9.00)]
Gen. 20 (20.00%): Max/Min/Avg Fitness(Raw) [22.80(20.00) / 15.20(18.00) / 19.00(19.00)]
Gen. 30 (30.00%): Max/Min/Avg Fitness(Raw) [20.00(20.00) / 20.00(20.00) / 20.00(20.00)]
(...)
Gen. 100 (100.00%): Max/Min/Avg Fitness(Raw) [20.00(20.00) / 20.00(20.00) / 20.00(20.00)]
```

Total time elapsed: 3.375 seconds.

```
- GenomeBase
  Score:                20.000000
  Fitness:              20.000000

  Slot [Evaluator] (Count: 1)
    Name: eval_func
  Slot [Initializer] (Count: 1)
    Name: G1DListInitializerInteger
    Doc: Integer initialization function of G1DList,
         accepts 'rangemin' and 'rangemax'
  Slot [Mutator] (Count: 1)
    Name: G1DListMutatorSwap
    Doc: The mutator of G1DList, Swap Mutator
  Slot [Crossover] (Count: 1)
    Name: G1DListCrossoverSinglePoint
    Doc: The crossover of G1DList, Single Point
- G1DList
  List size:           20
  List:                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

This is the evolution of our Genetic Algorithm with the best individual show at the end of the evolution. As you can see, the population have obtained the best raw score (20.00) near the generation 20.

Final source code

Here is the final source code:

```
from pyevolve import G1DList
from pyevolve import GSimpleGA

def eval_func(chromosome):
    score = 0.0
    # iterate over the chromosome
```

```
for value in chromosome:
    if value==0:
        score += 1
return score

genome = G1DList.G1DList(20)
genome.evaluator.set(eval_func)
ga = GSimpleGA.GSimpleGA(genome)
ga.evolve(freq_stats=10)
print ga.bestIndividual()
```

2.4.2 The Interactive Mode

Pyevolve have introduced the concept of the *Interactive Mode* in the course of evolution. When you are evolving, and the Interactive Mode is enabled, you can press the *ESC Key* anytime in the evolution process. By pressing that key, you will enter in the interactive mode, with a normal python prompt and the `Interaction` module exposed to you as the “it” module.

Warning: note that the Interactive Mode for Linux/Mac was disabled in the 0.6 release of Pyevolve. The cause was the platform dependant code. To use it in Linux/Mac you must set the generation in wich Pyevolve will enter in the Interactive Mode by using `GSimpleGA.GSimpleGA.setInteractiveGeneration()` method; see the `Interaction` module documentation for more information.

If you want to continue the evolution, just press *CTRL-D* on Linux or *CTRL-Z* on Windows.

See this session example:

```
# pyevolve_ex1_simple.py
Gen. 1 (0.20%): Max/Min/Avg Fitness(Raw) [6.18(11.00)/4.42(1.00)/5.15(5.15)]
Gen. 20 (4.00%): Max/Min/Avg Fitness(Raw) [11.70(15.00)/7.24(3.00)/9.75(9.75)]
Gen. 40 (8.00%): Max/Min/Avg Fitness(Raw) [17.99(21.00)/12.00(9.00)/14.99(14.99)]
Loading module pylab (matplotlib)... done!
Loading module numpy... done!

## Pyevolve v.0.6 - Interactive Mode ##
Press CTRL-D to quit interactive mode.
>>>
```

As you can see, when you press the *ESC Key*, a python prompt will be show and the evolution will be paused.

Now, *what you can do* with this prompt !?

- See all the current population individuals
- Change the individuals
- Plot graphics of the current population
- Data analysis, etc... python is your limit.

Note: to use graphical plots you will obviously need the Matplotlib, see more information in the *Requirements* section for more information.

Inspecting the population

This is a session example:

```

## Pyevolve v.0.6 - Interactive Mode ##
Press CTRL-Z to quit interactive mode.
>>> dir()
['__builtins__', 'ga_engine', 'it', 'population', 'pyevolve']
>>>
>>> population
- GPopulation
  Population Size:          80
  Sort Type:                Scaled
  Minimax Type:             Maximize
  Slot [Scale Method] (Count: 1)
    Name: LinearScaling
    Doc: Linear Scaling scheme

  .. warning :: Linear Scaling is only for positive raw scores

- Statistics
  Minimum raw score          = 10.00
  Minimum fitness           = 13.18
  Standard deviation of raw scores = 2.71
  Maximum fitness           = 19.92
  Maximum raw score         = 23.00
  Fitness average           = 16.60
  Raw scores variance       = 7.36
  Average of raw scores     = 16.60

>>> len(population)
80
>>> individual = population[0]
>>> individual
- GenomeBase
  Score:                    23.000000
  Fitness:                  19.920000

  Slot [Evaluator] (Count: 1)
    Name: eval_func
  Slot [Initializer] (Count: 1)
    Name: G1DListInitializerInteger
    Doc: Integer initialization function of G1DList

  This initializer accepts the *rangemin* and *rangemax* genome parameters.

  Slot [Mutator] (Count: 1)
    Name: G1DListMutatorSwap
    Doc: The mutator of G1DList, Swap Mutator
  Slot [Crossover] (Count: 1)
    Name: G1DListCrossoverSinglePoint
    Doc: The crossover of G1DList, Single Point

  .. warning:: You can't use this crossover method for lists with just one element.

- G1DList

```

```
List size:      50
List:          [0, 5, 6, 7, 2, 0, 8, 6, 0, 0, 8, 7, 5, 6, 6, 0, 0, 3, 0, 4, 0, 0, 9, 0, 9,
, 5, 0, 0, 2, 0, 0, 0, 1, 8, 7, 0, 8, 9, 0, 8, 0, 0, 0, 9, 0]
```

The exposed modules and objects

The `Interaction` module is imported with the name “it”, you can see calling the python native `dir()`:

```
>>> dir()
['__builtins__', 'ga_engine', 'it', 'population', 'pyevolve']
```

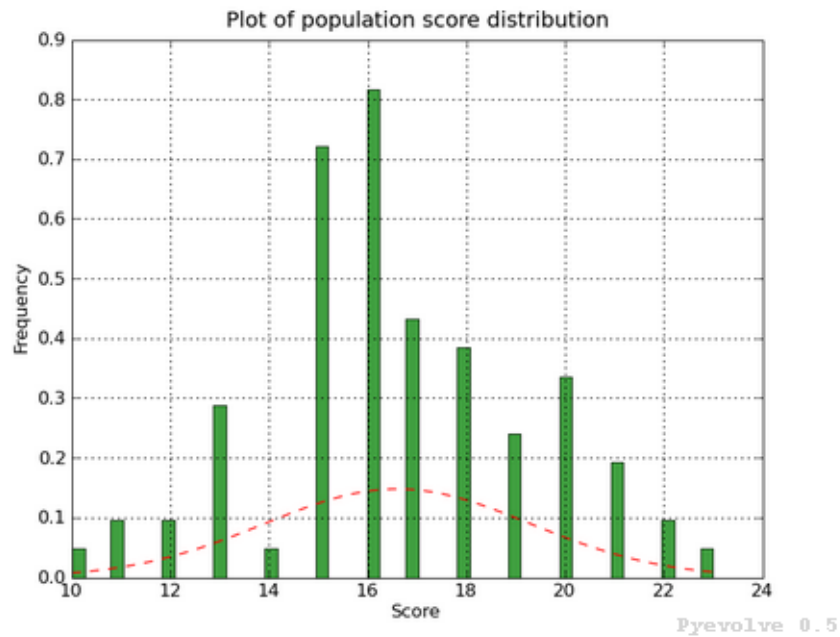
The namespace have the the following modules:

- ga_engine* The `GSimpleGA.GSimpleGA` instance, the GA Engine.
- it* The `Interaction` module, with the utilities and graph plotting functions.
- population* The current population.
- pyevolve* The main namespace, the `pyevolve` module.

Using the “it” module

Plotting the current population raw scores histogram

```
>>> it.plotHistPopScore(population)
```



Plotting the current population raw scores distribution

```
>>> it.plotPopScore(population)
```


Let's code the initial draft of our chromosome class:

```
from GenomeBase import GenomeBase

class G1DBinaryString(GenomeBase):
    pass
```

As you see, we have imported the `GenomeBase.GenomeBase` class from the `GenomeBase` module and we have created the `G1DBinaryString` class extending the base class.

The next step is to create our constructor method for our class, I'll show it before and explain later:

```
def __init__(self, length=10):
    GenomeBase.__init__(self)
    self.genomeString = []
    self.stringLength = length
    self.initializator.set(Consts.CDefG1DBinaryStringInit)
    self.mutator.set(Consts.CDefG1DBinaryStringMutator)
    self.crossover.set(Consts.CDefG1DBinaryStringCrossover)
```

Well, we start by calling the base class constructor and then creating an internal list to hold our '0's and '1's. It is important to note that we don't initialize the list, this will be done by our initializer function, and it is because of this that we must keep as an internal attribute the length of your 1D Binary String.

Next, we set our initializer, mutator and crossover to constants, this constants have just the functions of our genetic operators, but if you want, you can set they later, in this example, we will use the defaults for the G1D Binary String.

Note: The attributes `self.initializator`, `self.mutator` and `self.crossover` are all inherited from the `GenomeBase` class. They are all function slots (`FunctionSlot.FunctionSlot`).

Now, you *must* provide the `copy()` and `clone()` methods for your chromosome, because they are used to replicate the chromosome over the population or when needed by some genetic operators like reproduction.

The `copy()` method is very simple, what you need to do is to create a method that copy the contents of your chromosome to another chromosome of the `G1DBinaryString` class.

Here is our `copy()` method:

```
def copy(self, g):
    """ Copy genome to 'g' """
    GenomeBase.copy(self, g)
    g.stringLength = self.stringLength
    g.genomeString = self.genomeString[:]
```

As you can see, we first call the base class `copy()` method and later we copy our string length attribute and our internal `genomeString`, which is our list of '0's and '1's.

Warning: It is very important to note that you must **COPY** and not just create a reference to the object. On the line that we have the `self.genomeString[:]`, if you use just `self.genomeString`, you will create a **REFERENCE** to this object and not a copy. This a simple warning, but can avoid many headaches.

The next step is to create our `clone()` method, the clone method, as the name says, is a method which return another instance of the current chromosome with the same contents.

Let's code it:

```
def clone(self):
    """ Return a new instace copy of the genome """
    newcopy = G1DBinaryString(self.stringLength)
    self.copy(newcopy)
    return newcopy
```

We simple create a new instance and use the `copy()` method that we have created to copy the instance contents.

Ready ! We have our first representation chromosome. You can add many more features by implementing python operators like `__getitem__`, `__setitem__`.

Creating the initializer

Sorry, not written yet.

Creating the mutator

Sorry, not written yet.

Creating the crossover

The file `Crossovers.py` implements the crossover methods available in Pyevolve. So, that is where you should look to implement your new crossover method. The process of adding a new crossover method is as follows:

1. Create a new method such that the name reflects the type of chromosome representation it works with, and the crossover method name. For example, `Crossovers.G1DListCrossoverRealSBX()`, can work with 1D List representations and it operates on real values and it is the SBX crossover operator.
2. The method must take two parameters, 'genome' and 'args'.
3. From 'args', get the two parents which will take part in the crossover, `gMom` and `gDad`.
4. Once you have `gMom` and `gDad`, use them to create the two children, `sister` and `brother`.
5. Simply return the `sister` and `brother`.

Any constants that your crossover method uses should be defined in `Consts.py` (`Consts`).

2.4.4 Genetic Programming Tutorial

In the release 0.6 of Pyevolve, the new Genetic Programming core was added to the framework. In the *Example 18 - The Genetic Programming example* you'll see how simple and easy is Pyevolve GP core when compared with other static-typed languages.

Here is a simple example:

```
from pyevolve import Util
from pyevolve import GTree
from pyevolve import GSimpleGA
from pyevolve import Consts
import math

rmse_accum = Util.ErrorAccumulator()

def gp_add(a, b): return a+b
```

```

def gp_sub(a, b): return a-b
def gp_mul(a, b): return a*b
def gp_sqrt(a): return math.sqrt(abs(a))

def eval_func(chromosome):
    global rmse_accum
    rmse_accum.reset()
    code_comp = chromosome.getCompiledCode()

    for a in xrange(0, 5):
        for b in xrange(0, 5):
            evaluated = eval(code_comp)
            target = math.sqrt((a*a)+(b*b))
            rmse_accum += (target, evaluated)

    return rmse_accum.getRMSE()

def main_run():
    genome = GTree.GTreeGP()
    genome.setParams(max_depth=4, method="ramped")
    genome.evaluator += eval_func

    ga = GSimpleGA.GSimpleGA(genome)
    ga.setParams(gp_terminals = ['a', 'b'],
                gp_function_prefix = "gp")

    ga.setMinimax(Consts.minimaxType["minimize"])
    ga.setGenerations(50)
    ga.setCrossoverRate(1.0)
    ga.setMutationRate(0.25)
    ga.setPopulationSize(800)

    ga(freq_stats=10)
    best = ga.bestIndividual()
    print best

if __name__ == "__main__":
    main_run()

```

Let's work now step by step on the code to learn what each building block means, the first part you see the imports:

```

from pyevolve import Util
from pyevolve import GTree
from pyevolve import GSimpleGA
from pyevolve import Consts
import math

```

In the `Util` module is where we'll find many utilities functions and classes like `Util.ErrorAccumulator`. The `GTree` is where resides the `GTree.GTreeGP` class, which is the main genome used by the GP core of Pyevolve. Note that we are importing the `GSimpleGA` module, in fact, the GA core will detect when you use a Genetic Programming genome and will act as the GP core. The modules `Consts` and `math` imported here are for auxiliary use only. Next we have:

```
rmse_accum = Util.ErrorAccumulator()
```

Here we instantiate the `Util.ErrorAccumulator`, which is a simple accumulator for errors. It has methods for getting *Adjusted Fitness*, *Mean Square Error*, *Root Mean Square Error*, *mean*, *squared* or *non-squared* error measures.

In the next block we define some GP operators:

```
def gp_add(a, b): return a+b
def gp_sub(a, b): return a-b
def gp_mul(a, b): return a*b
def gp_sqrt(a): return math.sqrt(abs(a))
```

See that they are simple Python functions starting with the “gp” prefix, this is important if you want that Pyevolve automatically add them as non-terminals of the GP core. As you can note, the square root is a protected square root, since it uses the absolute value of “a” (we don’t have square root of negative numbers, except in the complex analysis). You can define any other function you want. Later we have the declaration of the *Evaluation function* for the GP core:

```
def eval_func(chromosome):
    global rmse_accum
    rmse_accum.reset()
    code_comp = chromosome.getCompiledCode()

    for a in xrange(0, 5):
        for b in xrange(0, 5):
            evaluated = eval(code_comp)
            target = math.sqrt((a*a)+(b*b))
            rmse_accum += (target, evaluated)

    return rmse_accum.getRMSE()
```

As you see, the `eval_func()` receives one parameter, the chromosome (the GP Tree in our case, an instance of the `GTree.GTreeGP` class). We first declare the global error accumulator and reset it, since we’ll start to evaluate a new individual, a new “program”. In the line where we call `GTree.GTreeGP.getCompiledCode()`, here is what happens: Pyevolve will get the pre ordered expression of the GP Tree and then will compile it into Python bytecode, and will return to you an object of the type “code”. This object can then be executed using the Python native `eval()` function. Why compiling it in bytecodes? Because if we don’t compile the program into Python bytecode, we will need to parse the Tree every time we want to evaluate our program using defined variables, and since this is a common use of the GP program, this is the fastest way we can do it in pure Python.

The next block we simple iterate using two variables “a” and “b”.

Note: Please note that the variable names here is the same that we will use as terminals later.

What you see now is the evaluation of the “code_comp” (which is the GP individual) and the evaluation of the objective function in which we want to fit (the Pythagorean theorem). Next we simple add the “target” value we got from the Pythagorean theorem and the “evaluated” value of the individual to the Error Accumulator. In the end of the evaluation function, we return the **Root Mean Square Error**. If you don’t like to add the evaluated and the target values using a tuple, you can use the `Util.ErrorAccumulator.append()` method, which will give the same results.

Next we start to define our `main_run()` function:

```
def main_run():
    genome = GTree.GTreeGP()
    genome.setParams(max_depth=4, method="ramped")
    genome.evaluator.set(eval_func)
```

The first thing we instantiate here is the `GTree.GTreeGP` class (the GP individual, the Tree). Next we set some parameters of the `GTreeGP`. The first is the “max_depth”, which is used by genetic operators and initializers to control bloat, in this case, we use 4, which means that no Tree with a height > 4 will grow. Next we set the “method”, this is the initialization method, and the values accepted here depends of the initialization method used, since we do not have specified the initialization method, Pyevolve will use the default, which is the

`Initializers.GTreeGPInitializer()` (it accepts “grow”, “full” and “ramped” methods for Tree initialization. And in the last line of this block, we set the previously defined evaluation function called `eval_func()`. In the next block we then instantiate the `GSimpleGA` core and set some parameters:

```
ga = GSimpleGA.GSimpleGA(genome)
ga.setParams(gp_terminals      = ['a', 'b'],
             gp_function_prefix = "gp")
```

The “ga” object will hold an instance of the `GSimpleGA.GSimpleGA` class, which is the core for both Genetic Algorithms and Genetic Programming. Pyevolve will automatically detect if you are creating a environment for a GP or for a GA. Next we set some parameters of the core, the first is a list called “gp_terminals”. The “gp_terminals” will hold the “variables” or in GP vocabulary . Note that the name of the terminals are the same we used in our evaluation function called `eval_func()`. The next step is to define the prefix of the GP operators (functions) or the *Non-terminal node*. Pyevolve will automatically search for all functions defined in the module which starts with “gp” (example: `gp_sub`, `gp_add`, `gp_IHateJava`, etc...) and will add these functions as the non-terminal nodes of the GP core.

The next part of the code is almost the same as used in the Genetic Algorithms applications, they are the EA parameters to setup and start the evolution:

```
ga.setMinimax(Consts.minimaxType["minimize"])
ga.setGenerations(50)
ga.setCrossoverRate(1.0)
ga.setMutationRate(0.25)
ga.setPopulationSize(800)

ga(freq_stats=10)
best = ga.bestIndividual()
print best
```

And in the last part of the source code, we have:

```
if __name__ == "__main__":
    main_run()
```

This part is important, since Pyevolve needs to know some information about objects in the main module using instrospection, you **MUST NEED** to declare this checking, the `multiprocessing` module of Python only works with this too, so if you’re planning to use it, please do not forget.

And that’s it, you have done your first GP program.

Visualizing individuals

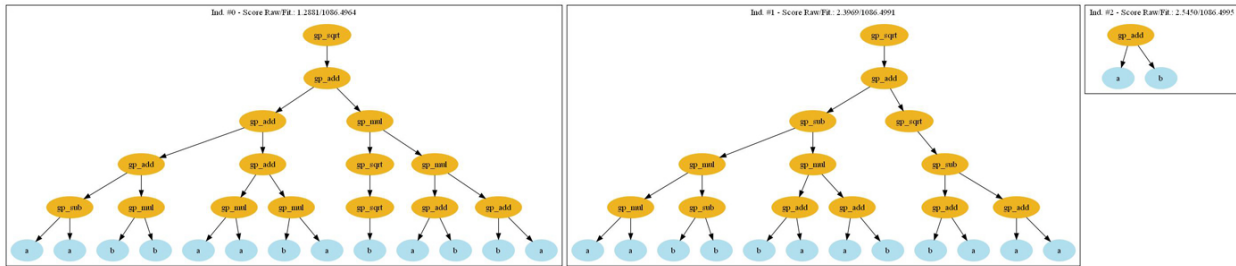
Pyevolve comes with a plotting utility to make pictures of your GP individuals, it uses the “pydot” and “Graphviz” to create those images. See more information in the *Requirements* section. What you need to change to see, for example, the first 3 best individuals of your first generation is to add a *Step callback function* into the code, let’s first define the callback function:

```
def step_callback(gp_engine):
    if gp_engine.getCurrentGeneration() == 0:
        GTree.GTreeGP.writePopulationDot(gp_engine, "trees.jpg", start=0, end=3)
```

The code is self explanative, the parameter is the GP core, first we check if it is the first generation and then we use the `GTree.GTreeGP.writePopulationDot()` method to write to the “trees.jpg” file, the range from 0 and 3 individuals of the population. Then in the main function where we instantiate the GP core, we simple use:

```
ga.stepCallback.set(step_callback)
```

And the result will be:



2.4.5 Snippets

Here are some snippets to help you.

Using two mutators at same time

To use two mutators at same time, you simple add one more to the mutator function slot, like this:

```
>>> genome.mutator.set(Mutators.G1DListMutatorRealGaussian)
>>> genome.mutator.add(Mutators.G1DListMutatorSwap)
```

The first line will set the `Mutators.G1DListMutatorRealGaussian()`, and the second line add one more mutator, the `Mutators.G1DListMutatorSwap()`.

As you can see, it's very simple and easy, and you will have two mutation operators at same time.

If you want, that just one of this mutators (random picked) be executed at the mutation process, set the *random apply* parameter of the `FunctionSlot.FunctionSlot` class to `True`

```
>>> genome.mutator.setRandomApply(True)
```

Using one allele to all list (chromosome) elements (genes)

Sometimes you want to use just one allele type to all genes on the 1D List or other chromosomes, you simple add one allele type and enable the *homogeneous* flag to **True**:

```
>>> setOfAlleles = GAllele.GAlleles(homogeneous=True)
>>> lst = [ "1", "two", 0, 777 ]
>>> a = GAllele.GAlleleList(lst)
>>> setOfAlleles.add(a)
```

Ready, your `setOfAlleles` is the `GAllele.GAlleles` class instance with the *lst* (["1", "two", 0, 777]) as alleles in all genes.

Changing the selection method

To change the default selection method, you must do this:

```
>>> ga = GSimpleGA.GSimpleGA(genome)
>>> ga.selector.set(Selectors.GTournamentSelector)
```

In this example, we are changing the selection method to the `Selectors.GTournamentSelector()`, the Tournament Selector.

Doing the same evolution on with random seed

Using a random seed, you can guarantee that the evolution will be always the same, no matter the number of executions you make. To initialize the GA Engine with the random seed, use the *seed* parameter when instantiating the `GSimpleGA.GSimpleGA` class:

```
ga_engine = GSimpleGA(genome, 123)
# or
ga_engine = GSimpleGA(genome, seed=123)
```

The value *123* will be passed as the random seed of the GA Engine.

Writing the evolution statistics to a CSV File

You can write all the statistics of an evolution to a CSV (Comma Separated Values) file using the DB Adapter called `DBAdapters.DBFileCSV`, just create an instance of the adapter and attach it to the GA Engine:

```
csv_adapter = DBFileCSV(identify="run1", filename="stats.csv")
ga_engine.setDBAdapter(csv_adapter)
```

Ready ! Now, when you run your GA, all the stats will be dumped to the CSV file. You can set the frequency in which the stats will be dumped, just use the parameter *frequency* of the `DBFileCSV`.

Use the HTTP Post to dump GA statistics

With the `DBAdapters.DBURLPost`, you can call an URL with the population statistics in every generation or at specific generation of the evolution:

```
urlpost_adapter = DBURLPost("http://localhost/post.py", identify="run1", frequency=100)
ga_engine.setDBAdapter(urlpost_adapter)
```

Now, the URL “`http://localhost/post.py`” will be called with the statistics params in every 100 generations. By default, the adapter will use the HTTP POST method to send the parameters, but you can use GET method setting the *post* paramter to *False*.

See the mod:*Statistics* and `DBAdapters.DBURLPost` documentation.

Using two or more *evaluation function*

To use two or more *evaluation function*, you can just add all the evaluators to the slot:

```
genome.evaluator.set(eval_func1)
genome.evaluator.add(eval_func2)
```

The result raw score of the genome, when evaluated using more then on evaluation function, will be the sum of all returned scores.

Note: the method *set* of the function slot remove all previous functions added to the slot.

Real-time statistics visualization

You have three options to view the statistics while in the course of the evolution:

Console statistics

You can view the statistics by setting the *freq_stats* parameter of the `GSimpleGA.GSimpleGA.evolve()` method. It will dump the statistics in the console.

Using the `sqlite3` DB Adapter

You can use the `DBAdapters.DBSQLite` DB Adapter and set the *commit_freq* to a low value, so you can use the Graphical Plotting Tool of Pyevolve to create graphics while evolving.

Using the `VPython` DB Adapter

Use the `DBAdapters.DBVPythonGraph` DB Adapter, this DB Adapter will show four statistical graphs, it is fast and easy to use.

How to manually add non-terminal functions to Genetic Programming core

When you set Pyevolve to automatically catch non-terminal functions for your GP core you do something like this:

```
ga = GSimpleGA.GSimpleGA(genome)
ga.setParams(gp_terminals = ['a', 'b'],
             gp_function_prefix = "gp")
```

The “gp_function_prefix” parameter tells Pyevolve to catch any function starting with “gp”. But there are times that you want to add each function manually, so you just need to add a dictionary parameter called “gp_function_set”, like this:

```
ga.setParams(gp_terminals = ['a', 'b'],
             gp_function_set = {"gp_add" :2,
                               "gp_sub" :2,
                               "gp_sqrt":1})
```

Note the “gp_function_set” dictionary parameter which holds as key the function name and for the value, the number of arguments from that function, in this case we have “gp_add” with 2 parameters, “gp_sub” with 2 and “gp_sqrt” with just one.

Passing extra parameters to the individual

Sometimes we want to add extra parameters which we need the individuals must carry, in this case, we can use the method `GenomeBase.GenomeBase.setParams()` to set internal parameters of the individual and the method `GenomeBase.GenomeBase.getParam()` to get it's parameters back, see an example:

```
def evaluation_function(genome):
    parameter_a = genome.getParam("parameter_a")

def main():
    # (...)
    genome = G1DList.G1DList(20)
    genome.setParams(rangemin=-5.2, rangemax=5.30, parameter_a="my_value")
    # (...)
```

Note: Due to performance issues, Pyevolve doesn't copy the internal parameters into each new created individual, it simple references the original parameters, this reduces memory and increases speed.

Using ephemeral constants in Genetic Programming

You can use an ephemeral constant in Pyevolve GP core by using the “ephemeral:” prefix in your GP terminals, like in:

```
ga = GSimpleGA.GSimpleGA(genome)
ga.setParams(gp_terminals = ['a', 'b', 'ephemeral:random.randint(1,10)'],
            gp_function_prefix = "gp")
```

In this example, the ephemeral constant will be an integer value between 1 and 10. You can use any method of the Python `random` module to specify the ephemeral constant.

2.5 Modules

Documentation of the all Pyevolve modules. All modules above listed are under the “pyevolve” namespace.

Contents:

2.5.1 `pyevolve` – the main pyevolve namespace

This is the main module of the pyevolve, every other module is above this namespace, for example, to import `Mutators`:

```
>>> from pyevolve import Mutators
```

2.5.2 General Modules

Contents:

Consts – constants module

Pyevolve have defaults in all genetic operators, settings and etc, this is an issue to helps the user in the API use and minimize the source code needed to make simple things. In the module `Consts`, you will find those defaults settings. You are encouraged to see the constants, but not to change directly on the module, there are methods for this.

General constants

CDefPythonRequire

The minimum version required to run Pyevolve.

CDefLogFile

The default log filename.

CDefLogLevel

Default log level.

sortType

Sort type, raw or scaled.

Example:

```
>>> sort_type = Consts.sortType["raw"]
>>> sort_type = Consts.sortType["scaled"]
```

minimaxType

The Min/Max type, maximize or minimize the evaluation function.

Example:

```
>>> minmax = Consts.minimaxType["minimize"]
>>> minmax = Consts.minimaxType["maximize"]
```

CDefESCKey

The ESC key ASCII code. Used to start Interactive Mode.

CDefRangeMin

Minimum range. This constant is used as integer and real max/min.

CDefRangeMax

Maximum range. This constant is used as integer and real max/min.

CDefBroadcastAddress

The broadcast address for UDP, 255.255.255.255

CDefImportList

The import list and messages

nodeType

The genetic programming node types, can be “TERMINAL”:0 or “NONTERMINAL”:1

CDefGPGenomes

The classes which are used in Genetic Programming, used to detected the correct mode when starting the evolution

Selection methods constants (Selectors)

CDefTournamentPoolSize

The default pool size for the Tournament Selector (`Selectors.GTournamentSelector()`).

Scaling scheme constants (Scaling)

CDefScaleLinearMultiplier

The multiplier of the Linear (`Scaling.LinearScaling()`) scaling scheme.

CDefScaleSigmaTruncMultiplier

The default Sigma Truncation (`Scaling.SigmaTruncScaling()`) scaling scheme.

CDefScalePowerLawFactor

The default Power Law (`Scaling.PowerLawScaling()`) scaling scheme factor.

CDefScaleBoltzMinTemp

The default minimum temperature of the (`Scaling.BoltzmannScaling()`) scaling scheme factor.

CDefScaleBoltzFactor

The default Boltzmann Factor of (`Scaling.BoltzmannScaling()`) scaling scheme factor. This is the factor that the temperature will be subtracted.

CDefScaleBoltzStart

The default Boltzmann start temperature (`Scaling.BoltzmannScaling()`). If you don't set the start temperature parameter, this will be the default initial temperature for the Boltzmann scaling scheme.

Population constants (GPopulation.GPopulation)

CDefPopSortType

Default sort type parameter.

CDefPopMinimax

Default min/max parameter.

CDefPopScale

Default scaling scheme.

1D Binary String Defaults (G1DBinaryString.G1DBinaryString)

CDefG1DBinaryStringMutator

The default mutator for the 1D Binary String (`G1DBinaryString.G1DBinaryString`) chromosome.

CDefG1DBinaryStringCrossover

The default crossover method for the 1D Binary String (`G1DBinaryString.G1DBinaryString`) chromosome.

CDefG1DBinaryStringInit

The default initializer for the 1D Binary String (`G1DBinaryString.G1DBinaryString`) chromosome.

CDefG1DBinaryStringUniformProb

The default uniform probability used for some uniform genetic operators for the 1D Binary String (`G1DBinaryString.G1DBinaryString`) chromosome.

2D Binary String Defaults (`G2DBinaryString.G2DBinaryString`)

`CDefG2DBinaryStringMutator`

The default mutator for the 2D Binary String (`G2DBinaryString.G2DBinaryString`) chromosome.

`CDefG2DBinaryStringCrossover`

The default crossover method for the 2D Binary String (`G2DBinaryString.G2DBinaryString`) chromosome.

`CDefG2DBinaryStringInit`

The default initializer for the 2D Binary String (`G2DBinaryString.G2DBinaryString`) chromosome.

`CDefG2DBinaryStringUniformProb`

The default uniform probability used for some uniform genetic operators for the 2D Binary String (`G2DBinaryString.G2DBinaryString`) chromosome.

1D List chromosome constants (`G1DList.G1DList`)

`CDefG1DListMutIntMU`

Default *mu* value of the 1D List Gaussian Integer Mutator (`Mutators.G1DListMutatorIntegerGaussian()`), the *mu* represents the mean of the distribution.

`CDefG1DListMutIntSIGMA`

Default *sigma* value of the 1D List Gaussian Integer Mutator (`Mutators.G1DListMutatorIntegerGaussian()`), the *sigma* represents the standard deviation of the distribution.

`CDefG1DListMutRealMU`

Default *mu* value of the 1D List Gaussian Real Mutator (`Mutators.G1DListMutatorRealGaussian()`), the *mu* represents the mean of the distribution.

`CDefG1DListMutRealSIGMA`

Default *sigma* value of the 1D List Gaussian Real Mutator (`Mutators.G1DListMutatorRealGaussian()`), the *sigma* represents the mean of the distribution.

Tree chromosome constants (`GTree.GTree`)

`CDefGTreeInit`

Default initializer of the tree chromosome.

`CDefGGTreeMutator`

Default mutator of the tree chromosome.

`CDefGTreeCrossover`

Default crossover of the tree chromosome.

2D List chromosome constants (`G2DList.G2DList`)

`CDefG2DListMutRealMU`

Default *mu* value of the 2D List Gaussian Real Mutator (`Mutators.G2DListMutatorRealGaussian()`), the *mu* represents the mean of the distribution.

`CDefG2DListMutRealSIGMA`

Default *sigma* value of the 2D List Gaussian Real Mutator (`Mutators.G2DListMutatorRealGaussian()`), the *sigma* represents the mean of the distribution.

CDefG2DListMutIntMU

Default *mu* value of the 2D List Gaussian Integer Mutator (`Mutators.G2DListMutatorIntegerGaussian()`), the *mu* represents the mean of the distribution.

CDefG2DListMutIntSIGMA

Default *sigma* value of the 2D List Gaussian Integer Mutator (`Mutators.G2DListMutatorIntegerGaussian()`), the *sigma* represents the mean of the distribution.

CDefG2DListMutator

Default mutator for the 2D List chromosome.

CDefG2DListCrossover

Default crossover method for the 2D List chromosome.

CDefG2DListInit

Default initializer for the 2D List chromosome.

CDefG2DListCrossUniformProb

Default uniform probability for the 2D List Uniform Crossover method (`Crossovers.G2DListCrossoverUniform()`).

GA Engine constants (`GSimpleGA.GSimpleGA`)

CDefGAGenerations

Default number of generations.

CDefGAMutationRate

Default mutation rate.

CDefGACrossoverRate

Default crossover rate.

CDefGAPopulationSize

Default population size.

CDefGASelector

Default selector method.

DB Adapters constants (`DBAdapters`)

Constants for the DB Adapters

SQLite3 DB Adapter Constants (`DBAdapters.DBSQLite`)

CDefSQLiteDBName

Default database filename.

CDefSQLiteDBTable

Default statistical table name.

CDefSQLiteDBTablePop

Default population statistical table name.

CDefSQLiteStatsGenFreq

Default generational frequency for dump statistics.

CDefSQLiteStatsCommitFreq

Default commit frequency.

MySQL DB Adapter Constants (DBAdapters.DBMySQLAdapter)**CDefMySQLDBName**

Default database name.

CDefMySQLDBTable

Default statistical table name.

CDefMySQLDBTablePop

Default population statistical table name.

CDefMySQLStatsGenFreq

Default generational frequency for dump statistics.

CDefMySQLStatsCommitFreq

Default commit frequency.

CDefMySQLDBHost

Default MySQL connection host.

CDefMySQLDBPort

Default MySQL connection TCP port.

URL Post DB Adapter Constants (DBAdapters.DBURLPost)**CDefURLPostStatsGenFreq**

Default generational frequency for dump statistics.

CSV File DB Adapter Constants (DBAdapters.DBFileCSV)**CDefCSVFileName**

The default CSV filename to dump statistics.

CDefCSVFileStatsGenFreq

Default generational frequency for dump statistics.

XMP RPC DB Adapter Constants (DBAdapters.DBXMLRPC)**CDefXMLRPCStatsGenFreq**

Default generational frequency for dump statistics.

Migration Constants (Migration)**CDefGenMigrationRate**

The default generations supposed to migrate and receive individuals

CDefMigrationNIndividuals

The default number of individuals that will migrate at the *CDefGenMigrationRate* interval

CDefNetworkIndividual

A migration code for network individual data

CDefNetworkInfo

A migration code for network info data

CDefGenMigrationReplacement

The default number of individuals to be replaced at the migration stage

Util – utility module

This is the utility module, with some utility functions of general use, like list item swap, random utilities and etc.

class ErrorAccumulator ()

An accumulator for the Root Mean Square Error (RMSE) and the Mean Square Error (MSE)

append (*target, evaluated*)

Add value to the accumulator

Parameters

- *target* – the target value
- *evaluated* – the evaluated value

getAdjusted ()

Returns the adjusted fitness This fitness is calculated as $1 / (1 + \text{standardized fitness})$

getMSE ()

Return the mean square error

Return type float MSE

getMean ()

Return the mean of the non-squared accumulator

getNonSquared ()

Returns the non-squared accumulator

getRMSE ()

Return the root mean square error

Return type float RMSE

getSquared ()

Returns the squared accumulator

reset ()

Reset the accumulator

G1DListGetEdges (*individual*)

Get the edges of a G1DList individual

Parameter *individual* – the G1DList individual

Return type the edges dictionary

G1DListGetEdgesComposite (*mom, dad*)

Get the edges and the merge between the edges of two G1DList individuals

Parameters

- *mom* – the mom G1DList individual
- *dad* – the dad G1DList individual

Return type a tuple (mom edges, dad edges, merge)

G1DListMergeEdges (*eda, edb*)

Get the merge between the two individual edges

Parameters

- *eda* – the edges of the first G1DList genome
- *edb* – the edges of the second G1DList genome

Return type the merged dictionary

class Graph ()

The Graph class

Example:

```
>>> g = Graph()
>>> g.addEdge("a", "b")
>>> g.addEdge("b", "c")
>>> for node in g:
...     print node
a
b
c
```

New in version 0.6: The *Graph* class.

addEdge (*a*, *b*)

Add an edge between two nodes, if the nodes doesn't exists, they will be created

Parameters

- *a* – the first node
- *b* – the second node

addNode (*node*)

Add the node

Parameter *node* – the node to add

getNeighbors (*node*)

Returns the neighbors of the node

Parameter *node* – the node

getNodes ()

Returns all the current nodes on the graph

Return type the list of nodes

reset ()

Deletes all nodes of the graph

cmp_individual_raw (*a*, *b*)

Compares two individual raw scores

Example:

```
>>> GPopulation.cmp_individual_raw(a, b)
```

Parameters

- *a* – the A individual instance
- *b* – the B individual instance

Return type 0 if the two individuals raw score are the same, -1 if the B individual raw score is greater than A and 1 if the A individual raw score is greater than B.

Note: this function is used to sorte the population individuals

cmp_individual_scaled (*a*, *b*)

Compares two individual fitness scores, used for sorting population

Example:

```
>>> GPopulation.cmp_individual_scaled(a, b)
```

Parameters

- *a* – the A individual instance
- *b* – the B individual instance

Return type 0 if the two individuals fitness score are the same, -1 if the B individual fitness score is greater than A and 1 if the A individual fitness score is greater than B.

Note: this function is used to sorte the population individuals

importSpecial (*name*)

This function will import the *name* module, if fails, it will raise an ImportError exception and a message

Parameter *name* – the module name

Return type the module object

New in version 0.6: The *import_special* function

list2DSwapElement (*lst*, *indexa*, *indexb*)

Swaps elements A and B in a 2D list (matrix).

Example:

```
>>> l = [ [1,2,3], [4,5,6] ]
>>> Util.list2DSwapElement(l, (0,1), (1,1) )
>>> l
[[1, 5, 3], [4, 2, 6]]
```

Parameters

- *lst* – the list
- *indexa* – the swap element A
- *indexb* – the swap element B

Return type None

listSwapElement (*lst*, *indexa*, *indexb*)

Swaps elements A and B in a list.

Example:

```
>>> l = [1, 2, 3]
>>> Util.listSwapElement(l, 1, 2)
>>> l
[1, 3, 2]
```

Parameters

- *lst* – the list
- *indexa* – the swap element A

- *indexb* – the swap element B

Return type None

raiseException (*message*, *expt=None*)

Raise an exception and logs the message.

Example:

```
>>> Util.raiseException('The value is not an integer', ValueError)
```

Parameters

- *message* – the message of exception
- *expt* – the exception class

Return type None

rand_random ()

random() -> x in the interval [0, 1).

randomFlipCoin (*p*)

Returns True with the *p* probability. If the *p* is 1.0, the function will always return True, or if is 0.0, the function will return always False.

Example:

```
>>> Util.randomFlipCoin(1.0)
True
```

Parameter *p* – probability, between 0.0 and 1.0

Return type True or False

Network – network utility module

In this module you'll find all the network related implementation New in version 0.6: The *Network* module.

class UDPThreadBroadcastClient (*host*, *port*, *target_port*)

The Broadcast UDP client thread class.

This class is a thread to serve as Pyevolve client on the UDP datagrams, it is used to send data over network lan/wan.

Example:

```
>>> s = Network.UDPThreadClient('192.168.0.2', 1500, 666)
>>> s.setData("Test data")
>>> s.start()
>>> s.join()
```

Parameters

- *host* – the hostname to bind the socket on sender (this is NOT the target host)
- *port* – the sender port (this is NOT the target port)
- *target_port* – the destination port target

close ()

Close the internal socket

getData ()

Get the data to send

Return type data to send

getSentBytes ()

Returns the number of sent bytes. The use of this method makes sense when you already have sent the data

Return type sent bytes

run ()

Method called when you call *.start()* of the thread

send ()

Broadcasts the data

setData (data)

Set the data to send

Parameter *data* – the data to send

class UDPThreadServer (host, port, poolSize=10, timeout=3)

The UDP server thread class.

This class is a thread to serve as Pyevolve server on the UDP datagrams, it is used to receive data from network lan/wan.

Example:

```
>>> s = UDPThreadServer("192.168.0.2", 666, 10)
>>> s.start()
>>> s.shutdown()
```

Parameters

- *host* – the host to bind the server
- *port* – the server port to bind
- *poolSize* – the size of the server pool
- *timeout* – the socket timeout

Note: this thread implements a pool to keep the received data, the *poolSize* parameter specifies how much individuals we must keep on the pool until the *popPool* method is called; when the pool is full, the sever will discard the received individuals.

close ()

Closes the internal socket

getBufferSize ()

Gets the current receive buffer size

Return type integer

getData ()

Calls the socket *recvfrom* method and waits for the data, when the data is received, the method will return a tuple with the IP of the sender and the data received. When a timeout exception occurs, the method return None.

Return type tuple (sender ip, data) or None when timeout exception

isReady ()

Returns True when there is data on the pool or False when not

Return type boolean

poolLength ()

Returns the size of the pool

Return type integer

popPool ()

Return the last data received on the pool

Return type object

run ()

Called when the thread is started by the user. This method is the main of the thread, when called, it will enter in loop to wait data or shutdown when needed.

setBufferSize (size)

Sets the receive buffer size

Parameter *size* – integer

shutdown ()

Shutdown the server thread, when called, this method will stop the thread on the next socket timeout

class UDPThreadUnicastClient (host, port, pool_size=10, timeout=0.5)

The Unicast UDP client thread class.

This class is a thread to serve as Pyevolve client on the UDP datagrams, it is used to send data over network lan/wan.

Example:

```
>>> s = Network.UDPThreadClient('192.168.0.2', 1500)
>>> s.setData("Test data")
>>> s.setTargetHost('192.168.0.50', 666)
>>> s.start()
>>> s.join()
```

Parameters

- *host* – the hostname to bind the socket on sender (this is not the target host)
- *port* – the sender port (this is not the target port)
- *pool_size* – the size of send pool
- *timeout* – the time interval to check if the client have data to send

addData (data)

Set the data to send

Parameter *data* – the data to send

close ()

Close the internal socket

isReady ()

Returns True when there is data on the pool or False when not

Return type boolean

poolLength ()

Returns the size of the pool

Return type integer

popPool ()

Return the last data received on the pool

Return type object

run ()

Method called when you call `.start()` of the thread

send (data)

Send the data

Parameter *data* – the data to send

Return type bytes sent to each destination

setMultipleTargetHost (address_list)

Sets multiple host/port targets, the destinations

Parameter *address_list* – a list with tuples (ip, port)

setTargetHost (host, port)

Set the host/port of the target, the destination

Parameters

- *host* – the target host
- *port* – the target port

Note: the host will be ignored when using broadcast mode

shutdown ()

Shutdown the server thread, when called, this method will stop the thread on the next socket timeout

getMachineIP ()

Return all the IPs from current machine.

Example:

```
>>> Util.getMachineIP ()
['200.12.124.181', '192.168.0.1']
```

Return type a python list with the string IPs

pickleAndCompress (obj, level=9)

Pickles the object and compress the dumped string with zlib

Parameters

- *obj* – the object to be pickled
- *level* – the compression level, 9 is the best and -1 is to not compress

unpickleAndDecompress (obj_dump, decompress=True)

Decompress a zlib compressed string and unpickle the data

Parameter *obj* – the object to be decompressend and unpickled

Migration – the migration schemes, distributed GA

This module contains all the migration schemes and the distributed GA related functions. New in version 0.6: The `Migration` module.

class MigrationScheme (*host, port, group_name*)

This is the base class for all migration schemes

Parameters

- *host* – the source hostname
- *port* – the source host port
- *group_name* – the group name

exchange ()

Exchange individuals

getCompressionLevel ()

Get the zlib compression level of network data

The values are in the interval described on the `Network.pickleAndCompress` ()

getGroupName ()

Gets the group name

Note: all islands of evolution which are supposed to exchange individuals, must have the same group name.

getMigrationRate ()

Return the the generation frequency supposed to migrate and receive individuals

Return type the number of generations

getNumIndividuals ()

Return the number of individuals that will migrate

Return type the number of individuals to be replaced

getNumReplacement ()

Return the number of individuals that will be replaced in the migration process

isReady ()

Returns true if is time to migrate

select ()

Pickes an individual from population using specific selection method

Return type an individual object

selectPool (*num_individuals*)

Select *num_individuals* number of individuals and return a pool

Parameter *num_individuals* – the number of individuals to select

Return type list with individuals

selector

This is the function slot for the selection method if you want to change the default selector, you must do this:

```
migration_scheme.selector.set(Selectors.GRouletteWheel)
```

setCompressionLevel (*level*)

Set the zlib compression level of network data

The values are in the interval described on the `Network.pickleAndCompress()`

Parameter *level* – the zlib compression level

setGAEngine (*ga_engine*)

Sets the GA Engine handler

setGroupName (*name*)

Sets the group name

Parameter *name* – the group name

Note: all islands of evolution which are supposed to exchange individuals, must have the same group name.

setMigrationRate (*generations*)

Sets the generation frequency supposed to migrate and receive individuals.

Parameter *generations* – the number of generations

setMyself (*host, port*)

Which interface you will use to send/receive data

Parameters

- *host* – your hostname
- *port* – your port

setNumIndividuals (*num_individuals*)

Set the number of individuals that will migrate

Parameter *num_individuals* – the number of individuals

setNumReplacement (*num_individuals*)

Return the number of individuals that will be replaced in the migration process

Parameter *num_individuals* – the number of individuals to be replaced

start ()

Initializes the migration scheme

stop ()

Stops the migration engine

class WANMigration (*host, port, group_name*)

This is the Simple Migration class for distributed GA

Example:

```
>>> mig = WANMigration("192.168.0.1", "10000", "group1")
```

Parameters

- *host* – the source hostname
- *port* – the source port number
- *group_name* – the group name

exchange ()

This is the main method, is where the individuals are exchanged

getCompressionLevel ()

Get the zlib compression level of network data

The values are in the interval described on the `Network.pickleAndCompress ()`

getGroupName ()

Gets the group name

Note: all islands of evolution which are supposed to exchange individuals, must have the same group name.

getMigrationRate ()

Return the the generation frequency supposed to migrate and receive individuals

Return type the number of generations

getNumIndividuals ()

Return the number of individuals that will migrate

Return type the number of individuals to be replaced

getNumReplacement ()

Return the number of individuals that will be replaced in the migration process

isReady ()

Returns true if is time to migrate

select ()

Pickes an individual from population using specific selection method

Return type an individual object

selectPool (num_individuals)

Select num_individuals number of individuals and return a pool

Parameter *num_individuals* – the number of individuals to select

Return type list with individuals

selector

This is the function slot for the selection method if you want to change the default selector, you must do this:

```
migration_scheme.selector.set(Selectors.GRouletteWheel)
```

setCompressionLevel (level)

Set the zlib compression level of network data

The values are in the interval described on the `Network.pickleAndCompress ()`

Parameter *level* – the zlib compression level

setGAEngine (ga_engine)

Sets the GA Engine handler

setGroupName (name)

Sets the group name

Parameter *name* – the group name

Note: all islands of evolution which are supposed to exchange individuals, must have the same group name.

setMigrationRate (generations)

Sets the generation frequency supposed to migrate and receive individuals.

Parameter *generations* – the number of generations

setMyself (*host, port*)

Which interface you will use to send/receive data

Parameters

- *host* – your hostname
- *port* – your port

setNumIndividuals (*num_individuals*)

Set the number of individuals that will migrate

Parameter *num_individuals* – the number of individuals

setNumReplacement (*num_individuals*)

Return the number of individuals that will be replaced in the migration process

Parameter *num_individuals* – the number of individuals to be replaced

setTopology (*graph*)

Sets the topology of the migrations

Parameter *graph* – the `Util.Graph` instance

start ()

Start capture of packets and initialize the migration scheme

stop ()

Stops the migration engine

Interaction – interaction module

In this module, you will find the functionality for the *Interactive mode*. When you enter in the Interactive Mode, Pyevolve will automatic import this module and exposes to you in the name space called “it”.

To use this mode, the parameter *interactiveMode* must be enabled in the `GSimpleGA.GSimpleGA`.

You can use the manual method to enter in the Interactive Mode at specific generation using the `GSimpleGA.GSimpleGA.setInteractiveGeneration()` method.

getPopScores (*population, fitness=False*)

Returns a list of population scores

Example:

```
>>> lst = Interaction.getPopScores(population)
```

Parameters

- *population* – population object (`GPopulation.GPopulation`)
- *fitness* – if is True, the fitness score will be used, otherwise, the raw.

Return type list of population scores

plotHistPopScore (*population, fitness=False*)

Population score distribution histogram

Example:

```
>>> Interaction.plotHistPopScore(population)
```

Parameters

- *population* – population object (`GPopulation.GPopulation`)
- *fitness* – if is True, the fitness score will be used, otherwise, the raw.

Return type None

plotPopScore (*population, fitness=False*)
Plot the population score distribution

Example:

```
>>> Interaction.plotPopScore(population)
```

Parameters

- *population* – population object (`GPopulation.GPopulation`)
- *fitness* – if is True, the fitness score will be used, otherwise, the raw.

Return type None

DBAdapters – database adapters for statistics

Warning: the use the of a DB Adapter can reduce the performance of the Genetic Algorithm.

Pyevolve have a feature in which you can save the statistics of every generation in a database, file or call an URL with the statistics as param. You can use the database to plot evolution statistics graphs later. In this module, you'll find the adapters above cited.

See Also:

Method `GSimpleGA.GSimpleGA.setDBAdapter()` DB Adapters are set in the `GSimpleGA` Class.

class `DBBaseAdapter` (*frequency, identify*)
DBBaseAdapter Class - The base class for all DB Adapters

If you want to create your own DB Adapter, you must subclass this class.

Parameter *frequency* – the the generational dump frequency

New in version 0.6: Added the `DBBaseAdapter` class.

`commitAndClose()`
This method is called at the end of the evolution, to closes the DB Adapter and commit the changes

`getIdentify()`
Return the statistics identify

Return type identify string

`getStatsGenFreq()`
Returns the frequency of statistical dump

Return type the generation interval of statistical dump

insert (*ga_engine*)

Insert the stats

Parameter *ga_engine* – the GA Engine

open (*ga_engine*)

This method is called one time to do the initialization of the DB Adapter

Parameter *ga_engine* – the GA Engine

setIdentify (*identify*)

Sets the identify of the statistics

Parameter *identify* – the id string

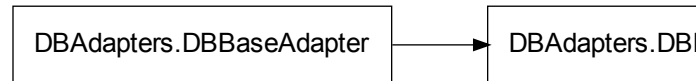
setStatsGenFreq (*statsGenFreq*)

Set the frequency of statistical dump

Parameter *statsGenFreq* – the generation interval of statistical dump

class DBFileCSV (*filename='pyevolve.csv', identify=None, frequency=1, reset=True*)

DBFileCSV Class - Adapter to dump statistics in CSV format



Inheritance diagram for `DBAdapters.DBFileCSV`:

Example:

```
>>> adapter = DBFileCSV(filename="file.csv", identify="run_01",
                        frequency = 1, reset = True)
```

param filename the CSV filename

param identify the identify of the run

param frequency the generational dump frequency

param reset if is True, the file old data will be overwrite with the new

New in version 0.6: Removed the stub methods and subclassed the `DBBaseAdapter` class.

close ()

Closes the CSV file handle

commitAndClose ()

Commits and closes

getIdentify ()

Return the statistics identify

Return type identify string

getStatsGenFreq ()

Returns the frequency of statistical dump

Return type the generation interval of statistical dump

insert (*ga_engine*)

Inserts the stats into the CSV file

Parameter *ga_engine* – the GA Engine

Changed in version 0.6: The method now receives the *ga_engine* parameter.

open (*ga_engine*)

Open the CSV file or creates a new file

Parameter *ga_engine* – the GA Engine

Changed in version 0.6: The method now receives the *ga_engine* parameter.

setIdentify (*identify*)

Sets the identify of the statistics

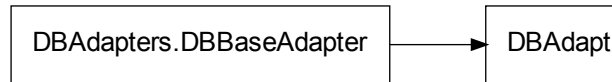
Parameter *identify* – the id string

setStatsGenFreq (*statsGenFreq*)

Set the frequency of statistical dump

Parameter *statsGenFreq* – the generation interval of statistical dump

class DBMySQLAdapter (*user, passwd, host='localhost', port=3306, db='pyevolve', identify=None, resetDB=False, resetIdentify=True, frequency=1, commit_freq=300*)
 DBMySQLAdapter Class - Adapter to dump data in MySQL database server



Inheritance diagram for `DBAdapters.DBMySQLAdapter`:

Example:

```
>>> dbadapter = DBMySQLAdapter("pyevolve_username", "password", identify="run1")
```

or

```
>>> dbadapter = DBMySQLAdapter(user="username", passwd="password",
...                             host="mysqlserver.com.br", port=3306, db="pyevolve_db")
```

When you run some GA for the first time, you need to create the database, for this, you must use the *resetDB* parameter as True.

This parameter will erase all the database tables and will create the new ones. The *resetDB* parameter is different from the *resetIdentify* parameter, the *resetIdentify* only erases the rows with the same “identify” name, and *resetDB* will drop and recreate the tables.

Parameters

- *user* – mysql username (must have permission to create, drop, insert, etc.. on tables)
- *passwd* – the user password on MySQL server
- *host* – the hostname, default is “localhost”
- *port* – the port, default is 3306

- *db* – the database name, default is “pyevolve”
- *identify* – the identify if the run
- *resetDB* – if True, the database structure will be recreated
- *resetIdentify* – if True, the identify with the same name will be overwrite with new data
- *frequency* – the generational dump frequency
- *commit_freq* – the commit frequency

close ()

Close the database connection

commit ()

Commit changes to database

commitAndClose ()

Commit changes on database and closes connection

createStructure (stats)

Create table using the Statistics class structure

Parameter *stats* – the statistics object

getCursor ()

Return a cursor from the pool

Return type the cursor

getIdentify ()

Return the statistics identify

Return type identify string

getStatsGenFreq ()

Returns the frequency of statistical dump

Return type the generation interval of statistical dump

insert (ga_engine)

Inserts the statistics data to database

Parameter *ga_engine* – the GA Engine

Changed in version 0.6: The method now receives the *ga_engine* parameter.

open (ga_engine)

Open the database connection

Parameter *ga_engine* – the GA Engine

Changed in version 0.6: The method now receives the *ga_engine* parameter.

resetStructure (stats)

Deletes de current structure and calls createStructure

Parameter *stats* – the statistics object

resetTableIdentify ()

Delete all records on the table with the same Identify

setIdentify (identify)

Sets the identify of the statistics

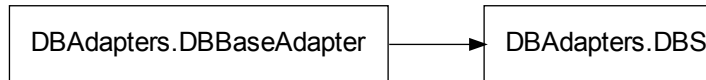
Parameter *identify* – the id string

setStatsGenFreq (*statsGenFreq*)
Set the frequency of statistical dump

Parameter *statsGenFreq* – the generation interval of statistical dump

class DBSQLite (*dbname='pyevolve.db', identify=None, resetDB=False, resetIdentify=True, frequency=1, commit_freq=300*)

DBSQLite Class - Adapter to dump data in SQLite3 database format



Inheritance diagram for `DBAdapters.DBSQLite`:

Example:

```
>>> dbadapter = DBSQLite(identify="test")
```

When you run some GA for the first time, you need to create the database, for this, you must use the *resetDB* parameter:

```
>>> dbadapter = DBSQLite(identify="test", resetDB=True)
```

This parameter will erase all the database tables and will create the new ones. The *resetDB* parameter is different from the *resetIdentify* parameter, the *resetIdentify* only erases the rows with the same “identify” name.

Parameters

- *dbname* – the database filename
- *identify* – the identify if the run
- *resetDB* – if True, the database structure will be recreated
- *resetIdentify* – if True, the identify with the same name will be overwrite with new data
- *frequency* – the generational dump frequency
- *commit_freq* – the commit frequency

close ()
Close the database connection

commit ()
Commit changes to database

commitAndClose ()
Commit changes on database and closes connection

createStructure (*stats*)
Create table using the Statistics class structure

Parameter *stats* – the statistics object

getCursor ()
Return a cursor from the pool

Return type the cursor

getIdentify()

Return the statistics identify

Return type identify string

getStatsGenFreq()

Returns the frequency of statistical dump

Return type the generation interval of statistical dump

insert(*ga_engine*)

Inserts the statistics data to database

Parameter *ga_engine* – the GA Engine

Changed in version 0.6: The method now receives the *ga_engine* parameter.

open(*ga_engine*)

Open the database connection

Parameter *ga_engine* – the GA Engine

Changed in version 0.6: The method now receives the *ga_engine* parameter.

resetStructure(*stats*)

Deletes de current structure and calls createStructure

Parameter *stats* – the statistics object

resetTableIdentify()

Delete all records on the table with the same Identify

setIdentify(*identify*)

Sets the identify of the statistics

Parameter *identify* – the id string

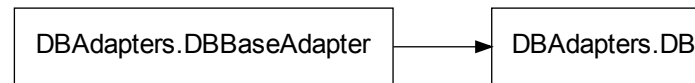
setStatsGenFreq(*statsGenFreq*)

Set the frequency of statistical dump

Parameter *statsGenFreq* – the generation interval of statistical dump

class DBURLPost(*url*, *identify=None*, *frequency=100*, *post=True*)

DBURLPost Class - Adapter to call an URL with statistics



Inheritance diagram for `DBAdapters.DBURLPost`:

Example:

```
>>> dbadapter = DBURLPost(url="http://localhost/post.py", identify="test")
```

The parameters that will be sent is all the statistics described in the `Statistics.Statistics` class, and the parameters:

generation The generation of the statistics

identify The id specified by user

Note: see the `Statistics.Statistics` documentation.

Parameters

- *url* – the URL to be used
- *identify* – the identify of the run
- *frequency* – the generational dump frequency
- *post* – if True, the POST method will be used, otherwise GET will be used.

New in version 0.6: Removed the stub methods and subclassed the `DBBaseAdapter` class.

commitAndClose ()

This method is called at the end of the evolution, to closes the DB Adapter and commit the changes

getIdentify ()

Return the statistics identify

Return type identify string

getStatsGenFreq ()

Returns the frequency of statistical dump

Return type the generation interval of statistical dump

insert (*ga_engine*)

Sends the data to the URL using POST or GET

Parameter *ga_engine* – the GA Engine

Changed in version 0.6: The method now receives the *ga_engine* parameter.

open (*ga_engine*)

Load the modules needed

Parameter *ga_engine* – the GA Engine

Changed in version 0.6: The method now receives the *ga_engine* parameter.

setIdentify (*identify*)

Sets the identify of the statistics

Parameter *identify* – the id string

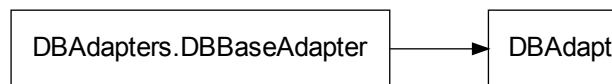
setStatsGenFreq (*statsGenFreq*)

Set the frequency of statistical dump

Parameter *statsGenFreq* – the generation interval of statistical dump

class DBVPythonGraph (*identify=None, frequency=20, genmax=False*)

The DBVPythonGraph Class - A DB Adapter for real-time visualization using VPython



Inheritance diagram for `DBAdapters.DBVPythonGraph`:

Note: to use this DB Adapter, you **must** install VPython first.

Example:

```
>>> adapter = DBAdapters.DBVPythonGraph(identify="run_01", frequency = 1)
>>> ga_engine.setDBAdapter(adapter)
```

Parameters

- *identify* – the identify of the run
- *genmax* – use the generations as max value for x-axis, default False
- *frequency* – the generational dump frequency

New in version 0.6: The *DBVPythonGraph* class.

commitAndClose ()

This method is called at the end of the evolution, to closes the DB Adapter and commit the changes

getIdentify ()

Return the statistics identify

Return type identify string

getStatsGenFreq ()

Returns the frequency of statistical dump

Return type the generation interval of statistical dump

insert (*ga_engine*)

Plot the current statistics to the graphs

Parameter *ga_engine* – the GA Engine

makeDisplay (*title_sec, x, y, ga_engine*)

Used internally to create a new display for VPython.

Parameters

- *title_sec* – the title of the window
- *x* – the x position of the window
- *y* – the y position of the window
- *ga_engine* – the GA Engine

Return type the window (the return of gdisplay call)

open (*ga_engine*)

Imports the VPython module and creates the four graph windows

Parameter *ga_engine* – the GA Engine

setIdentify (*identify*)

Sets the identify of the statistics

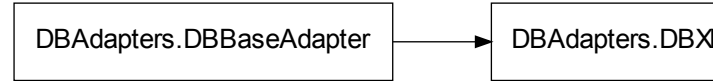
Parameter *identify* – the id string

setStatsGenFreq (*statsGenFreq*)

Set the frequency of statistical dump

Parameter *statsGenFreq* – the generation interval of statistical dump

class DBXMLRPC (*url, identify=None, frequency=20*)
 DBXMLRPC Class - Adapter to dump statistics to a XML Remote Procedure Call



Inheritance diagram for `DBAdapters.DBXMLRPC`:

Example:

```
>>> adapter = DBXMLRPC(url="http://localhost:8000/", identify="run_01",
                        frequency = 1)
```

- param url** the URL of the XML RPC
- param identify** the identify of the run
- param frequency** the generational dump frequency

Note: The XML RPC Server must implement the *insert* method, which receives a python dictionary as argument.

Example of an server in Python:

```
import xmlrpclib
from SimpleXMLRPCServer import SimpleXMLRPCServer

def insert(l):
    print "Received statistics: %s" % l

server = SimpleXMLRPCServer(("localhost", 8000), allow_none=True)
print "Listening on port 8000..."
server.register_function(insert, "insert")
server.serve_forever()
```

New in version 0.6: The `DBXMLRPC` class.

- commitAndClose()**
 This method is called at the end of the evolution, to closes the DB Adapter and commit the changes
- getIdentify()**
 Return the statistics identify
Return type identify string
- getStatsGenFreq()**
 Returns the frequency of statistical dump
Return type the generation interval of statistical dump
- insert(ga_engine)**
 Calls the XML RPC procedure
Parameter *ga_engine* – the GA Engine
 Changed in version 0.6: The method now receives the *ga_engine* parameter.

open (*ga_engine*)

Open the XML RPC Server proxy

Parameter *ga_engine* – the GA Engine

Changed in version 0.6: The method now receives the *ga_engine* parameter.

setIdentify (*identify*)

Sets the identify of the statistics

Parameter *identify* – the id string

setStatsGenFreq (*statsGenFreq*)

Set the frequency of statistical dump

Parameter *statsGenFreq* – the generation interval of statistical dump

FunctionSlot – function slots module

The *function slot* concept is large used by Pyevolve, the idea is simple, each genetic operator or any operator, can be assigned to a slot, by this way, we can add more than simple one operator, we can have for example, two or more mutator operators at same time, two or more evaluation functions, etc. In this `FunctionSlot` module, you'll find the class `FunctionSlot.FunctionSlot`, which is the slot class.

class FunctionSlot (*name='Anonymous Function', rand_apply=False*)

FunctionSlot Class - The function slot

Example:

```
>>> genome.evaluator.set(eval_func)
>>> genome.evaluator[0]
<function eval_func at 0x018C8930>
>>> genome.evaluator
Slot [Evaluation Function] (Count: 1)
      Name: eval_func
>>> genome.evaluator.clear()
>>> genome.evaluator
Slot [Evaluation Function] (Count: 0)
      No function
```

You can add weight to functions when using the *rand_apply* paramter:

```
>>> genome.evaluator.set(eval_main, 0.9)
>>> genome.evaluator.add(eval_sec, 0.3)
>>> genome.evaluator.setRandomApply()
```

In the above example, the function *eval_main* will be called with 90% of probability and the *eval_sec* will be called with 30% of probability.

There are another way to add functions too:

```
>>> genome.evaluator += eval_func
```

Parameters

- *name* – the slot name
- *rand_apply* – if True, just one of the functions in the slot will be applied, this function is randomly picked based on the weight of the function added.

add (*func*, *weight=0.5*)

Used to add a function to the slot

Parameters

- *func* – the function to be added in the slot
- *weight* – used when you enable the *random apply*, it's the weight of the function for the random selection

New in version 0.6: The *weight* parameter.

apply (*index*, *obj*, ***args*)

Apply the index function

Parameters

- *index* – the index of the function
- *obj* – this object is passes as parameter to the function
- *args* – this args dictionary is passed to the function

applyFunctions (*obj=None*, ***args*)

Generator to apply all function slots in obj

Parameters

- *obj* – this object is passes as parameter to the function
- *args* – this args dictionary is passed to the function

clear ()

Used to clear the functions in the slot

isEmpty ()

Return true if the function slot is empty

set (*func*, *weight=0.5*)

Used to clear all functions in the slot and add one

Parameters

- *func* – the function to be added in the slot
- *weight* – used when you enable the *random apply*, it's the weight of the function for the random selection

New in version 0.6: The *weight* parameter.

Note: the method *set* of the function slot remove all previous functions added to the slot.

setRandomApply (*flag=True*)

Sets the random function application, in this mode, the function will randomly choose one slot to apply

Parameter *flag* – True or False

Statistics – statistical structure module

This module have the class which is responsible to keep statistics of each generation. This class is used by the adapters and other statistics dump objects.

class Statistics ()

Statistics Class - A class bean-like to store the statistics

The statistics hold by this class are:

rawMax, rawMin, rawAve Maximum, minimum and average of raw scores

rawDev, rawVar Standard Deviation and Variance of raw scores

fitMax, fitMin, fitAve Maximum, minimum and average of fitness scores

rawTot, fitTot The total (sum) of raw scores and the fitness scores

Example:

```
>>> stats = ga_engine.getStatistics()
>>> st["rawMax"]
10.2
```

asTuple ()

Returns the stats as a python tuple

clear ()

Set all statistics to zero

clone ()

Instantiate a new Statistic class with the same contents

copy (obj)

Copy the values to the obj variable of the same class

Parameter *obj* – the Statistics object destination

items ()

Return a tuple (name, value) for all stored statistics

2.5.3 Genetic Algorithm Core Modules

GSimpleGA – the genetic algorithm by itself

This module contains the GA Engine, the GA Engine class is responsible for all the evolutionary process. It contains the GA Engine related functions, like the Termination Criteria functions for convergence analysis, etc.

Default Parameters

Number of Generations

Default is 100 generations

Mutation Rate

Default is 0.02, which represents 0.2%

Crossover Rate

Default is 0.9, which represents 90%

Elitism Replacement

Default is 1 individual

Population Size

Default is 80 individuals

Minimax

```
>>> Consts.minimaxType["maximize"]
```

Maximize the evaluation function

DB Adapter

Default is **None**

Migration Adapter

Default is **None**

Interactive Mode

Default is **True**

Selector (Selection Method)

```
Selectors.GRankSelector()
```

The Rank Selection method

Class

ConvergenceCriteria (*ga_engine*)

Terminate the evolution when the population have converged

Example:

```
>>> ga_engine.terminationCriteria.set(GSimpleGA.ConvergenceCriteria)
```

FitnessStatsCriteria (*ga_engine*)

Terminate the evolution based on the fitness stats

Example:

```
>>> ga_engine.terminationCriteria.set(GSimpleGA.FitnessStatsCriteria)
```

class GSimpleGA (*genome, seed=None, interactiveMode=True*)

GA Engine Class - The Genetic Algorithm Core

Example:

```
>>> ga = GSimpleGA.GSimpleGA(genome)
>>> ga.selector.set>Selectors.GRouletteWheel)
>>> ga.setGenerations(120)
>>> ga.terminationCriteria.set(GSimpleGA.ConvergenceCriteria)
```

Parameters

- *genome* – the *Sample Genome*
- *interactiveMode* – this flag enables the Interactive Mode, the default is True
- *seed* – the random seed value

Note: if you use the same random seed, all the runs of algorithm will be the same

bestIndividual ()

Returns the population best individual

Return type the best individual

clear ()

Petrowski's Clearing Method

dumpStatsDB ()

Dumps the current statistics to database adapter

evolve (*freq_stats=0*)

Do all the generations until the termination criteria, accepts the *freq_stats* (default is 0) to dump statistics at n-generation

Example:

```
>>> ga_engine.evolve(freq_stats=10)
(...)
```

Parameter *freq_stats* – if greater than 0, the statistics will be printed every *freq_stats* generation.

Return type returns the best individual of the evolution

New in version 0.6: the return of the best individual

getCurrentGeneration ()

Gets the current generation

Return type the current generation

getDBAdapter ()

Gets the DB Adapter of the GA Engine

Return type a instance from one of the `DBAdapters` classes

getGPMode ()

Get the Genetic Programming mode of the GA Engine

Return type True or False

getGenerations ()

Return the number of generations to evolve

Return type the number of generations

New in version 0.6: Added the *getGenerations* method

getInteractiveGeneration ()

returns the generation in which the GA must enter in the Interactive Mode

Return type the generation number or -1 if not set

getMinimax ()

Gets the minimize/maximize mode

Return type the `Consts.minimaxType` type

getParam (*key, nvl=None*)

Gets an internal parameter

Example:

```
>>> ga.getParam("gp_terminals")
['x', 'y']
```


Parameters

- *key* – the key of param
- *nvl* – if the key doesn't exist, the nvl will be returned

..versionadded:: 0.6 Added the *getParam* method.

getPopulation ()

Return the internal population of GA Engine

Return type the population (`GPopulation.GPopulation`)

getStatistics ()

Gets the Statistics class instance of current generation

Return type the statistics instance (`Statistics.Statistics`)

initialize ()

Initializes the GA Engine. Create and initialize population

printStats ()

Print generation statistics

Return type the printed statistics as string

Changed in version 0.6: The return of *printStats* method.

printTimeElapsed ()

Shows the time elapsed since the begin of evolution

select (args)**

Select one individual from population

Parameter *args* – this parameters will be sent to the selector

selector

This is the function slot for the selection method if you want to change the default selector, you must do this:

```
ga_engine.selector.set(Selectors.GRouletteWheel)
```

setCrossoverRate (rate)

Sets the crossover rate, between 0.0 and 1.0

Parameter *rate* – the rate, between 0.0 and 1.0

setDBAdapter (dbadapter=None)

Sets the DB Adapter of the GA Engine

Parameter *dbadapter* – one of the `DBAdapters` classes instance

Warning: the use the of a DB Adapter can reduce the speed performance of the Genetic Algorithm.

setElitism (flag)

Sets the elitism option, True or False

Parameter *flag* – True or False

setElitismReplacement (numreplace)

Set the number of best individuals to copy to the next generation on the elitism

Parameter *numreplace* – the number of individuals

New in version 0.6: The *setElitismReplacement* method.

setGPMode (*bool_value*)

Sets the Genetic Programming mode of the GA Engine

Parameter *bool_value* – True or False

setGenerations (*num_gens*)

Sets the number of generations to evolve

Parameter *num_gens* – the number of generations

setInteractiveGeneration (*generation*)

Sets the generation in which the GA must enter in the Interactive Mode

Parameter *generation* – the generation number, use “-1” to disable

setInteractiveMode (*flag=True*)

Enable/disable the interactive mode

Parameter *flag* – True or False

setMigrationAdapter (*migration_adapter=None*)

Sets the Migration Adapter New in version 0.6: The *setMigrationAdapter* method.

setMinimax (*mtype*)

Sets the minimize/maximize mode, use `Consts.minimaxType`

Parameter *mtype* – the minimax mode, from `Consts.minimaxType`

setMultiProcessing (*flag=True, full_copy=False*)

Sets the flag to enable/disable the use of python multiprocessing module. Use this option when you have more than one core on your CPU and when your evaluation function is very slow.

Pyevolve will automatically check if your Python version has **multiprocessing** support and if you have more than one single CPU core. If you don't have support or have just only one core, Pyevolve will not use the **multiprocessing** feature.

Pyevolve uses the **multiprocessing** to execute the evaluation function over the individuals, so the use of this feature will make sense if you have a truly slow evaluation function (which is common in GAs).

The parameter “full_copy” defines where the individual data should be copied back after the evaluation or not. This parameter is useful when you change the individual in the evaluation function.

Parameters

- *flag* – True (default) or False
- *full_copy* – True or False (default)

Warning: Use this option only when your evaluation function is slow, so you'll get a good tradeoff between the process communication speed and the parallel evaluation. The use of the **multiprocessing** doesn't mean always a better performance.

Note: To enable the multiprocessing option, you **MUST** add the `__main__` check on your application, otherwise, it will result in errors. See more on the [Python Docs](#) site. New in version 0.6: The *setMultiProcessing* method.

setMutationRate (*rate*)

Sets the mutation rate, between 0.0 and 1.0

Parameter *rate* – the rate, between 0.0 and 1.0

setParams (**args)

Set the internal params

Example:

```
>>> ga.setParams(gp_terminals=['x', 'y'])
```

Parameter *args* – params to save

..versionaddd:: 0.6 Added the *setParams* method.

setPopulationSize (*size*)

Sets the population size, calls setPopulationSize() of GPopulation

Parameter *size* – the population size

Note: the population size must be ≥ 2

setSortType (*sort_type*)

Sets the sort type, Consts.sortType["raw"]/Consts.sortType["scaled"]

Example:

```
>>> ga_engine.setSortType(Consts.sortType["scaled"])
```

Parameter *sort_type* – the Sort Type

step ()

Just do one step in evolution, one generation

stepCallback

This is the *step callback function* slot, if you want to set the function, you must do this:

```
def your_func(ga_engine):
    # Here you have access to the GA Engine
    return False
```

```
ga_engine.stepCallback.set(your_func)
```

now “*your_func*” will be called every generation. When this function returns True, the GA Engine will stop the evolution and show a warning, if is False, the evolution continues.

terminationCriteria

This is the termination criteria slot, if you want to set one termination criteria, you must do this:

```
ga_engine.terminationCriteria.set(GSimpleGA.ConvergenceCriteria)
```

Now, when you run your GA, it will stop when the population converges.

There are those termination criteria functions: `GSimpleGA.RawScoreCriteria()`,
`GSimpleGA.ConvergenceCriteria()`, `GSimpleGA.RawStatsCriteria()`,
`GSimpleGA.FitnessStatsCriteria()`

But you can create your own termination function, this function receives one parameter which is the GA Engine, follows an example:

```
def ConvergenceCriteria(ga_engine):
    pop = ga_engine.getPopulation()
    return pop[0] == pop[len(pop)-1]
```

When this function returns True, the GA Engine will stop the evolution and show a warning, if is False, the evolution continues, this function is called every generation.

RawScoreCriteria (*ga_engine*)

Terminate the evolution using the **bestrawscore** and **rounddecimal** parameter obtained from the individual

Example:

```
>>> genome.setParams(bestrawscore=0.00, rounddecimal=2)
(... )
>>> ga_engine.terminationCriteria.set(GSimpleGA.RawScoreCriteria)
```

RawStatsCriteria (*ga_engine*)

Terminate the evolution based on the raw stats

Example:

```
>>> ga_engine.terminationCriteria.set(GSimpleGA.RawStatsCriteria)
```

GPopulation – the population module

This module contains the `GPopulation.GPopulation` class, which is responsible to keep the population and the statistics.

Default Parameters

Sort Type

```
>>> Consts.sortType["scaled"]
```

The scaled sort type

Minimax

```
>>> Consts.minimaxType["maximize"]
```

Maximize the evaluation function

Scale Method

```
Scaling.LinearScaling()
```

The Linear Scaling scheme

Class

class GPopulation (*genome*)

GPopulation Class - The container for the population

Examples

Get the population from the `GSimpleGA.GSimpleGA` (GA Engine) instance

```
>>> pop = ga_engine.getPopulation()
```

Get the best fitness individual

```
>>> bestIndividual = pop.bestFitness()
```

Get the best raw individual

```
>>> bestIndividual = pop.bestRaw()
```

Get the statistics from the `Statistics.Statistics` instance

```
>>> stats = pop.getStatistics()
>>> print stats["rawMax"]
10.4
```

Iterate, get/set individuals

```
>>> for ind in pop:
>>>     print ind
(...)

>>> for i in xrange(len(pop)):
>>>     print pop[i]
(...)

>>> pop[10] = newGenome
>>> pop[10].fitness
12.5
```

Parameter *genome* – the *Sample genome*, or a `GPopulation` object, when cloning.

bestFitness (*index=0*)

Return the best scaled fitness individual of population

Parameter *index* – the *index* best individual

Return type the individual

bestRaw (*index=0*)

Return the best raw score individual of population

Parameter *index* – the *index* best raw individual

Return type the individual

New in version 0.6: The parameter *index*.

clear ()

Remove all individuals from population

clearFlags ()

Clear the sorted and statted internal flags

clone ()

Return a brand-new cloned population

copy (*pop*)

Copy current population to ‘pop’

Parameter *pop* – the destination population

Warning: this method do not copy the individuals, only the population logic

create (***args*)

Clone the example genome to fill the population

evaluate (***args*)

Evaluate all individuals in population, calls the evaluate() method of individuals

Parameter *args* – this params are passed to the evaluation function

getParam (*key, nvl=None*)

Gets an internal parameter

Example:

```
>>> population.getParam("tournamentPool")
5
```

Parameters

- *key* – the key of param
- *nvl* – if the key doesn’t exist, the nvl will be returned

getStatistics ()

Return a Statistics class for statistics

Return type the `Statistics.Statistics` instance

initialize (***args*)

Initialize all individuals of population, this calls the initialize() of individuals

printStats ()

Print statistics of the current population

scale (***args*)

Scale the population using the scaling method

Parameter *args* – this parameter is passed to the scale method

setMinimax (*minimax*)

Sets the population minimax

Example:

```
>>> pop.setMinimax(Consts.minimaxType["maximize"])
```

Parameter *minimax* – the minimax type

setMultiProcessing (*flag=True, full_copy=False*)

Sets the flag to enable/disable the use of python multiprocessing module. Use this option when you have more than one core on your CPU and when your evaluation function is very slow. The parameter “full_copy” defines where the individual data should be copied back after the evaluation or not. This parameter is useful when you change the individual in the evaluation function.

Parameters

- *flag* – True (default) or False
- *full_copy* – True or False (default)

Warning: Use this option only when your evaluation function is slow, se you will get a good tradeoff between the process communication speed and the parallel evaluation.

New in version 0.6: The *setMultiProcessing* method.

setParams (**args)

Gets an internal parameter

Example:

```
>>> population.setParams(tournamentPool=5)
```

Parameter *args* – parameters to set

New in version 0.6: The *setParams* method.

setPopulationSize (*size*)

Set the population size

Parameter *size* – the population size

setSortType (*sort_type*)

Sets the sort type

Example:

```
>>> pop.setSortType(Consts.sortType["scaled"])
```

Parameter *sort_type* – the Sort Type

sort ()

Sort the population

statistics ()

Do statistical analysis of population and set ‘statted’ to True

key_fitness_score (*individual*)

A key function to return fitness score, used by max()/min()

Parameter *individual* – the individual instance

Return type the individual fitness score

Note: this function is used by the max()/min() python functions

key_raw_score (*individual*)

A key function to return raw score

Parameter *individual* – the individual instance

Return type the individual raw score

Note: this function is used by the max()/min() python functions

multiprocessing_eval (*ind*)

Internal used by the multiprocessing

multiprocessing_eval_full (*ind*)

Internal used by the multiprocessing (full copy)

2.5.4 Genetic Operators Modules

Mutators – mutation methods module

In this module we have the genetic operators of mutation for each chromosome representation.

G1DBinaryStringMutatorFlip (*genome*, ***args*)

The classical flip mutator for binary strings

G1DBinaryStringMutatorSwap (*genome*, ***args*)

The 1D Binary String Swap Mutator

G1DListMutatorAllele (*genome*, ***args*)

The mutator of G1DList, Allele Mutator

To use this mutator, you must specify the *allele* genome parameter with the `GAllele.GAlleles` instance.

G1DListMutatorIntegerBinary (*genome*, ***args*)

The mutator of G1DList, the binary mutator

This mutator will random change the 0 and 1 elements of the 1D List.

G1DListMutatorIntegerGaussian (*genome*, ***args*)

A gaussian mutator for G1DList of Integers

Accepts the *rangemin* and *rangemax* genome parameters, both optional. Also accepts the parameter *gauss_mu* and the *gauss_sigma* which respectively represents the mean and the std. dev. of the random distribution.

G1DListMutatorIntegerRange (*genome*, ***args*)

Simple integer range mutator for G1DList

Accepts the *rangemin* and *rangemax* genome parameters, both optional.

G1DListMutatorRealGaussian (*genome*, ***args*)

The mutator of G1DList, Gaussian Mutator

Accepts the *rangemin* and *rangemax* genome parameters, both optional. Also accepts the parameter *gauss_mu* and the *gauss_sigma* which respectively represents the mean and the std. dev. of the random distribution.

G1DListMutatorRealRange (*genome*, ***args*)

Simple real range mutator for G1DList

Accepts the *rangemin* and *rangemax* genome parameters, both optional.

G1DListMutatorSIM (*genome*, ***args*)

The mutator of G1DList, Simple Inversion Mutation

Note: this mutator is *Data Type Independent*

G1DListMutatorSwap (*genome*, ***args*)

The mutator of G1DList, Swap Mutator

Note: this mutator is *Data Type Independent*

G2DBinaryStringMutatorFlip (*genome*, ***args*)

A flip mutator for G2DBinaryString New in version 0.6: The `G2DBinaryStringMutatorFlip` function

G2DBinaryStringMutatorSwap (*genome*, ***args*)

The mutator of G2DBinaryString, Swap Mutator New in version 0.6: The `G2DBinaryStringMutatorSwap` function

G2DListMutatorAllele (*genome*, ***args*)

The mutator of G2DList, Allele Mutator

To use this mutator, you must specify the *allele* genome parameter with the `GAllele.GAlleles` instance.

Warning: the `GAllele.GAlleles` instance must have the homogeneous flag enabled

G2DListMutatorIntegerGaussian (*genome*, ***args*)

A gaussian mutator for G2DList of Integers

Accepts the *rangemin* and *rangemax* genome parameters, both optional. Also accepts the parameter *gauss_mu* and the *gauss_sigma* which respectively represents the mean and the std. dev. of the random distribution.

G2DListMutatorIntegerRange (*genome*, ***args*)

Simple integer range mutator for G2DList

Accepts the *rangemin* and *rangemax* genome parameters, both optional.

G2DListMutatorRealGaussian (*genome*, ***args*)

A gaussian mutator for G2DList of Real

Accepts the *rangemin* and *rangemax* genome parameters, both optional. Also accepts the parameter *gauss_mu* and the *gauss_sigma* which respectively represents the mean and the std. dev. of the random distribution.

G2DListMutatorSwap (*genome*, ***args*)

The mutator of G1DList, Swap Mutator

Note: this mutator is *Data Type Independent*

GTreeGPMutatorOperation (*genome*, ***args*)

The mutator of GTreeGP, Operation Mutator New in version 0.6: The *GTreeGPMutatorOperation* function

GTreeGPMutatorSubtree (*genome*, ***args*)

The mutator of GTreeGP, Subtree Mutator

This mutator will recreate random subtree of the tree using the grow algorithm. New in version 0.6: The *GTreeGPMutatorSubtree* function

GTreeMutatorIntegerGaussian (*genome*, ***args*)

A gaussian mutator for GTree of Integers

Accepts the *rangemin* and *rangemax* genome parameters, both optional. Also accepts the parameter *gauss_mu* and the *gauss_sigma* which respectively represents the mean and the std. dev. of the random distribution.

GTreeMutatorIntegerRange (*genome*, ***args*)

The mutator of GTree, Integer Range Mutator

Accepts the *rangemin* and *rangemax* genome parameters, both optional. New in version 0.6: The *GTreeMutatorIntegerRange* function

GTreeMutatorRealGaussian (*genome*, ***args*)

A gaussian mutator for GTree of Real numbers

Accepts the *rangemin* and *rangemax* genome parameters, both optional. Also accepts the parameter *gauss_mu* and the *gauss_sigma* which respectively represents the mean and the std. dev. of the random distribution.

GTreeMutatorRealRange (*genome*, ***args*)

The mutator of GTree, Real Range Mutator

Accepts the *rangemin* and *rangemax* genome parameters, both optional. New in version 0.6: The *GTreeMutatorRealRange* function

GTreeMutatorSwap (*genome*, ***args*)

The mutator of GTree, Swap Mutator New in version 0.6: The *GTreeMutatorSwap* function

Crossovers – crossover methods module

In this module we have the genetic operators of crossover (or recombination) for each chromosome representation.

G1DBinaryStringXSinglePoint (*genome*, ***args*)

The crossover of 1D Binary String, Single Point

Warning: You can't use this crossover method for binary strings with length of 1.

G1DBinaryStringXTwoPoint (*genome*, ***args*)

The 1D Binary String crossover, Two Point

Warning: You can't use this crossover method for binary strings with length of 1.

G1DBinaryStringXUniform (*genome*, ***args*)

The G1DList Uniform Crossover

G1DListCrossoverCutCrossfill (*genome*, ***args*)

The crossover of G1DList, Cut and crossfill, for permutations

G1DListCrossoverEdge (*genome*, ***args*)

The Edge Recombination crossover for G1DList (widely used for TSP problem)

See more information in the [Edge Recombination Operator Wikipedia](#) entry.

G1DListCrossoverOX (*genome*, ***args*)

The OX Crossover for G1DList (order crossover)

G1DListCrossoverRealSBX (*genome*, ***args*)

Experimental SBX Implementation - Follows the implementation in NSGA-II (Deb, et.al)

Some implementation [reference](#).

Warning: This crossover method is Data Type Dependent, which means that must be used for 1D genome of real values.

G1DListCrossoverSinglePoint (*genome*, ***args*)

The crossover of G1DList, Single Point

Warning: You can't use this crossover method for lists with just one element.

G1DListCrossoverTwoPoint (*genome*, ***args*)

The G1DList crossover, Two Point

Warning: You can't use this crossover method for lists with just one element.

G1DListCrossoverUniform (*genome*, ***args*)

The G1DList Uniform Crossover

G2DBinaryStringXSingleHPoint (*genome*, ***args*)

The crossover of G2DBinaryString, Single Horizontal Point New in version 0.6: The *G2DBinaryStringXSingleHPoint* function

G2DBinaryStringXSingleVPoint (*genome*, ***args*)

The crossover of G2DBinaryString, Single Vertical Point New in version 0.6: The *G2DBinaryStringXSingleVPoint* function

G2DBinaryStringXUniform (*genome*, ***args*)

The G2DBinaryString Uniform Crossover New in version 0.6: The *G2DBinaryStringXUniform* function

G2DListCrossoverSingleHPoint (*genome*, ***args*)
 The crossover of G2DList, Single Horizontal Point

G2DListCrossoverSingleVPoint (*genome*, ***args*)
 The crossover of G2DList, Single Vertical Point

G2DListCrossoverUniform (*genome*, ***args*)
 The G2DList Uniform Crossover

GTreeCrossoverSinglePoint (*genome*, ***args*)
 The crossover for GTree, Single Point

GTreeCrossoverSinglePointStrict (*genome*, ***args*)
 The crossover of Tree, Strict Single Point

..note:: This crossover method creates offspring with restriction of the *max_depth* parameter.

Accepts the *max_attempt* parameter, *max_depth* (required), and the *distr_leaf* (≥ 0.0 and ≤ 1.0), which represents the probability of leaf selection when finding random nodes for crossover.

GTreeGPCrossoverSinglePoint (*genome*, ***args*)
 The crossover of the GTreeGP, Single Point for Genetic Programming

..note:: This crossover method creates offspring with restriction of the *max_depth* parameter.

Accepts the *max_attempt* parameter, *max_depth* (required).

rand_random ()
 random() -> x in the interval [0, 1).

Initializers – initialization methods module

In this module we have the genetic operators of initialization for each chromosome representation, the most part of initialization is done by choosing random data.

Note: In Pyevolve, the Initializer defines the data type that will be used on the chromosome, for example, the `G1DListInitializerInteger()` will initialize the G1DList with Integers.

G1DBinaryStringInitializer (*genome*, ***args*)
 1D Binary String initializer

G1DListInitializerAllele (*genome*, ***args*)
 Allele initialization function of G1DList

To use this initializer, you must specify the *allele* genome parameter with the `GAllele.GAlleles` instance.

G1DListInitializerInteger (*genome*, ***args*)
 Integer initialization function of G1DList

This initializer accepts the *rangemin* and *rangemax* genome parameters.

G1DListInitializerReal (*genome*, ***args*)
 Real initialization function of G1DList

This initializer accepts the *rangemin* and *rangemax* genome parameters.

G2DBinaryStringInitializer (*genome*, ***args*)
 Integer initialization function of 2D Binary String New in version 0.6: The `G2DBinaryStringInitializer` function

G2DListInitializerAllele (*genome*, ***args*)
 Allele initialization function of G2DList

To use this initializer, you must specify the *allele* genome parameter with the `GAllele.GAlleles` instance.

Warning: the `GAAllele.GAlleles` instance must have the homogeneous flag enabled

G2DListInitializerInteger (*genome*, ***args*)

Integer initialization function of G2DList

This initializer accepts the *rangemin* and *rangemax* genome parameters.

G2DListInitializerReal (*genome*, ***args*)

Integer initialization function of G2DList

This initializer accepts the *rangemin* and *rangemax* genome parameters.

GTreeGPInitializer (*genome*, ***args*)

This initializer accepts the follow parameters:

max_depth The max depth of the tree

method The method, accepts “grow”, “full” or “ramped”

New in version 0.6: The *GTreeGPInitializer* function.

GTreeInitializerAllele (*genome*, ***args*)

Allele initialization function of GTree

To use this initializer, you must specify the *allele* genome parameter with the `GAAllele.GAlleles` instance.

Warning: the `GAAllele.GAlleles` instance **must** have the homogeneous flag enabled

New in version 0.6: The *GTreeInitializerAllele* function.

GTreeInitializerInteger (*genome*, ***args*)

Integer initialization function of GTree

This initializer accepts the *rangemin* and *rangemax* genome parameters. It accepts the following parameters too:

max_depth The max depth of the tree

max_siblings The number of maximum siblings of an node

method The method, accepts “grow”, “full” or “ramped”.

New in version 0.6: The *GTreeInitializerInteger* function.

Selectors – selection methods module

This module have the *selection methods*, like roulette wheel, tournament, ranking, etc.

GRankSelector (*population*, ***args*)

The Rank Selector - This selector will pick the best individual of the population every time.

GRouletteWheel (*population*, ***args*)

The Roulette Wheel selector

GRouletteWheel_PrepareWheel (*population*)

A preparation for Roulette Wheel selection

GTournamentSelector (*population*, ***args*)

The Tournament Selector

It accepts the *tournamentPool* population parameter.

Note: the Tournament Selector uses the Roulette Wheel to pick individuals for the pool Changed in version 0.6: Changed the parameter *poolSize* to the *tournamentPool*, now the selector gets the pool size from the population.

GTournamentSelectorAlternative (*population*, ***args*)

The alternative Tournament Selector

This Tournament Selector don't uses the Roulette Wheel

It accepts the *tournamentPool* population parameter.

GUniformSelector (*population*, ***args*)

The Uniform Selector

key_fitness_score (*individual*)

A key function to return fitness score, used by max()/min()

Parameter *individual* – the individual instance

Return type the individual fitness score

Note: this function is used by the max()/min() python functions

key_raw_score (*individual*)

A key function to return raw score

Parameter *individual* – the individual instance

Return type the individual raw score

Note: this function is used by the max()/min() python functions

Scaling – scaling schemes module

This module have the *scaling schemes* like Linear scaling, etc.

BoltzmannScaling (*pop*)

Boltzmann scaling scheme. You can specify the **boltz_temperature** to the population parameters, this parameter will set the start temperature. You can specify the **boltz_factor** and the **boltz_min** parameters, the **boltz_factor** is the value that the temperature will be subtracted and the **boltz_min** is the minimum temperature of the scaling scheme.

ExponentialScaling (*pop*)

Exponential Scaling Scheme. The fitness will be the same as (e^{score}).

LinearScaling (*pop*)

Linear Scaling scheme

Warning: Linear Scaling is only for positive raw scores

PowerLawScaling (*pop*)

Power Law scaling scheme

Warning: Power Law Scaling is only for positive raw scores

SaturatedScaling (*pop*)

Saturated Scaling Scheme. The fitness will be the same as 1.0-(e^{score})

SigmaTruncScaling (*pop*)

Sigma Truncation scaling scheme, allows negative scores

2.5.5 Chromosomes/Representation Modules

GenomeBase – the genomes base module

This module have the class which every representation extends, if you are planning to create a new representation, you must take a inside look into this module.

class `G1DBase` (*size*)

G1DBase Class - The base class for 1D chromosomes

Parameter *size* – the 1D list size

New in version 0.6: Added to *G1DBase* class

append (*value*)

Appends an item to the end of the list

Example:

```
>>> genome.append(44)
```

Parameter *value* – value to be added

clearList ()

Remove all genes from Genome

copy (*g*)

Copy genome to 'g'

Example:

```
>>> genome_origin.copy(genome_destination)
```

Parameter *g* – the destination instance

getInternalList ()

Returns the internal list of the genome

... note:: this method was created to solve performance issues :rtype: the internal list

getListSize ()

Returns the list supposed size

Warning: this is different from what the len(obj) returns

remove (*value*)

Removes an item from the list

Example:

```
>>> genome.remove(44)
```

Parameter *value* – value to be added

resumeString ()

Returns a resumed string representation of the Genome

setInternalList (*lst*)

Assigns a list to the internal list of the chromosome

Parameter *lst* – the list to assign the internal list of the chromosome

class GTreeBase (*root_node*)

GTreeBase Class - The base class for the tree genomes

Parameter *root_node* – the root node of the tree

New in version 0.6: Added to *GTreeBase* class

clone ()

Clone this GenomeBase

Return type the clone genome

Note: If you are planning to create a new chromosome representation, you **must** implement this method on your class.

copy (*g*, *node=None*, *node_parent=None*)

Copy the current contents GTreeBase to 'g'

Parameter *g* – the destination GTreeBase tree

Note: If you are planning to create a new chromosome representation, you **must** implement this method on your class.

getAllNodes ()

Return a new list with all nodes

Return type the list with all nodes

getHeight ()

Return the tree height

Return type the tree height

getNodeDepth (*node*)

Returns the depth of a node

Return type the depth of the node, the depth of root node is 0

getNodeHeight (*node*)

Returns the height of a node

Note: If the node has no childs, the height will be 0.

Return type the height of the node

getNodeCount (*start_node=None*)

Return the number of the nodes on the tree starting at the *start_node*, if *start_node* is None, then the method will count all the tree nodes.

Return type the number of nodes

getRandomNode (*node_type=0*)

Returns a random node from the Tree

Parameter *node_type* – 0 = Any, 1 = Leaf, 2 = Branch

Return type random node

getRoot ()

Return the tree root node

Return type the tree root node

getTraversalString (*start_node=None, spc=0*)

Returns a tree-formated string of the tree. This method is used by the `__repr__` method of the tree

Return type a string representing the tree

processNodes (*cloning=False*)

Creates a *cache* on the tree, this method must be called every time you change the shape of the tree. It updates the internal nodes list and the internal nodes properties such as depth and height.

setRoot (*root*)

Sets the root of the tree

Parameter *root* – the tree root node

traversal (*callback, start_node=None*)

Traversal the tree, this method will call the user-defined callback function for each node on the tree

Parameters

- *callback* – a function
- *start_node* – the start node to begin the traversal

class GTreeNodeBase (*parent, childs=None*)

GTreeNodeBase Class - The base class for the node tree genomes

Parameters

- *parent* – the parent node of the node
- *childs* – the childs of the node, must be a list of nodes

New in version 0.6: Added to *GTreeNodeBase* class

addChild (*child*)

Adds a child to the node

Parameter *child* – the node to be added

clone ()

Clone this GenomeBase

Return type the clone genome

Note: If you are planning to create a new chromosome representation, you **must** implement this method on your class.

copy (*g*)

Copy the current contents GTreeNodeBase to 'g'

Parameter *g* – the destination node

Note: If you are planning to create a new chromosome representation, you **must** implement this method on your class.

getChild (*index*)

Returns the index-child of the node

Return type child node

getChilds ()

Return the childs of the node

<p>Warning: use <code>.getChilds()[:]</code> if you'll change the list itself, like using <code>childs.reverse()</code>, otherwise the original genome child order will be changed.</p>
--

Return type a list of nodes

getParent ()

Get the parent node of the node

Return type the parent node

isLeaf ()

Return True if the node is a leaf

Return type True or False

replaceChild (*older, newer*)

Replaces a child of the node

Parameters

- *older* – the child to be replaces
- *newer* – the new child which replaces the older

setParent (*parent*)

Sets the parent of the node

Parameter *parent* – the parent node

class GenomeBase ()

GenomeBase Class - The base of all chromosome representation

clone ()

Clone this GenomeBase

Return type the clone genome

Note: If you are planning to create a new chromosome representation, you **must** implement this method on your class.

copy (*g*)

Copy the current GenomeBase to 'g'

Parameter *g* – the destination genome

Note: If you are planning to create a new chromosome representation, you **must** implement this method on your class.

crossover

This is the reproduction function slot, the crossover. You can change the default crossover method using:

```
genome.crossover.set(Crossovers.G1DListCrossoverUniform)
```

evaluate (***args*)

Called to evaluate genome

Parameter *args* – this parameters will be passes to the evaluator

evaluator

This is the *evaluation function* slot, you can add a function with the *set* method:

```
genome.evaluator.set(eval_func)
```

getFitnessScore ()

Get the Fitness Score of the genome

Return type genome fitness score

getParam (*key*, *nvl=None*)
Gets an internal parameter

Example:

```
>>> genome.getParam("rangemax")
100
```

Note: All the individuals of the population shares this parameters and uses the same instance of this dict.

Parameters

- *key* – the key of param
- *nvl* – if the key doesn't exist, the *nvl* will be returned

getRawScore ()
Get the Raw Score of the genome

Return type genome raw score

initializer

This is the initialization function of the genome, you can change the default initializer using the function slot:

```
genome.initializer.set(Initializers.G1DListInitializerAllele)
```

In this example, the initializer `Initializers.G1DListInitializerAllele()` will be used to create the initial population.

initialize (***args*)
Called to initialize genome

Parameter *args* – this parameters will be passed to the initializer

mutate (***args*)
Called to mutate the genome

Parameter *args* – this parameters will be passed to the mutator

Return type the number of mutations returned by mutation operator

mutator

This is the mutator function slot, you can change the default mutator using the slot *set* function:

```
genome.mutator.set(Mutators.G1DListMutatorSwap)
```

resetStats ()
Clear score and fitness of genome

setParams (***args*)
Set the internal params

Example:

```
>>> genome.setParams(rangemin=0, rangemax=100, gauss_mu=0, gauss_sigma=1)
```

Note: All the individuals of the population shares this parameters and uses the same instance of this dict.

Parameter *args* – this params will saved in every chromosome for genetic op. use

GAllele – the genome alleles module

In this module, there are the `GAllele.GAlleles` class (which is the class that holds the allele types) and all the allele types to use with the supported chromosomes.

class GAlleleList (*options=None*)
GAlleleList Class - The list allele type

Example:

```
>>> alleles = GAlleles()
>>> choices = [1,2,3,4]
>>> lst = GAlleleList(choices)
>>> alleles.add(lst)
>>> alleles[0].getRandomAllele() in lst
True
```

add (*option*)

Appends one option to the options list

Parameter *option* – option to be added in the list

clear ()

Removes all the allele options from the list

getRandomAllele ()

Returns one random choice from the options list

remove (*option*)

Removes the option from list

Parameter *option* – remove the option from the list

class GAlleleRange (*begin=0, end=100, real=False*)
GAlleleRange Class - The range allele type

Example:

```
>>> ranges = GAlleleRange(0,100)
>>> ranges.getRandomAllele() >= 0 and ranges.getRandomAllele() <= 100
True
```

Parameters

- *begin* – the begin of the range
- *end* – the end of the range
- *real* – if True, the range will be of real values

add (*begin, end*)

Add a new range

Parameters

- *begin* – the begin of range
- *end* – the end of the range

clear ()

Removes all ranges

getMaximum()

Return the maximum of all the ranges

Return type the maximum value

getMinimum()

Return the minimum of all the ranges

Return type the minimum value

getRandomAllele()

Returns one random choice between the range

getReal()

Returns True if the range is real or False if it is integer

setReal(flag=True)

Sets True if the range is real or False if is integer

Parameter *flag* – True or False

class GAlleles (*allele_list=None, homogeneous=False*)

GAlleles Class - The set of alleles

Example:

```
>>> alleles = GAlleles()
>>> choices = [1,2,3,4]
>>> lst = GAlleleList(choices)
>>> alleles.add(lst)
>>> alleles[0].getRandomAllele() in lst
True
```

Parameters

- *allele_list* – the list of alleles
- *homogeneous* – if is True, all the alleles will be use only the first added

add(allele)

Appends one allele to the alleles list

Parameter *allele* – allele to be added

G1DBinaryString – the classical binary string chromosome

This is the classical chromosome representation on GAs, it is the 1D Binary String. This string looks like “00011101010”.

Default Parameters

Initializer

```
Initializers.G1DBinaryStringInitializer()
```

The Binatry String Initializer for G1DBinaryString

Mutator

```
Mutators.G1DBinaryStringMutatorFlip()
```

The Flip Mutator for G1DBinaryString

Crossover

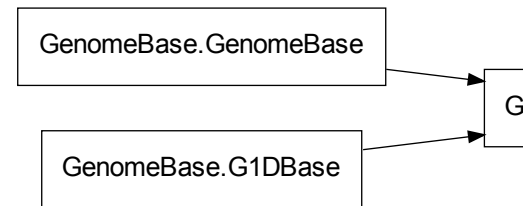
```
Crossovers.G1DBinaryStringXSinglePoint()
```

The Single Point Crossover for G1DBinaryString

Class

```
class G1DBinaryString (length=10)
```

G1DBinaryString Class - The 1D Binary String chromosome



Inheritance diagram for G1DBinaryString.G1DBinaryString:

This chromosome class extends the GenomeBase.GenomeBase and GenomeBase.G1DBase classes.

Example:

```
>>> genome = G1DBinaryString.G1DBinaryString(5)
```

Parameter *length* – the 1D Binary String size

```
append (value)
```

Appends an item to the list

Example:

```
>>> g = G1DBinaryString(2)
>>> g.append(0)
```

Parameter *value* – value to be added, 0 or 1

```
clearList ()
```

Remove all genes from Genome

```
clone ()
```

Return a new instace copy of the genome

Example:

```
>>> g = G1DBinaryString(5)
>>> for i in xrange(len(g)):
...     g.append(1)
>>> clone = g.clone()
>>> clone[0]
1
```

Return type the G1DBinaryString instance clone

copy (*g*)
Copy genome to 'g'

Example:

```
>>> g1 = G1DBinaryString(2)
>>> g1.append(0)
>>> g1.append(1)
>>> g2 = G1DBinaryString(2)
>>> g1.copy(g2)
>>> g2[1]
1
```

Parameter *g* – the destination genome

crossover

This is the reproduction function slot, the crossover. You can change the default crossover method using:

```
genome.crossover.set(Crossovers.G1DBinaryStringXUniform)
```

evaluate (***args*)

Called to evaluate genome

Parameter *args* – this parameters will be passes to the evaluator

evaluator

This is the *evaluation function* slot, you can add a function with the *set* method:

```
genome.evaluator.set(eval_func)
```

getBinary ()

Returns the binary string representation

Example:

```
>>> g = G1DBinaryString(2)
>>> g.append(0)
>>> g.append(1)
>>> g.getBinary()
'01'
```

Return type the binary string

getDecimal ()

Converts the binary string to decimal representation

Example:

```
>>> g = G1DBinaryString(5)
>>> for i in xrange(len(g)):
...     g.append(0)
>>> g[3] = 1
>>> g.getDecimal()
2
```

Return type decimal value

getFitnessScore()

Get the Fitness Score of the genome

Return type genome fitness score

getInternalList()

Returns the internal list of the genome

... note:: this method was created to solve performance issues :rtype: the internal list

getListSize()

Returns the list supposed size

Warning: this is different from what the len(obj) returns

getParam(key, nvl=None)

Gets an internal parameter

Example:

```
>>> genome.getParam("rangemax")
100
```

Note: All the individuals of the population shares this parameters and uses the same instance of this dict.

Parameters

- *key* – the key of param
- *nvl* – if the key doesn't exist, the nvl will be returned

getRawScore()

Get the Raw Score of the genome

Return type genome raw score

initializer

This is the initialization function of the genome, you can change the default initializer using the function slot:

```
genome.initializer.set(Initializers.G1DBinaryStringInitializer)
```

In this example, the initializer `Initializers.G1DBinaryStringInitializer()` will be used to create the initial population.

initialize(args)**

Called to initialize genome

Parameter *args* – this parameters will be passed to the initializer

mutate (**args)

Called to mutate the genome

Parameter *args* – this parameters will be passed to the mutator

Return type the number of mutations returned by mutation operator

mutator

This is the mutator function slot, you can change the default mutator using the slot *set* function:

```
genome.mutator.set (Mutators.G1DBinaryStringMutatorSwap)
```

remove (*value*)

Removes an item from the list

Example:

```
>>> genome.remove (44)
```

Parameter *value* – value to be added

resetStats ()

Clear score and fitness of genome

resumeString ()

Returns a resumed string representation of the Genome

setInternalList (*lst*)

Assigns a list to the internal list of the chromosome

Parameter *lst* – the list to assign the internal list of the chromosome

setParams (**args)

Set the internal params

Example:

```
>>> genome.setParams (rangemin=0, rangemax=100, gauss_mu=0, gauss_sigma=1)
```

Note: All the individuals of the population shares this parameters and uses the same instance of this dict.

Parameter *args* – this params will saved in every chromosome for genetic op. use

G2DBinaryString – the classical binary string chromosome

This representation is a 2D Binary String, the string looks like this matrix:

```
00101101010 00100011010 00101101010 10100101000
```

Default Parameters

Initializer

```
Initializers.G2DBinaryStringInitializer()
```

The Binatry String Initializer for G2DBinaryString

Mutator


```
Mutators.G2DBinaryStringMutatorFlip()
```

The Flip Mutator for G2DBinaryString

Crossover

```
Crossovers.G2DBinaryStringXSinglePoint()
```

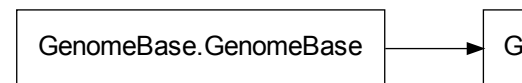
The Single Point Crossover for G2DBinaryString

New in version 0.6: Added the module `G2DBinaryString`

Class

```
class G2DBinaryString(height, width)
```

G3DBinaryString Class - The 2D Binary String chromosome



Inheritance diagram for `G2DBinaryString.G2DBinaryString`:

Example:

```
>>> genome = G2DBinaryString.G2DBinaryString(10, 12)
```

Parameters

- *height* – the number of rows
- *width* – the number of columns

clearString()

Remove all genes from Genome

clone()

Return a new instace copy of the genome

Return type the G2DBinaryString clone instance

copy(g)

Copy genome to 'g'

Example:

```
>>> genome_origin.copy(genome_destination)
```

Parameter *g* – the destination G2DBinaryString instance

crossover

This is the reproduction function slot, the crossover. You can change the default crossover method using:

```
genome.crossover.set(Crossovers.G2DBinaryStringXUniform)
```

evaluate (**args)

Called to evaluate genome

Parameter *args* – this parameters will be passes to the evaluator

evaluator

This is the *evaluation function* slot, you can add a function with the *set* method:

```
genome.evaluator.set(eval_func)
```

getFitnessScore ()

Get the Fitness Score of the genome

Return type genome fitness score

getHeight ()

Return the height (lines) of the List

getItem (x, y)

Return the specified gene of List

Example:

```
>>> genome.getItem(3, 1)
0
```

Parameters

- *x* – the x index, the column
- *y* – the y index, the row

Return type the item at x,y position

getParam (key, nvl=None)

Gets an internal parameter

Example:

```
>>> genome.getParam("rangemax")
100
```

Note: All the individuals of the population shares this parameters and uses the same instance of this dict.

Parameters

- *key* – the key of param
- *nvl* – if the key doesn't exist, the nvl will be returned

getRawScore ()

Get the Raw Score of the genome

Return type genome raw score

getSize ()

Returns a tuple (height, widht)

Example:

```
>>> genome.getSize()
(3, 2)
```

getWidth()

Return the width (lines) of the List

initializer

This is the initialization function of the genome, you can change the default initializer using the function slot:

```
genome.initializer.set(Initializers.G2DBinaryStringInitializer)
```

In this example, the initializer `Initializers.G1DBinaryStringInitializer()` will be used to create the initial population.

initialize (args)**

Called to initialize genome

Parameter *args* – this parameters will be passed to the initializer

mutate (args)**

Called to mutate the genome

Parameter *args* – this parameters will be passed to the mutator

Return type the number of mutations returned by mutation operator

mutator

This is the mutator function slot, you can change the default mutator using the slot *set* function:

```
genome.mutator.set(Mutators.G2DBinaryStringMutatorSwap)
```

resetStats()

Clear score and fitness of genome

resumeString()

Returns a resumed string representation of the Genome

setItem(x, y, value)

Set the specified gene of List

Example:

```
>>> genome.setItem(3, 1, 0)
```

Parameters

- *x* – the x index, the column
- *y* – the y index, the row
- *value* – the value (integers 0 or 1)

setParams (args)**

Set the internal params

Example:

```
>>> genome.setParams(rangemin=0, rangemax=100, gauss_mu=0, gauss_sigma=1)
```

Note: All the individuals of the population shares this parameters and uses the same instance of this dict.

Parameter *args* – this params will saved in every chromosome for genetic op. use

G1DList – the 1D list chromosome

This is the 1D List representation, this list can carry real numbers or integers or any kind of object, by default, we have genetic operators for integer and real lists, which can be found on the respective modules.

Default Parameters

Initializer

```
Initializers.G1DListInitializerInteger()
```

The Integer Initializer for G1DList

Mutator

```
Mutators.G1DListMutatorSwap()
```

The Swap Mutator for G1DList

Crossover

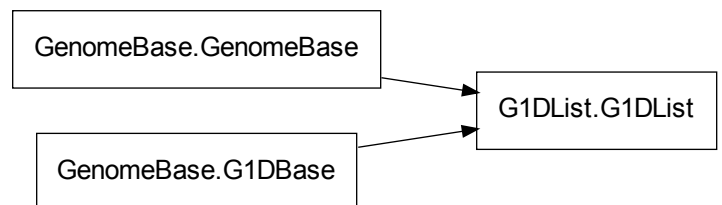
```
Crossovers.G1DListCrossoverSinglePoint()
```

The Single Point Crossover for G1DList

Class

class G1DList (*size=10, cloning=False*)

G1DList Class - The 1D List chromosome representation



Inheritance diagram for `G1DList.G1DList`:

This chromosome class extends the `GenomeBase.GenomeBase` and `GenomeBase.G1DBase` classes.

Examples

The instantiation

```
>>> g = G1DList(10)
```

Compare

```
>>> genome2 = genome1.clone()
>>> genome2 == genome1
True
```

Multiply

```
>>> genome = population[0]
>>> genome
(...)
[1, 2, 3, 4]
>>> genome_result = genome * 2
>>> genome_result
(...)
[2, 2, 6, 8]
```

Add

```
>>> genome
(...)
[1, 2, 3, 4]
>>> genome_result = genome + 2
(...)
[3, 4, 5, 6]
```

Iteration

```
>>> for i in genome:
>>>     print i
1
2
3
4
```

Size, slice, get/set, append

```
>>> len(genome)
4
>>> genome
(...)
[1, 2, 3, 4]
>>> genome[0:1]
[1, 2]
>>> genome[1] = 666
>>> genome
(...)
[1, 666, 3, 4]
>>> genome.append(99)
>>> genome
(...)
[1, 666, 3, 4, 99]
```

Parameter *size* – the 1D list size

append (*value*)

Appends an item to the end of the list

Example:

```
>>> genome.append(44)
```

Parameter *value* – value to be added

clearList ()

Remove all genes from Genome

clone ()

Return a new instace copy of the genome

Return type the G1DList clone instance

copy (*g*)

Copy genome to 'g'

Example:

```
>>> genome_origin.copy(genome_destination)
```

Parameter *g* – the destination G1DList instance

crossover

This is the reproduction function slot, the crossover. You can change the default crossover method using:

```
genome.crossover.set(Crossovers.G1DListCrossoverUniform)
```

evaluate (***args*)

Called to evaluate genome

Parameter *args* – this parameters will be passes to the evaluator

evaluator

This is the *evaluation function* slot, you can add a function with the *set* method:

```
genome.evaluator.set(eval_func)
```

getFitnessScore ()

Get the Fitness Score of the genome

Return type genome fitness score

getInternalList ()

Returns the internal list of the genome

... note:: this method was created to solve performance issues :rtype: the internal list

getListSize ()

Returns the list supposed size

Warning: this is different from what the len(obj) returns
--

getParam (*key*, *nvl=None*)

Gets an internal parameter

Example:

```
>>> genome.getParam("rangemax")
100
```

Note: All the individuals of the population shares this parameters and uses the same instance of this dict.

Parameters

- *key* – the key of param
- *nvl* – if the key doesn't exist, the nvl will be returned

getRawScore ()

Get the Raw Score of the genome

Return type genome raw score

initializer

This is the initialization function of the genome, you can change the default initializer using the function slot:

```
genome.initializer.set(Initializers.G1DListInitializerAllele)
```

In this example, the initializer `Initializers.G1DListInitializerAllele()` will be used to create the initial population.

initialize (args)**

Called to initialize genome

Parameter *args* – this parameters will be passed to the initializer

mutate (args)**

Called to mutate the genome

Parameter *args* – this parameters will be passed to the mutator

Return type the number of mutations returned by mutation operator

mutator

This is the mutator function slot, you can change the default mutator using the slot *set* function:

```
genome.mutator.set(Mutators.G1DListMutatorSwap)
```

remove (value)

Removes an item from the list

Example:

```
>>> genome.remove(44)
```

Parameter *value* – value to be added

resetStats ()

Clear score and fitness of genome

resumeString ()

Returns a resumed string representation of the Genome

setInternalList (lst)

Assigns a list to the internal list of the chromosome

Parameter *lst* – the list to assign the internal list of the chromosome

setParams (**args)
Set the internal params

Example:

```
>>> genome.setParams(rangemin=0, rangemax=100, gauss_mu=0, gauss_sigma=1)
```

Note: All the individuals of the population shares this parameters and uses the same instance of this dict.

Parameter *args* – this params will saved in every chromosome for genetic op. use

G2DList – the 2D list chromosome

This is the 2D List representation, this list can carry real numbers or integers or any kind of object, by default, we have genetic operators for integer and real lists, which can be found on the respective modules. This chromosome class extends the `GenomeBase.GenomeBase`.

Default Parameters

Initializer

```
Initializers.G2DListInitializerInteger()
```

The Integer Initializer for G2DList

Mutator

```
Mutators.G2DListMutatorSwap()
```

The Swap Mutator for G2DList

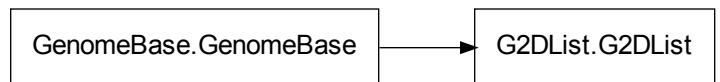
Crossover

```
Crossovers.G2DListCrossoverUniform()
```

The Uniform Crossover for G2DList

Class

class G2DList (*height, width, cloning=False*)
G2DList Class - The 2D List chromosome representation



Inheritance diagram for `G2DList.G2DList`:

Examples

The instantiation

```
>>> genome = G2DList.G2DList(10, 10)
```


Compare

```
>>> genome2 = genome1.clone()
>>> genome2 == genome1
True
```

Iteration

```
>>> for row in genome:
>>>     print row
[1, 3, 4, 1]
[7, 5, 3, 4]
[9, 0, 1, 2]
```

Size, slice, get/set, append

```
>>> len(genome)
3
>>> genome
(...)
[1, 3, 4, 1]
[7, 5, 3, 4]
[9, 0, 1, 2]
>>> genome[1][2]
3
>>> genome[1] = [666, 666, 666, 666]
>>> genome
(...)
[1, 3, 4, 1]
[666, 666, 666, 666]
[9, 0, 1, 2]
>>> genome[1][1] = 2
(...)
```

Parameters

- *height* – the number of rows
- *width* – the number of columns

`clearList()`

Remove all genes from Genome

`clone()`

Return a new instace copy of the genome

Return type the G2DList clone instance

`copy(g)`

Copy genome to 'g'

Example:

```
>>> genome_origin.copy(genome_destination)
```

Parameter *g* – the destination G2DList instance

crossover

This is the reproduction function slot, the crossover. You can change the default crossover method using:

```
genome.crossover.set(Crossovers.G2DListCrossoverSingleHPoint)
```

evaluate (**args)

Called to evaluate genome

Parameter *args* – this parameters will be passes to the evaluator

evaluator

This is the *evaluation function* slot, you can add a function with the *set* method:

```
genome.evaluator.set(eval_func)
```

getFitnessScore ()

Get the Fitness Score of the genome

Return type genome fitness score

getHeight ()

Return the height (lines) of the List

getItem (x, y)

Return the specified gene of List

Example:

```
>>> genome.getItem(3, 1)
666
>>> genome[3][1]
```

Parameters

- *x* – the x index, the column
- *y* – the y index, the row

Return type the item at x,y position

getParam (key, nvl=None)

Gets an internal parameter

Example:

```
>>> genome.getParam("rangemax")
100
```

Note: All the individuals of the population shares this parameters and uses the same instance of this dict.

Parameters

- *key* – the key of param
- *nvl* – if the key doesn't exist, the nvl will be returned

getRawScore ()

Get the Raw Score of the genome

Return type genome raw score

getSize()

Returns a tuple (height, width)

Example:

```
>>> genome.getSize()
(3, 2)
```

getWidth()

Return the width (lines) of the List

initializer

This is the initialization function of the genome, you can change the default initializer using the function slot:

```
genome.initializer.set(Initializers.G2DListInitializerAllele)
```

In this example, the initializer `Initializers.G2DListInitializerAllele()` will be used to create the initial population.

initialize(args)**

Called to initialize genome

Parameter *args* – this parameters will be passed to the initializer

mutate(args)**

Called to mutate the genome

Parameter *args* – this parameters will be passed to the mutator

Return type the number of mutations returned by mutation operator

mutator

This is the mutator function slot, you can change the default mutator using the slot *set* function:

```
genome.mutator.set(Mutators.G2DListMutatorIntegerGaussian)
```

resetStats()

Clear score and fitness of genome

resumeString()

Returns a resumed string representation of the Genome New in version 0.6: The *resumeString* method.

setItem(x, y, value)

Set the specified gene of List

Example:

```
>>> genome.setItem(3, 1, 666)
>>> genome[3][1] = 666
```

Parameters

- *x* – the x index, the column
- *y* – the y index, the row
- *value* – the value

setParams(args)**

Set the internal params

Example:

```
>>> genome.setParams(rangemin=0, rangemax=100, gauss_mu=0, gauss_sigma=1)
```

Note: All the individuals of the population shares this parameters and uses the same instance of this dict.

Parameter *args* – this params will saved in every chromosome for genetic op. use

GTree and GTreeGP – the tree chromosomes

This is the rooted tree representation, this chromosome representation can carry any data-type.

Default Parameters

Initializer

```
Initializers.GTreeInitializerInteger()
```

The Integer Initializer for GTree

Mutator

```
Mutators.GTreeMutatorIntegerRange()
```

The Integer Range mutator for GTree

Crossover

```
Crossovers.GTreeCrossoverSinglePointStrict()
```

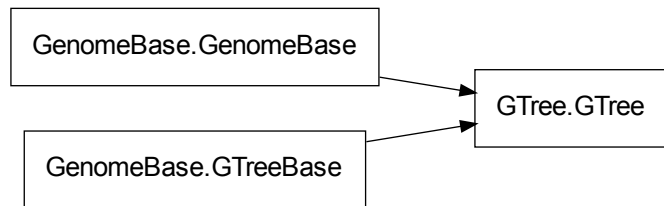
The Strict Single Point crossover for GTree

New in version 0.6: The *GTree* module.

Classes

class GTree (*root_node=None*)

The GTree class - The tree chromosome representation



Inheritance diagram for *GTree.GTree*:

Parameter *root_node* – the root node of the tree

clone ()

Return a new instance of the genome

Return type new GTree instance

copy (*g*)

Copy the contents to the destination *g*

Parameter *g* – the GTree genome destination

crossover

This is the reproduction function slot, the crossover. You can change the default crossover method using:

```
genome.crossover.set(Crossovers.G1DListCrossoverUniform)
```

evaluate (***args*)

Called to evaluate genome

Parameter *args* – this parameters will be passes to the evaluator

evaluator

This is the *evaluation function* slot, you can add a function with the *set* method:

```
genome.evaluator.set(eval_func)
```

getAllNodes ()

Return a new list with all nodes

Return type the list with all nodes

getFitnessScore ()

Get the Fitness Score of the genome

Return type genome fitness score

getHeight ()

Return the tree height

Return type the tree height

getNodeDepth (*node*)

Returns the depth of a node

Return type the depth of the node, the depth of root node is 0

getNodeHeight (*node*)

Returns the height of a node

Note: If the node has no childs, the height will be 0.

Return type the height of the node

getNodeCount (*start_node=None*)

Return the number of the nodes on the tree starting at the *start_node*, if *start_node* is None, then the method will count all the tree nodes.

Return type the number of nodes

getParam (*key, nvl=None*)

Gets an internal parameter

Example:

```
>>> genome.getParam("rangemax")
100
```

Note: All the individuals of the population shares this parameters and uses the same instance of this dict.

Parameters

- *key* – the key of param
- *nvl* – if the key doesn't exist, the nvl will be returned

getRandomNode (*node_type=0*)

Returns a random node from the Tree

Parameter *node_type* – 0 = Any, 1 = Leaf, 2 = Branch

Return type random node

getRawScore ()

Get the Raw Score of the genome

Return type genome raw score

getRoot ()

Return the tree root node

Return type the tree root node

getTraversalString (*start_node=None, spc=0*)

Returns a tree-formated string of the tree. This method is used by the `__repr__` method of the tree

Return type a string representing the tree

initializer

This is the initialization function of the genome, you can change the default initializer using the function slot:

```
genome.initializer.set(Initializers.G1DListInitializerAllele)
```

In this example, the initializer `Initializers.G1DListInitializerAllele()` will be used to create the initial population.

initialize (**args)

Called to initialize genome

Parameter *args* – this parameters will be passed to the initializer

mutate (**args)

Called to mutate the genome

Parameter *args* – this parameters will be passed to the mutator

Return type the number of mutations returned by mutation operator

mutator

This is the mutator function slot, you can change the default mutator using the slot `set` function:

```
genome.mutator.set(Mutators.G1DListMutatorSwap)
```

processNodes (*cloning=False*)

Creates a *cache* on the tree, this method must be called every time you change the shape of the tree. It updates the internal nodes list and the internal nodes properties such as depth and height.

resetStats ()

Clear score and fitness of genome

setParams (**args)

Set the internal params

Example:

```
>>> genome.setParams(rangemin=0, rangemax=100, gauss_mu=0, gauss_sigma=1)
```

Note: All the individuals of the population shares this parameters and uses the same instance of this dict.

Parameter *args* – this params will saved in every chromosome for genetic op. use

setRoot (*root*)

Sets the root of the tree

Parameter *root* – the tree root node

traversal (*callback, start_node=None*)

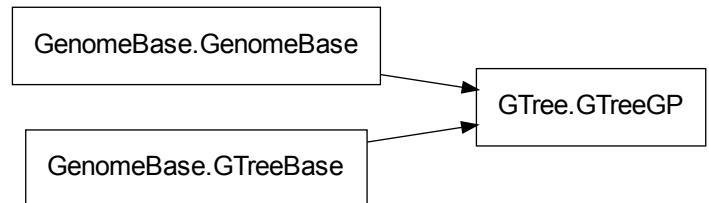
Traversal the tree, this method will call the user-defined callback function for each node on the tree

Parameters

- *callback* – a function
- *start_node* – the start node to begin the traversal

class GTreeGP (*root_node=None, cloning=False*)

The GTreeGP Class - The Genetic Programming Tree representation



Inheritance diagram for `GTree.GTreeGP`:

Parameter *root_node* – the Root node of the GP Tree

clone ()

Return a new instance of the genome

Return type the new GTreeGP instance

compare (*other*)

This method will compare the currently tree with another one

Parameter *other* – the other GTreeGP to compare

copy (*g*)

Copy the contents to the destination g

Parameter *g* – the GTreeGP genome destination

evaluate (***args*)

Called to evaluate genome

Parameter *args* – this parameters will be passes to the evaluator

getAllNodes ()

Return a new list with all nodes

Return type the list with all nodes

getCompiledCode ()

Get the compiled code for the Tree expression After getting the compiled code object, you just need to evaluate it using the `eval ()` native Python method.

Return type compiled python code

getFitnessScore ()

Get the Fitness Score of the genome

Return type genome fitness score

getHeight ()

Return the tree height

Return type the tree height

getNodeDepth (node)

Returns the depth of a node

Return type the depth of the node, the depth of root node is 0

getNodeHeight (node)

Returns the height of a node

Note: If the node has no childs, the height will be 0.

Return type the height of the node

getNodesCount (start_node=None)

Return the number of the nodes on the tree starting at the *start_node*, if *start_node* is None, then the method will count all the tree nodes.

Return type the number of nodes

getParam (key, nvl=None)

Gets an internal parameter

Example:

```
>>> genome.getParam("rangemax")
100
```

Note: All the individuals of the population shares this parameters and uses the same instance of this dict.

Parameters

- *key* – the key of param
- *nvl* – if the key doesn't exist, the nvl will be returned

getPreOrderExpression (start_node=None)

Return the pre order expression string of the Tree, used to python *eval*.

Return type the expression string

getRandomNode (node_type=0)

Returns a random node from the Tree

Parameter *node_type* – 0 = Any, 1 = Leaf, 2 = Branch

Return type random node

getRawScore ()

Get the Raw Score of the genome

Return type genome raw score

getRoot ()

Return the tree root node

Return type the tree root node

getSEExpression (*start_node=None*)

Returns a tree-formated string (s-expression) of the tree.

Return type a S-Expression representing the tree

getTraversalString (*start_node=None, spc=0*)

Returns a tree-formated string of the tree. This method is used by the `__repr__` method of the tree

Return type a string representing the tree

initialize (***args*)

Called to initialize genome

Parameter *args* – this parameters will be passed to the initializer

mutate (***args*)

Called to mutate the genome

Parameter *args* – this parameters will be passed to the mutator

Return type the number of mutations returned by mutation operator

processNodes (*cloning=False*)

Creates a *cache* on the tree, this method must be called every time you change the shape of the tree. It updates the internal nodes list and the internal nodes properties such as depth and height.

resetStats ()

Clear score and fitness of genome

setParams (***args*)

Set the internal params

Example:

```
>>> genome.setParams(rangemin=0, rangemax=100, gauss_mu=0, gauss_sigma=1)
```

Note: All the individuals of the population shares this parameters and uses the same instance of this dict.

Parameter *args* – this params will saved in every chromosome for genetic op. use

setRoot (*root*)

Sets the root of the tree

Parameter *root* – the tree root node

traversal (*callback, start_node=None*)

Traversal the tree, this method will call the user-defined callback function for each node on the tree

Parameters

- *callback* – a function
- *start_node* – the start node to begin the traversal

writeDotGraph (*graph, startNode=0*)

Write a graph to the pydot Graph instance

Parameters

- *graph* – the pydot Graph instance

- *startNode* – used to plot more than one individual

writeDotImage (*filename*)

Writes a image representation of the individual

Parameter *filename* – the output file image

writeDotRaw (*filename*)

Writes the raw dot file (text-file used by dot/neato) with the representation of the individual

Parameter *filename* – the output file, ex: individual.dot

static **writePopulationDot** (*ga_engine, filename, format='jpeg', start=0, end=0*)

Writes to a graphical file using pydot, the population of trees

Example:

```
>>> GTreeGP.writePopulationDot(ga_engine, "pop.jpg", "jpeg", 0, 10)
```

This example will draw the first ten individuals of the population into the file called “pop.jpg”.

Parameters

- *ga_engine* – the GA Engine
- *filename* – the filename, ie. population.jpg
- *start* – the start index of individuals
- *end* – the end index of individuals

static **writePopulationDotRaw** (*ga_engine, filename, start=0, end=0*)

Writes to a raw dot file using pydot, the population of trees

Example:

```
>>> GTreeGP.writePopulationDotRaw(ga_engine, "pop.dot", 0, 10)
```

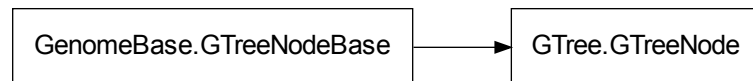
This example will draw the first ten individuals of the population into the file called “pop.dot”.

Parameters

- *ga_engine* – the GA Engine
- *filename* – the filename, ie. population.dot
- *start* – the start index of individuals
- *end* – the end index of individuals

class GTreeNode (*data, parent=None*)

The GTreeNode class - The node representation



Inheritance diagram for `GTree.GTreeNode`:

Parameters

- *data* – the root node of the tree
- *parent* – the parent node, if root, this must be *None*

addChild (*child*)

Adds a child to the node

Parameter *child* – the node to be added

clone ()

Return a new instance of the genome

Return type new GTree instance

copy (*g*)

Copy the contents to the destination *g*

Parameter *g* – the GTreeNode genome destination

getChild (*index*)

Returns the index-child of the node

Return type child node

getChildren ()

Return the children of the node

Warning: use `.getChildren()[:]` if you'll change the list itself, like using `children.reverse()`, otherwise the original genome child order will be changed.

Return type a list of nodes

getData ()

Return the data of the node

Return type the data of the node

getParent ()

Get the parent node of the node

Return type the parent node

isLeaf ()

Return True if the node is a leaf

Return type True or False

newNode (*data*)

Created a new child node

Parameter *data* – the data of the new created node

replaceChild (*older*, *newer*)

Replaces a child of the node

Parameters

- *older* – the child to be replaced
- *newer* – the new child which replaces the older

setData (*data*)

Sets the data of the node

Parameter *data* – the data of the node

setParent (*parent*)

Sets the parent of the node

Parameter *parent* – the parent node

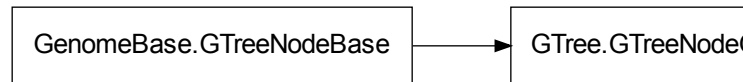
swapNodeData (*node*)

Swaps the node data with another node

Parameter *node* – the node to do the data swap

class GTreeNodeGP (*data, node_type=0, parent=None*)

The GTreeNodeGP Class - The Genetic Programming Node representation



Inheritance diagram for `GTree.GTreeNodeGP`:

Parameters

- *data* – the node data
- *type* – the node type
- *parent* – the node parent

addChild (*child*)

Adds a child to the node

Parameter *child* – the node to be added

clone ()

Return a new copy of the node

Return type the new GTreeNodeGP instance

compare (*other*)

Compare this node with other

Parameter *other* – the other GTreeNodeGP

copy (*g*)

Copy the contents to the destination g

Parameter *g* – the GTreeNodeGP genome destination

getChild (*index*)

Returns the index-child of the node

Return type child node

getChilds ()

Return the childs of the node

Warning: use `.getChilds()[:]` if you'll change the list itself, like using `childs.reverse()`, otherwise the original genome child order will be changed.

Return type a list of nodes

getData ()

Gets the node internal data

Return type the internal data

getParent ()

Get the parent node of the node

Return type the parent node

getType ()

Get the node type

Return type the node type is type of `Consts.nodeType`

isLeaf ()

Return True if the node is a leaf

Return type True or False

newNode (data)

Creates a new node and adds this node as children of current node

Parameter *data* – the internal node data

replaceChild (older, newer)

Replaces a child of the node

Parameters

- *older* – the child to be replaces
- *newer* – the new child which replaces the older

setData (data)

Sets the node internal data

Parameter *data* – the internal data

setParent (parent)

Sets the parent of the node

Parameter *parent* – the parent node

setType (node_type)

Sets the node type

Parameter *node_type* – the node type is type of `Consts.nodeType`

swapNodeData (node)

Swaps the node data and type with another node

Parameter *node* – the node

buildTreeFull (depth, value_callback, max_siblings, max_depth)

Random generates a Tree structure using the *value_callback* for data generation and the method “Full”

Parameters

- *depth* – the initial depth, zero
- *value_callback* – the function which generates the random values for nodes
- *max_siblings* – the maximum number of sisters of a node
- *max_depth* – the maximum depth of the tree

Return type the root node of created tree

buildGTreeGPFull (*ga_engine*, *depth*, *max_depth*)

Creates a new random GTreeGP root node with subtrees using the “Full” method.

Parameters

- *ga_engine* – the GA Core
- *depth* – the initial depth

Max_depth the maximum depth of the tree

Return type the root node

buildGTreeGPGrow (*ga_engine*, *depth*, *max_depth*)

Creates a new random GTreeGP root node with subtrees using the “Grow” method.

Parameters

- *ga_engine* – the GA Core
- *depth* – the initial depth

Max_depth the maximum depth of the tree

Return type the root node

buildGTreeGrow (*depth*, *value_callback*, *max_siblings*, *max_depth*)

Random generates a Tree structure using the *value_callback* for data generation and the method “Grow”

Parameters

- *depth* – the initial depth, zero
- *value_callback* – the function which generates the random values for nodes
- *max_siblings* – the maximum number of sisters of a node
- *max_depth* – the maximum depth of the tree

Return type the root node of created tree

checkTerminal (*terminal*)

Do some check on the terminal, to evaluate ephemeral constants

Parameter *terminal* – the terminal string

gpdec (***kws*)

This is a decorator to use with genetic programming non-terminals

It currently accepts the attributes: shape, color and representation.

2.6 Graphical Analysis - Plots

Pyevolve comes with an Graphical Plotting Tool, this utility uses the great python plotting library called Matplotlib.

See Also:

[Requirements](#) section.

You can find the Pyevolve plotting tool in your python Scripts directory, the tool is named **pyevolve_graph.py**.

2.6.1 Graphical Plotting Tool Options

`pyevolve_graph.py`, installed in `\Python2x\Scripts\pyevolve_graph.py`.

This is the documentation where you call the `-help` option:

```
Pyevolve 0.6rc1 - Graph Plot Tool
By Christian S. Perone
```

```
Usage: pyevolve_graph.py [options]
```

Options:

```
-h, --help                show this help message and exit
-f FILENAME, --file=FILENAME
                          Database file to read (default is 'pyevolve.db').
-i IDENTIFY, --identify=IDENTIFY
                          The identify of evolution.
-o OUTFILE, --outfile=OUTFILE
                          Write the graph image to a file (don't use extension,
                          just the filename, default is png format, but you can
                          change using --extension (-e) parameter).
-e EXTENSION, --extension=EXTENSION
                          Graph image file format. Supported options (formats)
                          are: emf, eps, pdf, png, ps, raw, rgba, svg, svgz.
                          Default is 'png'.
-g GENRANGE, --genrange=GENRANGE
                          This is the generation range of the graph, ex: 1:30
                          (interval between 1 and 30).
-l LINDRANGE, --lindrang=LINDRANGE
                          This is the individual range of the graph, ex: 1:30
                          (individuals between 1 and 30), only applies to
                          heatmaps.
-c COLORMAP, --colormap=COLORMAP
                          Sets the Color Map for the graph types 8 and 9. Some
                          options are: summer, bone, gray, hot, jet, cooper,
                          spectral. The default is 'jet'.
-m, --minimize            Sets the 'Minimize' mode, default is the Maximize
                          mode. This option makes sense if you are minimizing
                          your evaluation function.
```

Graph types:

This is the supported graph types

```
-0                        Write all graphs to files. Graph types: 1, 2, 3, 4 and
                          5.
-1                        Error bars graph (raw scores).
-2                        Error bars graph (fitness scores).
-3                        Max/min/avg/std. dev. graph (raw scores).
-4                        Max/min/avg graph (fitness scores).
-5                        Raw and Fitness min/max difference graph.
-6                        Compare best raw score of two or more evolutions (you
                          must specify the identify comma-separated list with
                          --identify (-i) parameter, like 'one, two, three'),
                          the maximum is 6 items.
-7                        Compare best fitness score of two or more evolutions
                          (you must specify the identify comma-separated list with
                          --identify (-i) parameter, like 'one, two, three'),
                          the maximum is 6 items.
```

- 8 Show a heat map of population raw score distribution between generations.
- 9 Show a heat map of population fitness score distribution between generations.

2.6.2 Usage

To use this graphical plotting tool, you need to use the `DBAdapters.DBSQLite` adapter and create the database file. Pyevolve have the “identify” concept, the value of this parameter means the same value used in the “identify” parameter of the DB Adapter.

See this example:

```
sqlite_adapter = DBAdapters.DBSQLite(identify="ex1")
ga.setDBAdapter(sqlite_adapter)
```

This DB Adapter attached to the GA Engine will create the database file named “pyevolve.db”.

See Also:

Sqliteman, a tool for sqlite3 databases I recommend the Sqliteman tool to open the database and see the contents or structure, if you are interested.

When you run your GA, all the statistics will be dumped to this database, and you have an ID for this run, which is the identify parameter. So when you use the graph tool, it will read the statistics from this database file. The “identify” parameter is passed to the tool using the “-i” option, like this:

```
pyevolve_graph.py -i ex1 -l
```

By default, this tool will use the database file named *pyevolve.db*, but you can change using the “-f” option like this:

```
pyevolve_graph.py -i ex1 -l -f another_db.db
```

2.6.3 Usage Examples

Writing graph to a file

PDF File:

```
pyevolve_graph.py -i ex1 -l -o graph_ex1 -e pdf
```

PNG File (default extension when using “-o” option):

```
pyevolve_graph.py -i ex1 -l -o graph_ex1
```

Using the generation range

```
# this command wil plot the evolution of the generations between 10 and 20.
pyevolve_graph.py -i ex1 -l -g 10:20
```

When you have minimized the evaluation function

```
pyevolve_graph.py -i ex1 -l -m
```


To specify an identify list (graphs “-6” and “-7”)

```
pyevolve_graph.py -i ex1_run1,ex1_run2,ex1_run3 -6s
```

2.6.4 Graph Types and Screenshots

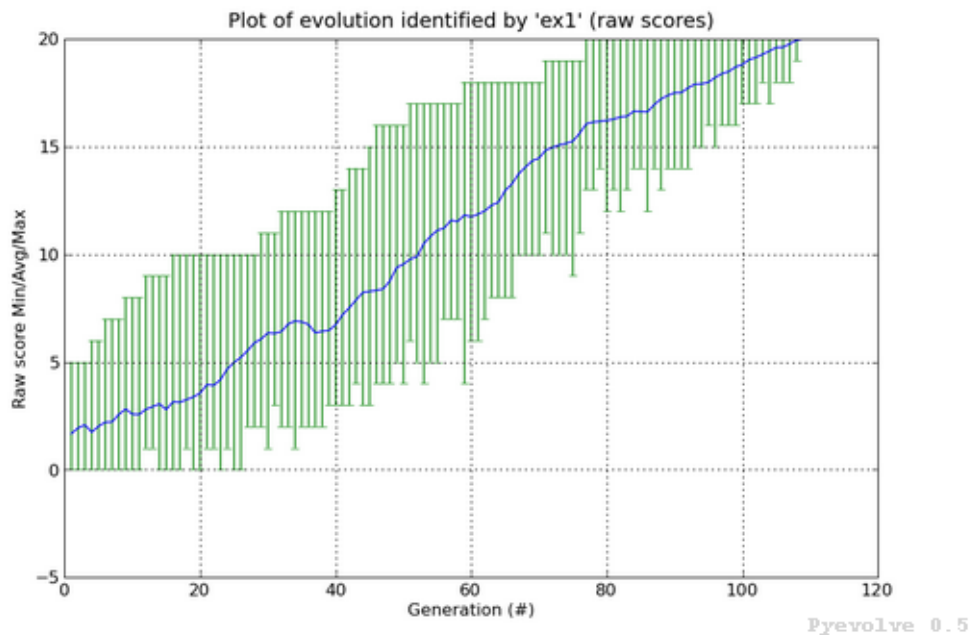
Here are described all the graph types and some screenshots.

Error bars graph (raw scores) / “-1” option

In this graph, you will find the generations on the x-axis and the raw scores on the y-axis. The green vertical bars represents the **maximum and the minimum raw scores** of the current population at generation indicated in the x-axis. The blue line between them is the **average raw score** of the population.

This graph was generated using:

```
pyevolve_graph.py -i ex1 -1
```

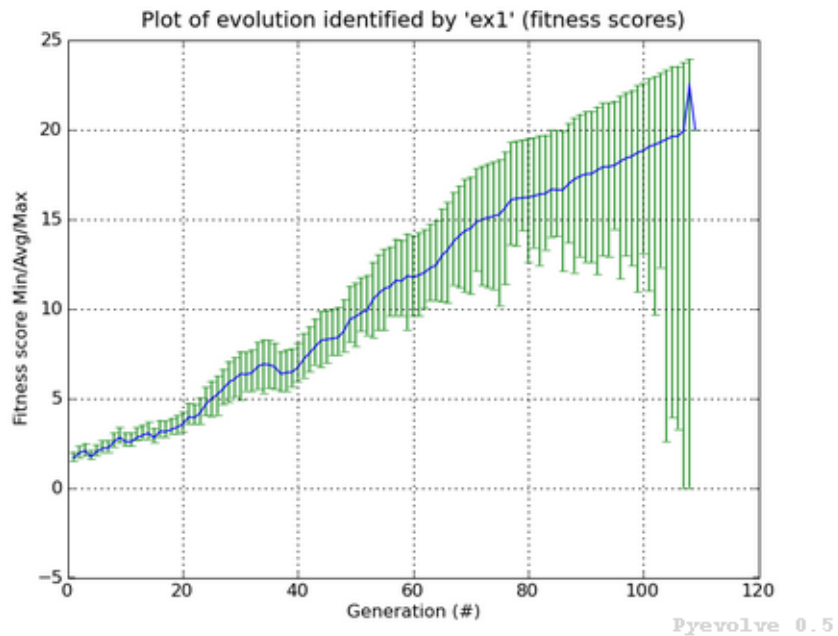


Error bars graph (fitness scores) / “-2” option

The difference between this graph option and the “-1” option is that we are using the **fitness scores** instead of the raw scores.

This graph was generated using:

```
pyevolve_graph.py -i ex1 -2
```



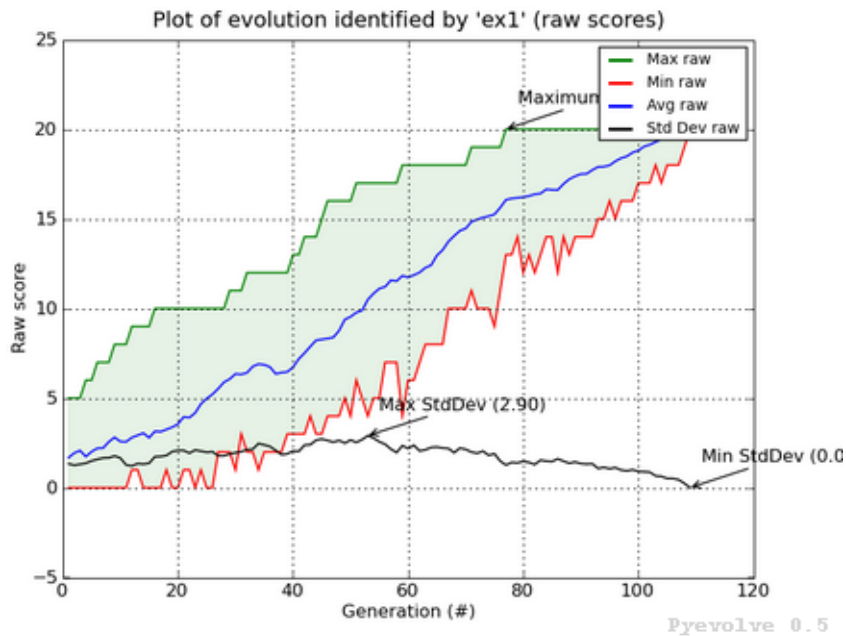
Note: This graph is from a GA using the **Linear Scaling** scheme and the **Roulette Wheel** selection method.

Max/min/avg/std. dev. graph (raw scores) / “-3” option

In this graph we have the green line showing the maximum raw score at the generation in the x-axis, the red line shows the minimum raw score, and the blue line shows the average raw scores. The green shaded region represents the difference between our max. and min. raw scores. The black line shows the standard deviation of the average raw scores. We also have some annotations like the maximum raw score, maximum std. dev. and the min std. dev.

This graph was generated using:

```
pyevolve_graph.py -i ex1 -3
```



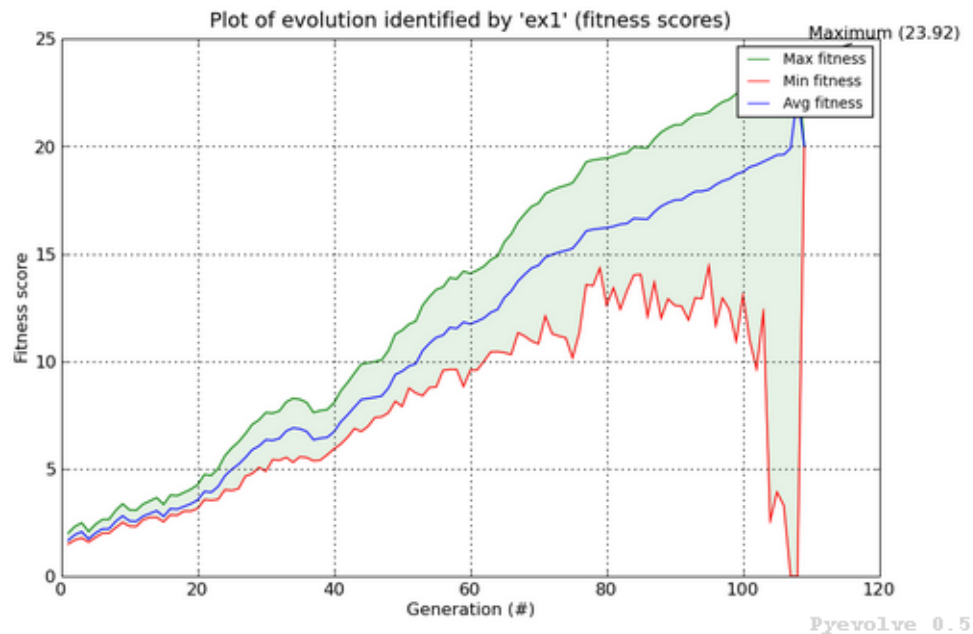
Note: We can see in this graph the minimum standard deviation at the convergence point. The GA Engine have stopped the evolution using this criteria.

Max/min/avg graph (fitness scores) / “-4” option

This graphs shows the maximum fitness score from the population at the x-axis generation using the green line. The red line shows the minimum fitness score and the blue line shows the average fitness score from the population. The green shaded region between the green and red line shows the difference between the best and worst individual of population.

This graph was generated using:

```
pyevolve_graph.py -i ex1 -4
```



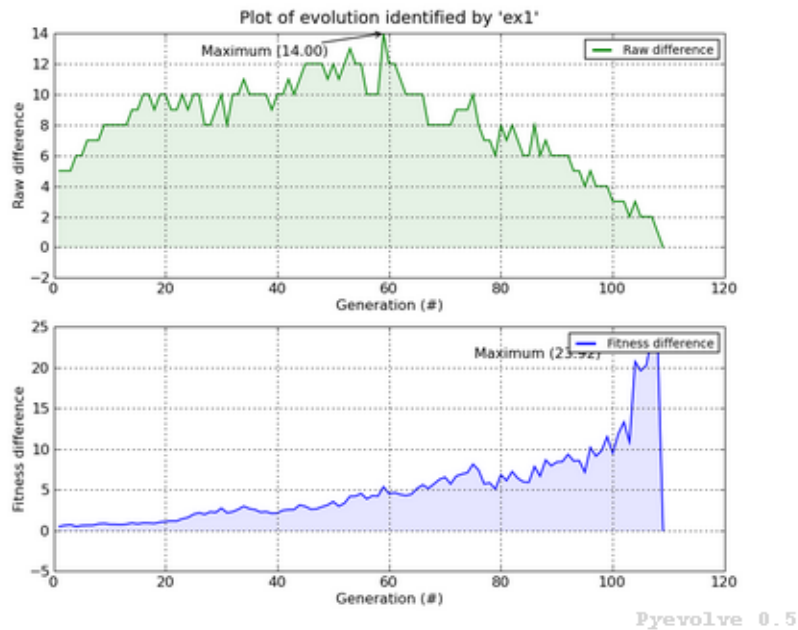
Note: This graph is from a GA using the **Linear Scaling** scheme and the **Roulette Wheel** selection method.

Min/max difference graph, raw and fitness scores / “-5” option

In this graph, we have two subplots, the first is the difference between the best individual raw score and the worst individual raw score. The second graph shows the difference between the best individual fitness score and the worst individual fitness score. Both subplots show the generation on the x-axis and the score difference in the y-axis.

This graph was generated using:

```
pyevolve_graph.py -i ex1 -5
```



Compare best raw score of two or more evolutions / “-6” option

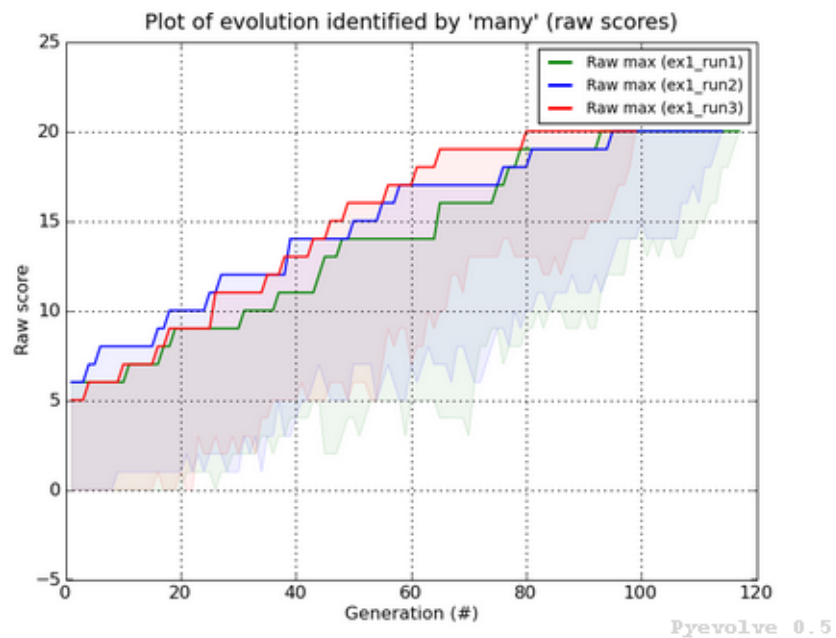
This graph is used to compare two or more evolutions (the max is 6 evolutions) of the same or different GA.

It uses a different color for each identify you use, in the example, you can see the three evolutions (green, blue and red lines) of the same GA.

All the lines have a shaded transparent region of the same line color, they represent the difference between the maximum and the minimum raw scores of the evolution.

This graph was generated using:

```
pyevolve_graph.py -i ex1_run1,ex1_run2,ex1_run3 -6
```



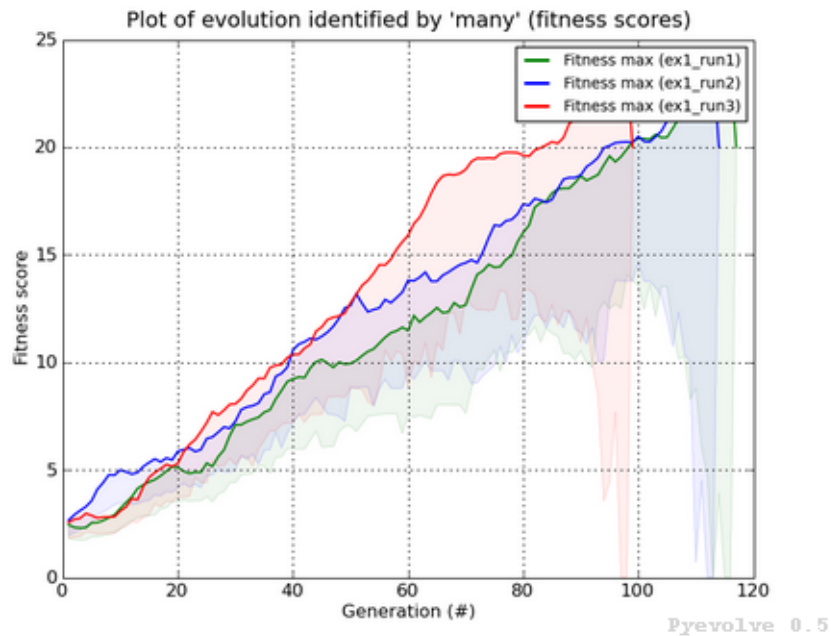
Note: The evolution identified by “ex1_run3” (red color) is the evolution in which the best raw score (20) was got first at the generation 80, compared to the other runs.

Compare best fitness score of two or more evolutions / “-7” option

The difference between this graph option and the “-6” option is that we are using the **fitness scores** instead of the raw scores.

This graph was generated using:

```
pyevolve_graph.py -i ex1_run1,ex1_run2,ex1_run3 -7
```

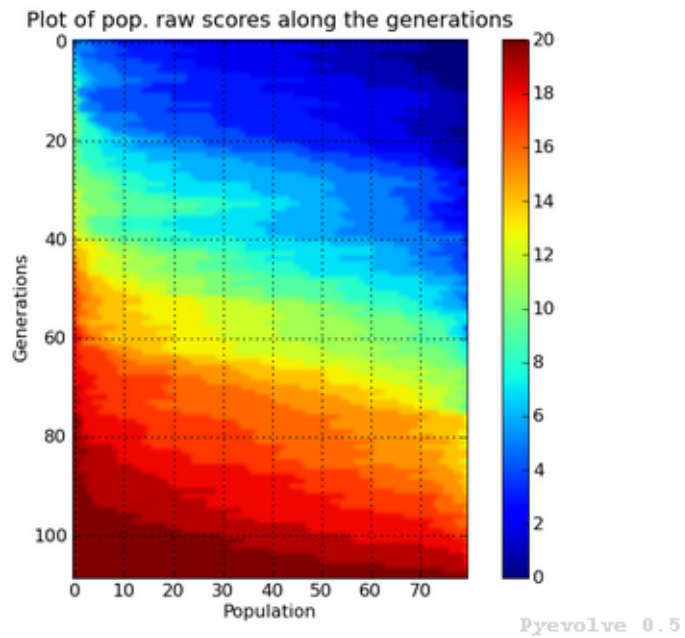


Heat map of population raw score distribution / “-8” option

The heat map graph is a plot with the population individual plotted as the x-axis and the generation plotted in the y-axis. On the right side we have a legend with the color/score relation. As you can see, on the initial populations, the last individuals scores are the worst (represented in this colormap with the dark blue). To create this graph, we use the Gaussian interpolation method.

This graph was generated using:

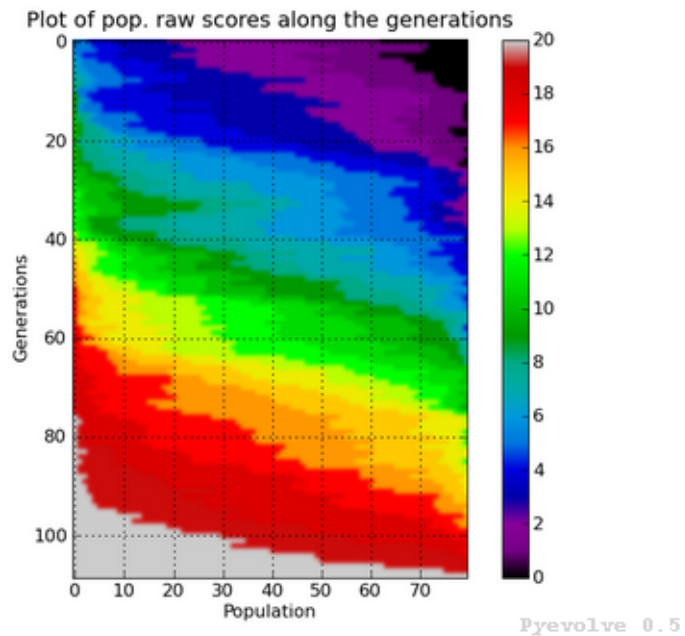
```
pyevolve_graph.py -i ex1 -8
```



Using another colormap like the “spectral”, we can see more interesting patterns:

This graph was generated using:

```
pyevolve_graph.py -i ex1 -8 -c spectral
```



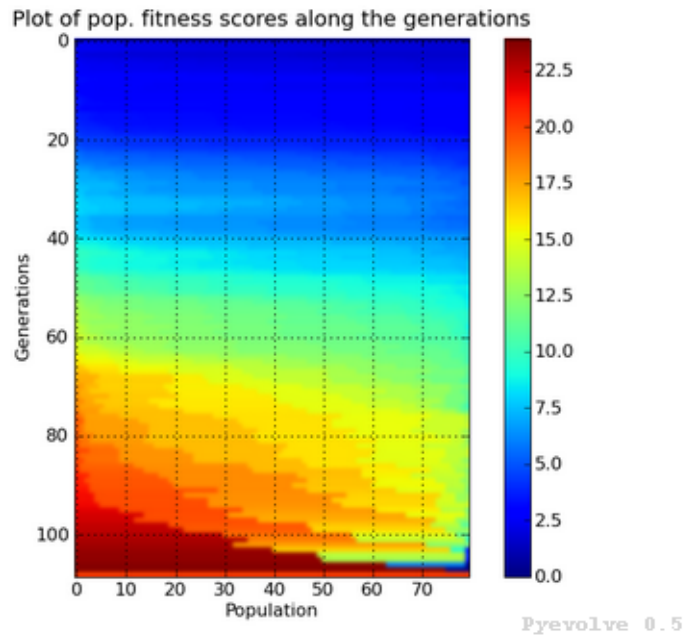
Warning: This graph generation can be very slow if you have too many generations. You can use the “-g” option to limit your generations.

Heat map of population fitness score distribution / “-9” option

The difference between this graph option and the “-8” option is that we are using the **fitness scores** instead of the raw scores.

This graph was generated using:

```
pyevolve_graph.py -i ex1 -9
```

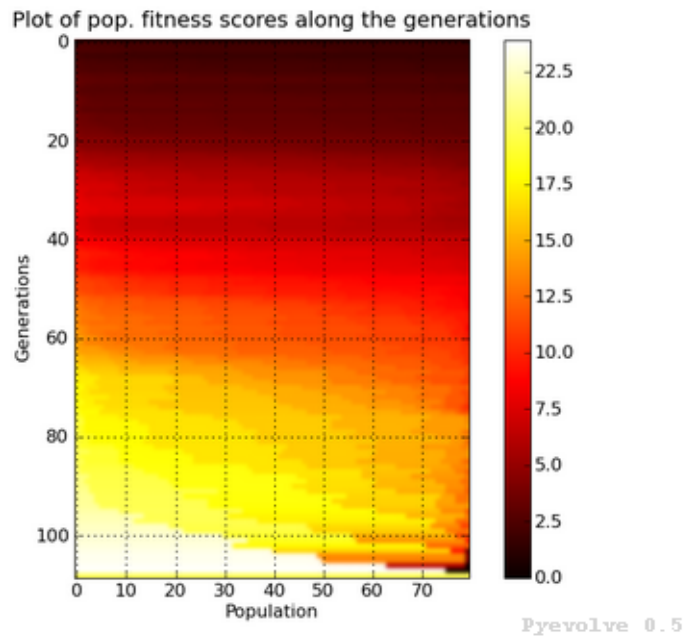


Note: Here you can note some interesting thing, in this graph of the scaled score, the individuals fitness seems almost equally distributed in the population.

Now, the same plot using the “hot” colormap.

This graph was generated using:

```
pyevolve_graph.py -i ex1 -9 -c hot
```



Warning: This graph generation can be very slow if you have too many generations. You can use the “-g” option to limit your generations.

2.7 Examples

All the examples can be download from the [Downloads](#) section, **they are not** included in the installation package.

2.7.1 Example 1 - Simple example

Filename: `examples/pyevolve_ex1_simple.py`

This is the Example #1, it is a very simple example:

```
from pyevolve import G1DList
from pyevolve import GSimpleGA
from pyevolve import Selectors
from pyevolve import Statistics
from pyevolve import DBAdapters

# This function is the evaluation function, we want
# to give high score to more zero'ed chromosomes
def eval_func(genome):
    score = 0.0

    # iterate over the chromosome
    # The same as "score = len(filter(lambda x: x==0, genome))"
    for value in genome:
        if value==0:
            score += 1
```

```

    return score

def run_main():
    # Genome instance, 1D List of 50 elements
    genome = G1DList.G1DList(50)

    # Sets the range max and min of the 1D List
    genome.setParams(rangemin=0, rangemax=10)

    # The evaluator function (evaluation function)
    genome.evaluator.set(eval_func)

    # Genetic Algorithm Instance
    ga = GSimpleGA.GSimpleGA(genome)

    # Set the Roulette Wheel selector method, the number of generations and
    # the termination criteria
    ga.selector.set>Selectors.GRouletteWheel)
    ga.setGenerations(500)
    ga.terminationCriteria.set(GSimpleGA.ConvergenceCriteria)

    # Sets the DB Adapter, the resetDB flag will make the Adapter recreate
    # the database and erase all data every run, you should use this flag
    # just in the first time, after the pyevolve.db was created, you can
    # omit it.
    sqlite_adapter = DBAdapters.DBSQLite(identify="ex1", resetDB=True)
    ga.setDBAdapter(sqlite_adapter)

    # Do the evolution, with stats dump
    # frequency of 20 generations
    ga.evolve(freq_stats=20)

    # Best individual
    print ga.bestIndividual()

if __name__ == "__main__":
    run_main()

```

2.7.2 Example 2 - Real numbers, Gaussian Mutator

Filename: examples/pyevolve_ex2_realgauss.py

This example uses the `Initializers.G1DListInitializerReal()` initializer and the `Mutators.G1DListMutatorRealGaussian()` mutator:

```

from pyevolve import GSimpleGA
from pyevolve import G1DList
from pyevolve import Selectors
from pyevolve import Initializers, Mutators

# Find negative element
def eval_func(genome):
    score = 0.0

    for element in genome:
        if element < 0: score += 0.1

```

```
    return score

def run_main():
    # Genome instance
    genome = G1DList.G1DList(20)
    genome.setParams(rangemin=-6.0, rangemax=6.0)

    # Change the initializer to Real values
    genome.initializer.set(Initializers.G1DListInitializerReal)

    # Change the mutator to Gaussian Mutator
    genome.mutator.set(Mutators.G1DListMutatorRealGaussian)

    # The evaluator function (objective function)
    genome.evaluator.set(eval_func)

    # Genetic Algorithm Instance
    ga = GSimpleGA.GSimpleGA(genome)
    ga.selector.set(Selectors.GRouletteWheel)
    ga.setGenerations(100)

    # Do the evolution
    ga.evolve(freq_stats=10)

    # Best individual
    print ga.bestIndividual()

if __name__ == "__main__":
    run_main()
```

2.7.3 Example 3 - Schaffer F6 deceptive function

Filename: examples/pyevolve_ex3_schaffer.py

This examples tries to minimize the Schaffer F6 function, this function is a deceptive function, considered a GA-hard function to optimize:

```
from pyevolve import G1DList, GSimpleGA, Selectors
from pyevolve import Initializers, Mutators, Consts
import math

# This is the Schaffer F6 Function
# This function has been conceived by Schaffer, it's a
# multimodal function and it's hard for GAs due to the
# large number of local minima, the global minimum is
# at x=0,y=0 and there are many local minima around it
def schafferF6(genome):
    t1 = math.sin(math.sqrt(genome[0]**2 + genome[1]**2));
    t2 = 1.0 + 0.001*(genome[0]**2 + genome[1]**2);
    score = 0.5 + (t1*t1 - 0.5)/(t2*t2)
    return score

def run_main():
    # Genome instance
    genome = G1DList.G1DList(2)
    genome.setParams(rangemin=-100.0, rangemax=100.0, bestrawscore=0.0000, rounddecimal=4)
```

```

genome.initializator.set(Initializators.G1DListInitializatorReal)
genome.mutator.set(Mutators.G1DListMutatorRealGaussian)

# The evaluator function (objective function)
genome.evaluator.set(schafferF6)

# Genetic Algorithm Instance
ga = GSimpleGA.GSimpleGA(genome)
ga.selector.set(Selectors.GRouletteWheel)

ga.setMinimax(Consts.minimaxType["minimize"])
ga.setGenerations(8000)
ga.setMutationRate(0.05)
ga.setPopulationSize(100)
ga.terminationCriteria.set(GSimpleGA.RawScoreCriteria)

# Do the evolution, with stats dump
# frequency of 10 generations
ga.evolve(freq_stats=250)

# Best individual
best = ga.bestIndividual()
print best
print "Best individual score: %.2f" % best.getRawScore()

if __name__ == "__main__":
    run_main()

```

2.7.4 Example 4 - Using Sigma truncation scaling

Filename: examples/pyevolve_ex4_sigmatrunc.py

This example shows the use of the sigma truncation scale method, it tries to minimize a function with negative results:

```

from pyevolve import G1DList
from pyevolve import GSimpleGA
from pyevolve import Selectors
from pyevolve import Initializators, Mutators
from pyevolve import Scaling
from pyevolve import Consts
import math

def eval_func(ind):
    score = 0.0
    var_x = ind[0]
    var_z = var_x**2+2*var_x+1*math.cos(var_x)
    return var_z

def run_main():
    # Genome instance
    genome = G1DList.G1DList(1)
    genome.setParams(rangemin=-60.0, rangemax=60.0)

    # Change the initializer to Real values
    genome.initializator.set(Initializators.G1DListInitializatorReal)

```

```
# Change the mutator to Gaussian Mutator
genome.mutator.set (Mutators.G1DListMutatorRealGaussian)

# Removes the default crossover
genome.crossover.clear()

# The evaluator function (objective function)
genome.evaluator.set (eval_func)

# Genetic Algorithm Instance
ga = GSimpleGA.GSimpleGA (genome)
ga.setMinimax (Consts.minimaxType["minimize"])

pop = ga.getPopulation()
pop.scaleMethod.set (Scaling.SigmaTruncScaling)

ga.selector.set (Selectors.GRouletteWheel)
ga.setGenerations (100)

# Do the evolution
ga.evolve (10)

# Best individual
print ga.bestIndividual()

if __name__ == "__main__":
    run_main()
```

2.7.5 Example 5 - Step callback function

Filename: examples/pyevolve_ex5_callback.py

This example shows the use of the *step callback function*:

```
from pyevolve import G1DList
from pyevolve import GSimpleGA
from pyevolve import Selectors

# The step callback function, this function
# will be called every step (generation) of the GA evolution
def evolve_callback(ga_engine):
    generation = ga_engine.getCurrentGeneration()
    if generation % 100 == 0:
        print "Current generation: %d" % (generation,)
        print ga_engine.getStatistics()
    return False

# This function is the evaluation function, we want
# to give high score to more zero'ed chromosomes
def eval_func(genome):
    score = 0.0
    # iterate over the chromosome
    for value in genome:
        if value==0: score += 0.1
    return score
```

```

def run_main():
    # Genome instance
    genome = G1DList.G1DList(200)
    genome.setParams(rangemin=0, rangemax=10)

    # The evaluator function (objective function)
    genome.evaluator.set(eval_func)

    # Genetic Algorithm Instance
    ga = GSimpleGA.GSimpleGA(genome)
    ga.selector.set(Selectors.GRouletteWheel)
    ga.setGenerations(800)
    ga.stepCallback.set(evolve_callback)

    # Do the evolution
    ga.evolve()

    # Best individual
    print ga.bestIndividual()

if __name__ == "__main__":
    run_main()

```

2.7.6 Example 6 - The DB Adapters

Filename: examples/pyevolve_ex6_dbadapter.py

This example show the use of the DB Adapters (DBAdapters):

```

from pyevolve import G1DList
from pyevolve import GSimpleGA
from pyevolve import Selectors
from pyevolve import DBAdapters
from pyevolve import Statistics

# This function is the evaluation function, we want
# to give high score to more zero'ed chromosomes
def eval_func(chromosome):
    score = 0.0

    # iterate over the chromosome
    for value in chromosome:
        if value==0:
            score += 0.5
    return score

# Genome instance
genome = G1DList.G1DList(100)
genome.setParams(rangemin=0, rangemax=10)

# The evaluator function (objective function)
genome.evaluator.set(eval_func)

# Genetic Algorithm Instance
ga = GSimpleGA.GSimpleGA(genome, 666)
ga.setGenerations(80)

```

```
ga.setMutationRate(0.2)

# Create DB Adapter and set as adapter
#sqlite_adapter = DBAdapters.DBSQLite(identify="ex6", resetDB=True)
#ga.setDBAdapter(sqlite_adapter)

# Using CSV Adapter
#csvfile_adapter = DBAdapters.DBFileCSV()
#ga.setDBAdapter(csvfile_adapter)

# Using the URL Post Adapter
# urlpost_adapter = DBAdapters.DBURLPost(url="http://whatismyip.oceanus.ro/server_variables.php", po
# ga.setDBAdapter(urlpost_adapter)

# Do the evolution, with stats dump
# frequency of 10 generations
ga.evolve(freq_stats=10)

# Best individual
#print ga.bestIndividual()
```

2.7.7 Example 7 - The Rastrigin function

Filename: examples/pyevolve_ex7_rastrigin.py

This example minimizes the deceptive function Rastrigin with 20 variables:

```
from pyevolve import GSimpleGA
from pyevolve import G1DList
from pyevolve import Mutators, Initializers
from pyevolve import Selectors
from pyevolve import Consts
import math

# This is the Rastrigin Function, a deception function
def rastrigin(genome):
    n = len(genome)
    total = 0
    for i in xrange(n):
        total += genome[i]**2 - 10*math.cos(2*math.pi*genome[i])
    return (10*n) + total

def run_main():
    # Genome instance
    genome = G1DList.G1DList(20)
    genome.setParams(rangemin=-5.2, rangemax=5.30, bestrawscore=0.00, rounddecimal=2)
    genome.initializator.set(Initializers.G1DListInitializatorReal)
    genome.mutator.set(Mutators.G1DListMutatorRealGaussian)

    genome.evaluator.set(rastrigin)

    # Genetic Algorithm Instance
    ga = GSimpleGA.GSimpleGA(genome)
    ga.terminationCriteria.set(GSimpleGA.RawScoreCriteria)
    ga.setMinimax(Consts.minimaxType["minimize"])
    ga.setGenerations(3000)
```



```

ga.setCrossoverRate(0.8)
ga.setPopulationSize(100)
ga.setMutationRate(0.06)

ga.evolve(freq_stats=50)

best = ga.bestIndividual()
print best

if __name__ == "__main__":
    run_main()

```

2.7.8 Example 8 - The Gaussian Integer Mutator

Filename: examples/pyevolve_ex8_gauss_int.py

This example shows the use of the Gaussian Integer Mutator (`Mutators.G1DListMutatorIntegerGaussian`):

```

from pyevolve import G1DList
from pyevolve import GSimpleGA
from pyevolve import Selectors
from pyevolve import Mutators

# This function is the evaluation function, we want
# to give high score to more zero'ed chromosomes
def eval_func(chromosome):
    score = 0.0

    # iterate over the chromosome
    for value in chromosome:
        if value==0:
            score += 0.1
    return score

def run_main():
    # Genome instance
    genome = G1DList.G1DList(40)

    # The gauss_mu and gauss_sigma is used to the Gaussian Mutator, but
    # if you don't specify, the mutator will use the defaults
    genome.setParams(rangemin=0, rangemax=10, gauss_mu=4, gauss_sigma=6)
    genome.mutator.set(Mutators.G1DListMutatorIntegerGaussian)

    # The evaluator function (objective function)
    genome.evaluator.set(eval_func)

    # Genetic Algorithm Instance
    ga = GSimpleGA.GSimpleGA(genome)
    #ga.selector.set(Selectors.GRouletteWheel)
    ga.setGenerations(800)

    # Do the evolution, with stats dump
    # frequency of 10 generations
    ga.evolve(freq_stats=150)

```

```
# Best individual
print ga.bestIndividual()

if __name__ == "__main__":
    run_main()
```

2.7.9 Example 9 - The 2D List genome

Filename: examples/pyevolve_ex9_g2dlist.py

This example shows the use of the 2d list genome (`G2DList.G2DList`):

```
from pyevolve import G2DList
from pyevolve import GSimpleGA
from pyevolve import Selectors
from pyevolve import Crossovers
from pyevolve import Mutators

# This function is the evaluation function, we want
# to give high score to more zero'ed chromosomes
def eval_func(chromosome):
    score = 0.0

    # iterate over the chromosome
    for i in xrange(chromosome.getHeight()):
        for j in xrange(chromosome.getWidth()):
            # You can use the chromosome.getItem(i, j) too
            if chromosome[i][j]==0:
                score += 0.1
    return score

def run_main():
    # Genome instance
    genome = G2DList.G2DList(8, 5)
    genome.setParams(rangemin=0, rangemax=100)

    # The evaluator function (objective function)
    genome.evaluator.set(eval_func)
    genome.crossover.set(Crossovers.G2DListCrossoverSingleHPoint)
    genome.mutator.set(Mutators.G2DListMutatorIntegerRange)

    # Genetic Algorithm Instance
    ga = GSimpleGA.GSimpleGA(genome)
    ga.setGenerations(800)

    # Do the evolution, with stats dump
# frequency of 10 generations
    ga.evolve(freq_stats=100)

    # Best individual
    print ga.bestIndividual()

if __name__ == "__main__":
    run_main()
```

2.7.10 Example 10 - The 1D Binary String

Filename: examples/pyevolve_ex10_g1dbinstr.py

This example shows the use of the 1D Binary String genome:

```

from pyevolve import G1DBinaryString
from pyevolve import GSimpleGA
from pyevolve import Selectors
from pyevolve import Mutators

# This function is the evaluation function, we want
# to give high score to more zero'ed chromosomes
def eval_func(chromosome):
    score = 0.0

    # iterate over the chromosome
    for value in chromosome:
        if value == 0:
            score += 0.1

    return score

def run_main():
    # Genome instance
    genome = G1DBinaryString.G1DBinaryString(50)

    # The evaluator function (objective function)
    genome.evaluator.set(eval_func)
    genome.mutator.set(Mutators.G1DBinaryStringMutatorFlip)

    # Genetic Algorithm Instance
    ga = GSimpleGA.GSimpleGA(genome)
    ga.selector.set(Selectors.GTournamentSelector)
    ga.setGenerations(70)

    # Do the evolution, with stats dump
    # frequency of 10 generations
    ga.evolve(freq_stats=20)

    # Best individual
    print ga.bestIndividual()

if __name__ == "__main__":
    run_main()

```

2.7.11 Example 11 - The use of alleles

Filename: examples/pyevolve_ex11_allele.py

This example shows the use of alleles:

```

from pyevolve import G1DList
from pyevolve import GSimpleGA
from pyevolve import Mutators
from pyevolve import Initializators
from pyevolve import GAllele

```

```

# This function is the evaluation function, we want
# to give high score to more zero'ed chromosomes
def eval_func(chromosome):
    score = 0.0

    # iterate over the chromosome
    for value in chromosome:
        if value == 0:
            score += 0.5

    # Remember from the allele set defined above
    # this value 'a' is possible at this position
    if chromosome[18] == 'a':
        score += 1.0

    # Remember from the allele set defined above
    # this value 'xxx' is possible at this position
    if chromosome[12] == 'xxx':
        score += 1.0

    return score

def run_main():
    # Genome instance
    setOfAlleles = GAllele.GAlleles()

    # From 0 to 10 we can have only some
    # defined ranges of integers
    for i in xrange(11):
        a = GAllele.GAlleleRange(0, i)
        setOfAlleles.add(a)

    # From 11 to 19 we can have a set
    # of elements
    for i in xrange(11, 20):
        # You can even add objects instead of strings or
        # primitive values
        a = GAllele.GAlleleList(['a', 'b', 'xxx', 666, 0])
        setOfAlleles.add(a)

    genome = G1DList.G1DList(20)
    genome.setParams(allele=setOfAlleles)

    # The evaluator function (objective function)
    genome.evaluator.set(eval_func)

    # This mutator and initializer will take care of
    # initializing valid individuals based on the allele set
    # that we have defined before
    genome.mutator.set(Mutators.G1DListMutatorAllele)
    genome.initializator.set(Initializators.G1DListInitializatorAllele)

    # Genetic Algorithm Instance
    ga = GSimpleGA.GSimpleGA(genome)
    ga.setGenerations(40)

    # Do the evolution, with stats dump

```

```

# frequency of 10 generations
ga.evolve(freq_stats=5)

# Best individual
print ga.bestIndividual()

if __name__ == "__main__":
    run_main()

```

2.7.12 Example 12 - The Travelling Salesman Problem (TSP)

Filename: examples/pyevolve_ex12_tsp.py

This example shows the use of Pyevolve to solve the TSP:

```

from pyevolve import G1DList, GAllele
from pyevolve import GSimpleGA
from pyevolve import Mutators
from pyevolve import Crossovers
from pyevolve import Consts

import sys, random
random.seed(1024)
from math import sqrt

PIL_SUPPORT = None

try:
    from PIL import Image, ImageDraw, ImageFont
    PIL_SUPPORT = True
except:
    PIL_SUPPORT = False

cm      = []
coords = []
CITIES = 100
WIDTH  = 1024
HEIGHT = 768
LAST_SCORE = -1

def cartesian_matrix(coords):
    """ A distance matrix """
    matrix={}
    for i,(x1,y1) in enumerate(coords):
        for j,(x2,y2) in enumerate(coords):
            dx, dy = x1-x2, y1-y2
            dist=sqrt(dx*dx + dy*dy)
            matrix[i,j] = dist
    return matrix

def tour_length(matrix, tour):
    """ Returns the total length of the tour """
    total = 0
    t = tour.getInternalList()

```

```

    for i in range(CITIES):
        j = (i+1)%CITIES
        total += matrix[t[i], t[j]]
    return total

def write_tour_to_img(coords, tour, img_file):
    """ The function to plot the graph """
    padding=20
    coords=[(x+padding,y+padding) for (x,y) in coords]
    maxx,maxy=0,0
    for x,y in coords:
        maxx, maxy = max(x,maxx), max(y,maxy)
    maxx+=padding
    maxy+=padding
    img=Image.new("RGB", (int(maxx),int(maxy)), color=(255,255,255))
    font=ImageFont.load_default()
    d=ImageDraw.Draw(img);
    num_cities=len(tour)
    for i in range(num_cities):
        j=(i+1)%num_cities
        city_i=tour[i]
        city_j=tour[j]
        x1,y1=coords[city_i]
        x2,y2=coords[city_j]
        d.line((int(x1),int(y1),int(x2),int(y2)), fill=(0,0,0))
        d.text((int(x1)+7,int(y1)-5), str(i), font=font, fill=(32,32,32))

    for x,y in coords:
        x,y=int(x),int(y)
        d.ellipse((x-5,y-5,x+5,y+5), outline=(0,0,0), fill=(196,196,196))
    del d
    img.save(img_file, "PNG")
    print "The plot was saved into the %s file." % (img_file,)

def G1DListTSPInitializer(genome, **args):
    """ The initializer for the TSP """
    lst = [i for i in xrange(genome.getListSize())]
    random.shuffle(lst)
    genome.setInternalList(lst)

# This is to make a video of best individuals along the evolution
# Use mencoder to create a video with the file list list.txt
# mencoder mf://@list.txt -mf w=400:h=200:fps=3:type=png -ovc lavc
# -lavcopts vcodec=mpeg4:mbd=2:trell -oac copy -o output.avi
#

def evolve_callback(ga_engine):
    global LAST_SCORE
    if ga_engine.getCurrentGeneration() % 100 == 0:
        best = ga_engine.bestIndividual()
        if LAST_SCORE != best.getRawScore():
            write_tour_to_img( coords, best, "tspimg/tsp_result_%d.png" % ga_engine.getCurrentGeneration())
            LAST_SCORE = best.getRawScore()
    return False

def main_run():
    global cm, coords, WIDTH, HEIGHT

    coords = [(random.randint(0, WIDTH), random.randint(0, HEIGHT))

```

```

        for i in xrange(CITIES)]
    cm      = cartesian_matrix(coords)
    genome = G1DList.G1DList(len(coords))

    genome.evaluator.set(lambda chromosome: tour_length(cm, chromosome))
    genome.crossover.set(Crossovers.G1DListCrossoverEdge)
    genome.initializator.set(G1DListTSPInitializer)

    # 3662.69
    ga = GSimpleGA.GSimpleGA(genome)
    ga.setGenerations(200000)
    ga.setMinimax(Consts.minimaxType["minimize"])
    ga.setCrossoverRate(1.0)
    ga.setMutationRate(0.02)
    ga.setPopulationSize(80)

    # This is to make a video
    ga.stepCallback.set(evolve_callback)
    # 21666.49
    import psyco
    psyco.full()

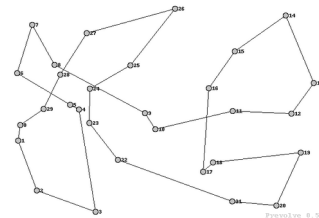
    ga.evolve(freq_stats=500)
    best = ga.bestIndividual()

    if PIL_SUPPORT:
        write_tour_to_img(coords, best, "tsp_result.png")
    else:
        print "No PIL detected, cannot plot the graph !"

if __name__ == "__main__":
    main_run()

```

This example will plot a file called `tsp_result.png` in the same directory of the execution, this image will be the best result of the TSP, it looks like:



To plot this image, you will need the Python Imaging Library (PIL).

See Also:

Python Imaging Library (PIL) The Python Imaging Library (PIL) adds image processing capabilities to your Python interpreter. This library supports many file formats, and provides powerful image processing and graphics capabilities.

2.7.13 Example 13 - The sphere function

Filename: `examples/pyevolve_ex13_sphere.py`

This is the GA to solve the sphere function:

```
from pyevolve import G1DList
from pyevolve import Mutators, Initializers
from pyevolve import GSimpleGA, Consts

# This is the Sphere Function
def sphere(xlist):
    total = 0
    for i in xlist:
        total += i**2
    return total

def run_main():
    genome = G1DList.G1DList(140)
    genome.setParams(rangemin=-5.12, rangemax=5.13)
    genome.initializer.set(Initializers.G1DListInitializerReal)
    genome.mutator.set(Mutators.G1DListMutatorRealGaussian)
    genome.evaluator.set(sphere)

    ga = GSimpleGA.GSimpleGA(genome, seed=666)
    ga.setMinimax(Consts.minimaxType["minimize"])
    ga.setGenerations(1500)
    ga.setMutationRate(0.01)
    ga.evolve(freq_stats=500)

    best = ga.bestIndividual()

if __name__ == "__main__":
    run_main()
```

2.7.14 Example 14 - The Ackley function

Filename: examples/pyevolve_ex14_ackley.py

This example minimizes the Ackley F1 function, a deceptive function:

```
from pyevolve import G1DList, GSimpleGA, Selectors
from pyevolve import Initializers, Mutators, Consts, DBAdapters
import math

# This is the Rastringin Function, a deception function
def ackley(xlist):
    sum1 = 0
    score = 0
    n = len(xlist)
    for i in xrange(n):
        sum1 += xlist[i]*xlist[i]
    t1 = math.exp(-0.2*(math.sqrt((1.0/5.0)*sum1)))

    sum1 = 0
    for i in xrange(n):
        sum1 += math.cos(2.0*math.pi*xlist[i]);
    t2 = math.exp((1.0/5.0)*sum1);
    score = 20 + math.exp(1) - 20 * t1 - t2;

    return score
```



```

def run_main():
    # Genome instance
    genome = G1DList.G1DList(5)
    genome.setParams(rangemin=-8, rangemax=8, bestrawscore=0.00, rounddecimal=2)
    genome.initializator.set(Initializators.G1DListInitializatorReal)
    genome.mutator.set(Mutators.G1DListMutatorRealGaussian)

    # The evaluator function (objective function)
    genome.evaluator.set(ackley)

    # Genetic Algorithm Instance
    ga = GSimpleGA.GSimpleGA(genome)
    ga.setMinimax(Consts.minimaxType["minimize"])
    ga.setGenerations(1000)
    ga.setMutationRate(0.04)
    ga.terminationCriteria.set(GSimpleGA.RawScoreCriteria)

    # Create DB Adapter and set as adapter
    # sqlite_adapter = DBAdapters.DBSQLite(identify="ackley")
    # ga.setDBAdapter(sqlite_adapter)

    # Do the evolution, with stats dump
    # frequency of 10 generations
    ga.evolve(freq_stats=50)

    # Best individual
    best = ga.bestIndividual()
    print "\nBest individual score: %.2f" % (best.getRawScore(),)
    print best

if __name__ == "__main__":
    run_main()

```

2.7.15 Example 15 - The Rosenbrock function

Filename: examples/pyevolve_ex15_rosenbrock.py

This example minimizes the Rosenbrock function, another deceptive function:

```

from pyevolve import G1DList, GSimpleGA, Selectors, Statistics
from pyevolve import Initializators, Mutators, Consts, DBAdapters

# This is the Rosenbrock Function
def rosenbrock(xlist):
    sum_var = 0
    for x in xrange(1, len(xlist)):
        sum_var += 100.0 * (xlist[x] - xlist[x-1]**2)**2 + (1 - xlist[x-1])**2
    return sum_var

def run_main():
    # Genome instance
    genome = G1DList.G1DList(15)
    genome.setParams(rangemin=-1, rangemax=1.1)
    genome.initializator.set(Initializators.G1DListInitializatorReal)
    genome.mutator.set(Mutators.G1DListMutatorRealRange)

```

```

# The evaluator function (objective function)
genome.evaluator.set(rosenbrock)

# Genetic Algorithm Instance
ga = GSimpleGA.GSimpleGA(genome)
ga.setMinimax(Consts.minimaxType["minimize"])
ga.selector.set(Selectors.GRouletteWheel)
ga.setGenerations(4000)
ga.setCrossoverRate(0.9)
ga.setPopulationSize(100)
ga.setMutationRate(0.03)

ga.evolve(freq_stats=500)

# Best individual
best = ga.bestIndividual()
print "\nBest individual score: %.2f" % (best.score,)
print best

if __name__ == "__main__":
    run_main()

```

2.7.16 Example 16 - The 2D Binary String

Filename: examples/pyevolve_ex16_g2dbinstr.py

This example shows the use of the 2D Binary String genome:

```

from pyevolve import G2DBinaryString
from pyevolve import GSimpleGA
from pyevolve import Selectors
from pyevolve import Crossovers
from pyevolve import Mutators

# This function is the evaluation function, we want
# to give high score to more zero'ed chromosomes
def eval_func(chromosome):
    score = 0.0

    # iterate over the chromosome
    for i in xrange(chromosome.getHeight()):
        for j in xrange(chromosome.getWidth()):
            # You can use the chromosome.getItem(i, j)
            if chromosome[i][j]==0:
                score += 0.1
    return score

# Genome instance
genome = G2DBinaryString.G2DBinaryString(8, 5)

# The evaluator function (objective function)
genome.evaluator.set(eval_func)
genome.crossover.set(Crossovers.G2DBinaryStringXSingleHPoint)
genome.mutator.set(Mutators.G2DBinaryStringMutatorSwap)

```

```
# Genetic Algorithm Instance
ga = GSimpleGA.GSimpleGA(genome)
ga.setGenerations(200)

# Do the evolution, with stats dump
# frequency of 10 generations
ga.evolve(freq_stats=10)

# Best individual
print ga.bestIndividual()
```

2.7.17 Example 17 - The Tree genome example

Filename: examples/pyevolve_ex17_gtree.py

This example shows the use of the Tree genome:

```
from pyevolve import GSimpleGA
from pyevolve import GTree
from pyevolve import Crossovers
from pyevolve import Mutators
import time
import random

def eval_func(chromosome):
    score = 0.0
    # If you want to add score values based
    # in the height of the Tree, the extra
    # code is commented.

    #height = chromosome.getHeight()

    for node in chromosome:
        score += (100 - node.getData())*0.1

    #if height <= chromosome.getParam("max_depth"):
    #    score += (score*0.8)

    return score

def run_main():
    genome = GTree.GTree()
    root = GTree.GTreeNode(2)
    genome.setRoot(root)
    genome.processNodes()

    genome.setParams(max_depth=3, max_siblings=2, method="grow")
    genome.evaluator.set(eval_func)
    genome.crossover.set(Crossovers.GTreeCrossoverSinglePointStrict)

    ga = GSimpleGA.GSimpleGA(genome)
    ga.setGenerations(100)
    ga.setMutationRate(0.05)

    ga.evolve(freq_stats=10)
    print ga.bestIndividual()
```

```
if __name__ == "__main__":
    run_main()
```

2.7.18 Example 18 - The Genetic Programming example

Filename: examples/pyevolve_ex18_gp.py

This example shows the use of the GTreeGP genome (for Genetic Programming):

```
from pyevolve import Util
from pyevolve import GTree
from pyevolve import GSimpleGA
from pyevolve import Consts
import math

rmse_accum = Util.ErrorAccumulator()

def gp_add(a, b): return a+b
def gp_sub(a, b): return a-b
def gp_mul(a, b): return a*b
def gp_sqrt(a): return math.sqrt(abs(a))

def eval_func(chromosome):
    global rmse_accum
    rmse_accum.reset()
    code_comp = chromosome.getCompiledCode()

    for a in xrange(0, 5):
        for b in xrange(0, 5):
            evaluated = eval(code_comp)
            target = math.sqrt((a*a)+(b*b))
            rmse_accum += (target, evaluated)

    return rmse_accum.getRMSE()

def main_run():
    genome = GTree.GTreeGP()
    genome.setParams(max_depth=4, method="ramped")
    genome.evaluator += eval_func

    ga = GSimpleGA.GSimpleGA(genome)
    ga.setParams(gp_terminals = ['a', 'b'],
                gp_function_prefix = "gp")

    ga.setMinimax(Consts.minimaxType["minimize"])
    ga.setGenerations(50)
    ga.setCrossoverRate(1.0)
    ga.setMutationRate(0.25)
    ga.setPopulationSize(800)

    ga(freq_stats=10)
    best = ga.bestIndividual()
    print best

if __name__ == "__main__":
    main_run()
```

2.7.19 Example 21 - The n-queens problem (64x64 chess board)

Filename: examples/pyevolve_ex21_nqueens.py

This example shows the use of GA to solve the n-queens problem for a chess board of size 64x64:

```

from pyevolve import G1DList
from pyevolve import Mutators, Crossovers
from pyevolve import Consts, GSimpleGA
from pyevolve import DBAdapters
from random import shuffle

# The "n" in n-queens
BOARD_SIZE = 64

# The n-queens fitness function
def queens_eval(genome):
    collisions = 0
    for i in xrange(0, BOARD_SIZE):
        if i not in genome: return 0
    for i in xrange(0, BOARD_SIZE):
        col = False
        for j in xrange(0, BOARD_SIZE):
            if (i != j) and (abs(i-j) == abs(genome[j]-genome[i])):
                col = True
        if col == True: collisions +=1
    return BOARD_SIZE-collisions

def queens_init(genome, **args):
    genome.genomeList = range(0, BOARD_SIZE)
    shuffle(genome.genomeList)

def run_main():
    genome = G1DList.G1DList(BOARD_SIZE)
    genome.setParams(bestrawscore=BOARD_SIZE, rounddecimal=2)
    genome.initializator.set(queens_init)
    genome.mutator.set(Mutators.G1DListMutatorSwap)
    genome.crossover.set(Crossovers.G1DListCrossoverCutCrossfill)
    genome.evaluator.set(queens_eval)

    ga = GSimpleGA.GSimpleGA(genome)
    ga.terminationCriteria.set(GSimpleGA.RawScoreCriteria)
    ga.setMinimax(Consts.minimaxType["maximize"])

    ga.setPopulationSize(100)
    ga.setGenerations(250)
    ga.setMutationRate(0.02)
    ga.setCrossoverRate(1.0)

    #sqlite_adapter = DBAdapters.DBSQLite(identify="queens")
    #ga.setDBAdapter(sqlite_adapter)

    vpython_adapter = DBAdapters.DBVPythonGraph(identify="queens", frequency=1)
    ga.setDBAdapter(vpython_adapter)

    ga.evolve(freq_stats=10)

    best = ga.bestIndividual()

```

```

    print best
    print "Best individual score: %.2f\n" % (best.getRawScore(),)

if __name__ == "__main__":
    run_main()

```

2.7.20 Example 22 - The Infinite Monkey Theorem

Filename: examples/pyevolve_ex22_monkey.py

This example was a kindly contribution by Jelle Feringa, it shows the [Infinite Monkey Theorem](#):

```

=====
# Pyevolve version of the Infinite Monkey Theorem
# See: http://en.wikipedia.org/wiki/Infinite\_monkey\_theorem
# By Jelle Feringa
=====

from pyevolve import G1DList
from pyevolve import GSimpleGA, Consts
from pyevolve import Selectors
from pyevolve import Initializers, Mutators, Crossovers
import math

sentence = """
'Just living is not enough,' said the butterfly,
'one must have sunshine, freedom, and a little flower.'
"""
numeric_sentence = map(ord, sentence)

def evolve_callback(ga_engine):
    generation = ga_engine.getCurrentGeneration()
    if generation%50==0:
        indiv = ga_engine.bestIndividual()
        print ''.join(map(chr, indiv))
    return False

def run_main():
    genome = G1DList.G1DList(len(sentence))
    genome.setParams(rangemin=min(numeric_sentence),
                    rangemax=max(numeric_sentence),
                    bestrawscore=0.00,
                    gauss_mu=1, gauss_sigma=4)

    genome.initializator.set(Initializers.G1DListInitializerInteger)
    genome.mutator.set(Mutators.G1DListMutatorIntegerGaussian)
    genome.evaluator.set(lambda genome: sum(
        [abs(a-b) for a, b in zip(genome, numeric_sentence)]
    ))

    ga = GSimpleGA.GSimpleGA(genome)
    #ga.stepCallback.set(evolve_callback)
    ga.setMinimax(Consts.minimaxType["minimize"])
    ga.terminationCriteria.set(GSimpleGA.RawScoreCriteria)
    ga.setPopulationSize(60)
    ga.setMutationRate(0.02)

```

```
ga.setCrossoverRate(0.9)
ga.setGenerations(5000)
ga.evolve(freq_stats=100)

best = ga.bestIndividual()
print "Best individual score: %.2f" % (best.score,)
print ''.join(map(chr, best))

if __name__ == "__main__":
    run_main()
```

2.8 F.A.Q.

What is Pyevolve ?


Pyevolve is an Evolutionary Computation framework written in pure python.

Why you have created this framework ?

Python is a powerful language, the features of Python together with the Evolutionary Algorithms (EA) can be very interesting. There is no good GA library written in Python today, the main effort of the Pyevolve is to solve this problem, since the Evolutionary Computation research field is growing faster.

2.9 Contributors


Boris Gorelik, from Procognia Ltd, Blog inthehaystack.com, @boris_gorelik, Ashdod, Israel.

 boris@gorelik.net

Amit Saha, Blog amitsaha.wordpress.com, @amitsaha, India.

 amitsaha.in@gmail.com

Jelle Feringa, Jelle Feringa, Tu Delft University, Netherlands.

 jelleferinga@gmail.com

Henrik Rudstrom, Blog uniqueidentifier.net, @henrk, Rotterdam, Netherlands.

 hrudstrom@gmail.com

2.10 License

The Pyevolve license is based on the [PSF](http://www.python.org/psf/license/) license, which is GPL compatible license.

LICENSE AGREEMENT FOR PYEVOLVE 0.6

1. This LICENSE AGREEMENT is between Christian S. Perone (“CSP”), and the Individual or Organization (“Licensee”) accessing and otherwise using Pyevolve software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CSP hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Pyevolve 0.6 alone or in any derivative version, provided, however, that CSP’s License Agreement and CSP’s notice of copyright, i.e., “Copyright (c) 2007-2010 Christian S. Perone; All Rights Reserved” are retained in Pyevolve 0.6 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Pyevolve 0.6 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Pyevolve 0.6.
4. CSP is making Pyevolve 0.6 available to Licensee on an “AS IS” basis. CSP MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CSP MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF Pyevolve 0.6 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CSP SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF Pyevolve 0.6 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING Pyevolve 0.6, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CSP and Licensee. This License Agreement does not grant permission to use CSP trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Pyevolve 0.6, Licensee agrees to be bound by the terms and conditions of this License Agreement.

2.11 Credits

Pyevolve was written by Christian S. Perone and is currently maintained by himself and it’s contributors.

2.12 Contact the author

If you have any question or suggestion, you can contact me at:

christian.perone@gmail.com



Please use the word “pyevolve” in the the subject of mail.

2.13 Donate

Please, [donate](#) to support the development of this project.

INDEX

- *Index*
- *Module Index*

This documentation was updated on April 25, 2010.

INDEX

- add() (FunctionSlot.FunctionSlot method), 56
- add() (GAllele.GAlleleList method), 79
- add() (GAllele.GAlleleRange method), 79
- add() (GAllele.GAlleles method), 80
- addChild() (GenomeBase.GTreeNodeBase method), 76
- addChild() (GTree.GTreeNode method), 103
- addChild() (GTree.GTreeNodeGP method), 104
- addData() (Network.UDPThreadUnicastClient method), 41
- addEdge() (Util.Graph method), 37
- addNode() (Util.Graph method), 37
- Adjusted Fitness, 13
- append() (G1DBinaryString.G1DBinaryString method), 81
- append() (G1DList.G1DList method), 89
- append() (GenomeBase.G1DBase method), 74
- append() (Util.ErrorAccumulator method), 36
- apply() (FunctionSlot.FunctionSlot method), 57
- applyFunctions() (FunctionSlot.FunctionSlot method), 57
- asTuple() (Statistics.Statistics method), 58

- bestFitness() (GPopulation.GPopulation method), 65
- bestIndividual() (GSimpleGA.GSimpleGA method), 59
- bestRaw() (GPopulation.GPopulation method), 65
- BoltzmannScaling() (in module Scaling), 73
- buildGTreeFull() (in module GTree), 105
- buildGTreeGPFull() (in module GTree), 105
- buildGTreeGPGrow() (in module GTree), 106
- buildGTreeGrow() (in module GTree), 106

- CDefBroadcastAddress (in module Consts), 31
- CDefCSVFileName (in module Consts), 35
- CDefCSVFileStatsGenFreq (in module Consts), 35
- CDefESCKey (in module Consts), 31
- CDefG1DBinaryStringCrossover (in module Consts), 32
- CDefG1DBinaryStringInit (in module Consts), 32
- CDefG1DBinaryStringMutator (in module Consts), 32
- CDefG1DBinaryStringUniformProb (in module Consts), 32
- CDefG1DListMutIntMU (in module Consts), 33
- CDefG1DListMutIntSIGMA (in module Consts), 33
- CDefG1DListMutRealMU (in module Consts), 33
- CDefG1DListMutRealSIGMA (in module Consts), 33
- CDefG2DBinaryStringCrossover (in module Consts), 33
- CDefG2DBinaryStringInit (in module Consts), 33
- CDefG2DBinaryStringMutator (in module Consts), 33
- CDefG2DBinaryStringUniformProb (in module Consts), 33
- CDefG2DListCrossover (in module Consts), 34
- CDefG2DListCrossUniformProb (in module Consts), 34
- CDefG2DListInit (in module Consts), 34
- CDefG2DListMutator (in module Consts), 34
- CDefG2DListMutIntMU (in module Consts), 33
- CDefG2DListMutIntSIGMA (in module Consts), 34
- CDefG2DListMutRealMU (in module Consts), 33
- CDefG2DListMutRealSIGMA (in module Consts), 33
- CDefGACrossoverRate (in module Consts), 34
- CDefGAGenerations (in module Consts), 34
- CDefGAMutationRate (in module Consts), 34
- CDefGAPopulationSize (in module Consts), 34
- CDefGASelector (in module Consts), 34
- CDefGenMigrationRate (in module Consts), 35
- CDefGenMigrationReplacement (in module Consts), 35
- CDefGGTreeMutator (in module Consts), 33
- CDefGPGenomes (in module Consts), 31
- CDefGTreeCrossover (in module Consts), 33
- CDefGTreeInit (in module Consts), 33
- CDefImportList (in module Consts), 31
- CDefLogFile (in module Consts), 31
- CDefLogLevel (in module Consts), 31
- CDefMigrationNIndividuals (in module Consts), 35
- CDefMySQLDBHost (in module Consts), 35
- CDefMySQLDBName (in module Consts), 35
- CDefMySQLDBPort (in module Consts), 35
- CDefMySQLDBTable (in module Consts), 35
- CDefMySQLDBTablePop (in module Consts), 35
- CDefMySQLStatsCommitFreq (in module Consts), 35
- CDefMySQLStatsGenFreq (in module Consts), 35
- CDefNetworkIndividual (in module Consts), 35
- CDefNetworkInfo (in module Consts), 35
- CDefPopMinimax (in module Consts), 32
- CDefPopScale (in module Consts), 32
- CDefPopSortType (in module Consts), 32
- CDefPythonRequire (in module Consts), 31

- CDefRangeMax (in module Consts), 31
- CDefRangeMin (in module Consts), 31
- CDefScaleBoltzFactor (in module Consts), 32
- CDefScaleBoltzMinTemp (in module Consts), 32
- CDefScaleBoltzStart (in module Consts), 32
- CDefScaleLinearMultiplier (in module Consts), 32
- CDefScalePowerLawFactor (in module Consts), 32
- CDefScaleSigmaTruncMultiplier (in module Consts), 32
- CDefSQLiteDBName (in module Consts), 34
- CDefSQLiteDBTable (in module Consts), 34
- CDefSQLiteDBTablePop (in module Consts), 34
- CDefSQLiteStatsCommitFreq (in module Consts), 34
- CDefSQLiteStatsGenFreq (in module Consts), 34
- CDefTournamentPoolSize (in module Consts), 32
- CDefURLPostStatsGenFreq (in module Consts), 35
- CDefXMLRPCStatsGenFreq (in module Consts), 35
- checkTerminal() (in module GTree), 106
- clear() (FunctionSlot.FunctionSlot method), 57
- clear() (GAllele.GAlleleList method), 79
- clear() (GAllele.GAlleleRange method), 79
- clear() (GPopulation.GPopulation method), 65
- clear() (GSimpleGA.GSimpleGA method), 60
- clear() (Statistics.Statistics method), 58
- clearFlags() (GPopulation.GPopulation method), 65
- clearList() (G1DBinaryString.G1DBinaryString method), 81
- clearList() (G1DList.G1DList method), 90
- clearList() (G2DList.G2DList method), 93
- clearList() (GenomeBase.G1DBase method), 74
- clearString() (G2DBinaryString.G2DBinaryString method), 85
- clone() (G1DBinaryString.G1DBinaryString method), 81
- clone() (G1DList.G1DList method), 90
- clone() (G2DBinaryString.G2DBinaryString method), 85
- clone() (G2DList.G2DList method), 93
- clone() (GenomeBase.GenomeBase method), 77
- clone() (GenomeBase.GTreeBase method), 75
- clone() (GenomeBase.GTreeNodeBase method), 76
- clone() (GPopulation.GPopulation method), 65
- clone() (GTree.GTree method), 96
- clone() (GTree.GTreeGP method), 99
- clone() (GTree.GTreeNode method), 103
- clone() (GTree.GTreeNodeGP method), 104
- clone() (Statistics.Statistics method), 58
- close() (DBAdapters.DBFileCSV method), 48
- close() (DBAdapters.DBMySQLAdapter method), 50
- close() (DBAdapters.DBSQLite method), 51
- close() (Network.UDPThreadBroadcastClient method), 40
- close() (Network.UDPThreadServer method), 40
- close() (Network.UDPThreadUnicastClient method), 41
- cmp_individual_raw() (in module Util), 37
- cmp_individual_scaled() (in module Util), 37
- commit() (DBAdapters.DBMySQLAdapter method), 50
- commit() (DBAdapters.DBSQLite method), 51
- commitAndClose() (DBAdapters.DBBaseAdapter method), 47
- commitAndClose() (DBAdapters.DBFileCSV method), 48
- commitAndClose() (DBAdapters.DBMySQLAdapter method), 50
- commitAndClose() (DBAdapters.DBSQLite method), 51
- commitAndClose() (DBAdapters.DBURLPost method), 53
- commitAndClose() (DBAdapters.DBVPythonGraph method), 54
- commitAndClose() (DBAdapters.DBXMLRPC method), 55
- compare() (GTree.GTreeGP method), 99
- compare() (GTree.GTreeNodeGP method), 104
- Consts (module), 30
- ConvergenceCriteria() (in module GSimpleGA), 59
- copy() (G1DBinaryString.G1DBinaryString method), 82
- copy() (G1DList.G1DList method), 90
- copy() (G2DBinaryString.G2DBinaryString method), 85
- copy() (G2DList.G2DList method), 93
- copy() (GenomeBase.G1DBase method), 74
- copy() (GenomeBase.GenomeBase method), 77
- copy() (GenomeBase.GTreeBase method), 75
- copy() (GenomeBase.GTreeNodeBase method), 76
- copy() (GPopulation.GPopulation method), 65
- copy() (GTree.GTree method), 97
- copy() (GTree.GTreeGP method), 99
- copy() (GTree.GTreeNode method), 103
- copy() (GTree.GTreeNodeGP method), 104
- copy() (Statistics.Statistics method), 58
- create() (GPopulation.GPopulation method), 66
- createStructure() (DBAdapters.DBMySQLAdapter method), 50
- createStructure() (DBAdapters.DBSQLite method), 51
- crossover (G1DBinaryString.G1DBinaryString attribute), 82
- crossover (G1DList.G1DList attribute), 90
- crossover (G2DBinaryString.G2DBinaryString attribute), 85
- crossover (G2DList.G2DList attribute), 93
- crossover (GenomeBase.GenomeBase attribute), 77
- crossover (GTree.GTree attribute), 97
- Crossovers (module), 69
- Data Type Independent, 13
- DBAdapters (module), 47
- DBBaseAdapter (class in DBAdapters), 47
- DBFileCSV (class in DBAdapters), 48
- DBMySQLAdapter (class in DBAdapters), 49
- DBSQLite (class in DBAdapters), 51
- DBURLPost (class in DBAdapters), 52
- DBVPythonGraph (class in DBAdapters), 53

- DBXMLRPC (class in DBAdapters), 54
- dumpStatsDB() (GSimpleGA.GSimpleGA method), 60
- ErrorAccumulator (class in Util), 36
- evaluate() (G1DBinaryString.G1DBinaryString method), 82
- evaluate() (G1DList.G1DList method), 90
- evaluate() (G2DBinaryString.G2DBinaryString method), 86
- evaluate() (G2DList.G2DList method), 94
- evaluate() (GenomeBase.GenomeBase method), 77
- evaluate() (GPopulation.GPopulation method), 66
- evaluate() (GTree.GTree method), 97
- evaluate() (GTree.GTreeGP method), 99
- Evaluation function, 12
- evaluator (G1DBinaryString.G1DBinaryString attribute), 82
- evaluator (G1DList.G1DList attribute), 90
- evaluator (G2DBinaryString.G2DBinaryString attribute), 86
- evaluator (G2DList.G2DList attribute), 94
- evaluator (GenomeBase.GenomeBase attribute), 77
- evaluator (GTree.GTree attribute), 97
- evolve() (GSimpleGA.GSimpleGA method), 60
- exchange() (Migration.MigrationScheme method), 43
- exchange() (Migration.WANMigration method), 44
- ExponentialScaling() (in module Scaling), 73
- Fitness score, 12
- FitnessStatsCriteria() (in module GSimpleGA), 59
- FunctionSlot (class in FunctionSlot), 56
- FunctionSlot (module), 56
- G1DBase (class in GenomeBase), 74
- G1DBinaryString (class in G1DBinaryString), 81
- G1DBinaryString (module), 80
- G1DBinaryStringInitializer() (in module Initializers), 71
- G1DBinaryStringMutatorFlip() (in module Mutators), 68
- G1DBinaryStringMutatorSwap() (in module Mutators), 68
- G1DBinaryStringXSinglePoint() (in module Crossovers), 70
- G1DBinaryStringXTwoPoint() (in module Crossovers), 70
- G1DBinaryStringXUniform() (in module Crossovers), 70
- G1DList (class in G1DList), 88
- G1DList (module), 88
- G1DListCrossoverCutCrossfill() (in module Crossovers), 70
- G1DListCrossoverEdge() (in module Crossovers), 70
- G1DListCrossoverOX() (in module Crossovers), 70
- G1DListCrossoverRealSBX() (in module Crossovers), 70
- G1DListCrossoverSinglePoint() (in module Crossovers), 70
- G1DListCrossoverTwoPoint() (in module Crossovers), 70
- G1DListCrossoverUniform() (in module Crossovers), 70
- G1DListGetEdges() (in module Util), 36
- G1DListGetEdgesComposite() (in module Util), 36
- G1DListInitializerAllele() (in module Initializers), 71
- G1DListInitializerInteger() (in module Initializers), 71
- G1DListInitializerReal() (in module Initializers), 71
- G1DListMergeEdges() (in module Util), 36
- G1DListMutatorAllele() (in module Mutators), 68
- G1DListMutatorIntegerBinary() (in module Mutators), 68
- G1DListMutatorIntegerGaussian() (in module Mutators), 68
- G1DListMutatorIntegerRange() (in module Mutators), 68
- G1DListMutatorRealGaussian() (in module Mutators), 68
- G1DListMutatorRealRange() (in module Mutators), 68
- G1DListMutatorSIM() (in module Mutators), 68
- G1DListMutatorSwap() (in module Mutators), 68
- G2DBinaryString (class in G2DBinaryString), 85
- G2DBinaryString (module), 84
- G2DBinaryStringInitializer() (in module Initializers), 71
- G2DBinaryStringMutatorFlip() (in module Mutators), 68
- G2DBinaryStringMutatorSwap() (in module Mutators), 68
- G2DBinaryStringXSingleHPoint() (in module Crossovers), 70
- G2DBinaryStringXSingleVPoint() (in module Crossovers), 70
- G2DBinaryStringXUniform() (in module Crossovers), 70
- G2DList (class in G2DList), 92
- G2DList (module), 92
- G2DListCrossoverSingleHPoint() (in module Crossovers), 70
- G2DListCrossoverSingleVPoint() (in module Crossovers), 71
- G2DListCrossoverUniform() (in module Crossovers), 71
- G2DListInitializerAllele() (in module Initializers), 71
- G2DListInitializerInteger() (in module Initializers), 72
- G2DListInitializerReal() (in module Initializers), 72
- G2DListMutatorAllele() (in module Mutators), 68
- G2DListMutatorIntegerGaussian() (in module Mutators), 69
- G2DListMutatorIntegerRange() (in module Mutators), 69
- G2DListMutatorRealGaussian() (in module Mutators), 69
- G2DListMutatorSwap() (in module Mutators), 69
- GAllele (module), 78
- GAlleleList (class in GAllele), 79
- GAlleleRange (class in GAllele), 79
- GAlleles (class in GAllele), 80
- GenomeBase (class in GenomeBase), 77
- GenomeBase (module), 74

- getAdjusted() (Util.ErrorAccumulator method), 36
- getAllNodes() (GenomeBase.GTreeBase method), 75
- getAllNodes() (GTree.GTree method), 97
- getAllNodes() (GTree.GTreeGP method), 99
- getBinary() (G1DBinaryString.G1DBinaryString method), 82
- getBufferSize() (Network.UDPThreadServer method), 40
- getChild() (GenomeBase.GTreeNodeBase method), 76
- getChild() (GTree.GTreeNode method), 103
- getChild() (GTree.GTreeNodeGP method), 104
- getChilds() (GenomeBase.GTreeNodeBase method), 76
- getChilds() (GTree.GTreeNode method), 103
- getChilds() (GTree.GTreeNodeGP method), 104
- getCompiledCode() (GTree.GTreeGP method), 99
- getCompressionLevel() (Migration.MigrationScheme method), 43
- getCompressionLevel() (Migration.WANMigration method), 44
- getCurrentGeneration() (GSimpleGA.GSimpleGA method), 60
- getCursor() (DBAdapters.DBMySQLAdapter method), 50
- getCursor() (DBAdapters.DBSQLite method), 51
- getData() (GTree.GTreeNode method), 103
- getData() (GTree.GTreeNodeGP method), 104
- getData() (Network.UDPThreadBroadcastClient method), 40
- getData() (Network.UDPThreadServer method), 40
- getDBAdapter() (GSimpleGA.GSimpleGA method), 60
- getDecimal() (G1DBinaryString.G1DBinaryString method), 82
- getFitnessScore() (G1DBinaryString.G1DBinaryString method), 83
- getFitnessScore() (G1DList.G1DList method), 90
- getFitnessScore() (G2DBinaryString.G2DBinaryString method), 86
- getFitnessScore() (G2DList.G2DList method), 94
- getFitnessScore() (GenomeBase.GenomeBase method), 77
- getFitnessScore() (GTree.GTree method), 97
- getFitnessScore() (GTree.GTreeGP method), 100
- getGenerations() (GSimpleGA.GSimpleGA method), 60
- getGPMODE() (GSimpleGA.GSimpleGA method), 60
- getGroupName() (Migration.MigrationScheme method), 43
- getGroupName() (Migration.WANMigration method), 45
- getHeight() (G2DBinaryString.G2DBinaryString method), 86
- getHeight() (G2DList.G2DList method), 94
- getHeight() (GenomeBase.GTreeBase method), 75
- getHeight() (GTree.GTree method), 97
- getHeight() (GTree.GTreeGP method), 100
- getIdentify() (DBAdapters.DBBaseAdapter method), 47
- getIdentify() (DBAdapters.DBFileCSV method), 48
- getIdentify() (DBAdapters.DBMySQLAdapter method), 50
- getIdentify() (DBAdapters.DBSQLite method), 51
- getIdentify() (DBAdapters.DBURLPost method), 53
- getIdentify() (DBAdapters.DBVPythonGraph method), 54
- getIdentify() (DBAdapters.DBXMLRPC method), 55
- getInteractiveGeneration() (GSimpleGA.GSimpleGA method), 60
- getInternalList() (G1DBinaryString.G1DBinaryString method), 83
- getInternalList() (G1DList.G1DList method), 90
- getInternalList() (GenomeBase.G1DBase method), 74
- getItem() (G2DBinaryString.G2DBinaryString method), 86
- getItem() (G2DList.G2DList method), 94
- getListSize() (G1DBinaryString.G1DBinaryString method), 83
- getListSize() (G1DList.G1DList method), 90
- getListSize() (GenomeBase.G1DBase method), 74
- getMachineIP() (in module Network), 42
- getMaximum() (GAllele.GAlleleRange method), 79
- getMean() (Util.ErrorAccumulator method), 36
- getMigrationRate() (Migration.MigrationScheme method), 43
- getMigrationRate() (Migration.WANMigration method), 45
- getMinimax() (GSimpleGA.GSimpleGA method), 60
- getMinimum() (GAllele.GAlleleRange method), 80
- getMSE() (Util.ErrorAccumulator method), 36
- getNeighbors() (Util.Graph method), 37
- getNodeDepth() (GenomeBase.GTreeBase method), 75
- getNodeDepth() (GTree.GTree method), 97
- getNodeDepth() (GTree.GTreeGP method), 100
- getNodeHeight() (GenomeBase.GTreeBase method), 75
- getNodeHeight() (GTree.GTree method), 97
- getNodeHeight() (GTree.GTreeGP method), 100
- getNodes() (Util.Graph method), 37
- getNodesCount() (GenomeBase.GTreeBase method), 75
- getNodesCount() (GTree.GTree method), 97
- getNodesCount() (GTree.GTreeGP method), 100
- getNonSquared() (Util.ErrorAccumulator method), 36
- getNumIndividuals() (Migration.MigrationScheme method), 43
- getNumIndividuals() (Migration.WANMigration method), 45
- getNumReplacement() (Migration.MigrationScheme method), 43
- getNumReplacement() (Migration.WANMigration method), 45
- getParam() (G1DBinaryString.G1DBinaryString method), 83
- getParam() (G1DList.G1DList method), 90

- getParam() (G2DBinaryString.G2DBinaryString method), 86
- getParam() (G2DList.G2DList method), 94
- getParam() (GenomeBase.GenomeBase method), 77
- getParam() (GPopulation.GPopulation method), 66
- getParam() (GSimpleGA.GSimpleGA method), 60
- getParam() (GTree.GTree method), 97
- getParam() (GTree.GTreeGP method), 100
- getParent() (GenomeBase.GTreeNodeBase method), 77
- getParent() (GTree.GTreeNode method), 103
- getParent() (GTree.GTreeNodeGP method), 105
- getPopScores() (in module Interaction), 46
- getPopulation() (GSimpleGA.GSimpleGA method), 61
- getPreOrderExpression() (GTree.GTreeGP method), 100
- getRandomAllele() (GAllele.GAlleleList method), 79
- getRandomAllele() (GAllele.GAlleleRange method), 80
- getRandomNode() (GenomeBase.GTreeBase method), 75
- getRandomNode() (GTree.GTree method), 98
- getRandomNode() (GTree.GTreeGP method), 100
- getRawScore() (G1DBinaryString.G1DBinaryString method), 83
- getRawScore() (G1DList.G1DList method), 91
- getRawScore() (G2DBinaryString.G2DBinaryString method), 86
- getRawScore() (G2DList.G2DList method), 94
- getRawScore() (GenomeBase.GenomeBase method), 78
- getRawScore() (GTree.GTree method), 98
- getRawScore() (GTree.GTreeGP method), 100
- getReal() (GAllele.GAlleleRange method), 80
- getRMSE() (Util.ErrorAccumulator method), 36
- getRoot() (GenomeBase.GTreeBase method), 75
- getRoot() (GTree.GTree method), 98
- getRoot() (GTree.GTreeGP method), 100
- getSentBytes() (Network.UDPThreadBroadcastClient method), 40
- getSExpression() (GTree.GTreeGP method), 101
- getSize() (G2DBinaryString.G2DBinaryString method), 86
- getSize() (G2DList.G2DList method), 94
- getSquared() (Util.ErrorAccumulator method), 36
- getStatistics() (GPopulation.GPopulation method), 66
- getStatistics() (GSimpleGA.GSimpleGA method), 61
- getStatsGenFreq() (DBAdapters.DBBaseAdapter method), 47
- getStatsGenFreq() (DBAdapters.DBFileCSV method), 48
- getStatsGenFreq() (DBAdapters.DBMySQLAdapter method), 50
- getStatsGenFreq() (DBAdapters.DBSQLite method), 52
- getStatsGenFreq() (DBAdapters.DBURLPost method), 53
- getStatsGenFreq() (DBAdapters.DBVPythonGraph method), 54
- getStatsGenFreq() (DBAdapters.DBXMLRPC method), 55
- getTraversalString() (GenomeBase.GTreeBase method), 75
- getTraversalString() (GTree.GTree method), 98
- getTraversalString() (GTree.GTreeGP method), 101
- getType() (GTree.GTreeNodeGP method), 105
- getWidth() (G2DBinaryString.G2DBinaryString method), 87
- getWidth() (G2DList.G2DList method), 95
- gpdec() (in module GTree), 106
- GPopulation (class in GPopulation), 64
- GPopulation (module), 64
- GRankSelector() (in module Selectors), 72
- Graph (class in Util), 37
- GRouletteWheel() (in module Selectors), 72
- GRouletteWheel_PrepareWheel() (in module Selectors), 72
- GSimpleGA (class in GSimpleGA), 59
- GSimpleGA (module), 58
- GTournamentSelector() (in module Selectors), 72
- GTournamentSelectorAlternative() (in module Selectors), 72
- GTree (class in GTree), 96
- GTree (module), 96
- GTreeBase (class in GenomeBase), 75
- GTreeCrossoverSinglePoint() (in module Crossovers), 71
- GTreeCrossoverSinglePointStrict() (in module Crossovers), 71
- GTreeGP (class in GTree), 99
- GTreeGPCrossoverSinglePoint() (in module Crossovers), 71
- GTreeGPInitializer() (in module Initializers), 72
- GTreeGPMutatorOperation() (in module Mutators), 69
- GTreeGPMutatorSubtree() (in module Mutators), 69
- GTreeInitializerAllele() (in module Initializers), 72
- GTreeInitializerInteger() (in module Initializers), 72
- GTreeMutatorIntegerGaussian() (in module Mutators), 69
- GTreeMutatorIntegerRange() (in module Mutators), 69
- GTreeMutatorRealGaussian() (in module Mutators), 69
- GTreeMutatorRealRange() (in module Mutators), 69
- GTreeMutatorSwap() (in module Mutators), 69
- GTreeNode (class in GTree), 102
- GTreeNodeBase (class in GenomeBase), 76
- GTreeNodeGP (class in GTree), 104
- GUniformSelector() (in module Selectors), 73
- importSpecial() (in module Util), 38
- initializer (G1DBinaryString.G1DBinaryString attribute), 83
- initializer (G1DList.G1DList attribute), 91
- initializer (G2DBinaryString.G2DBinaryString attribute), 87
- initializer (G2DList.G2DList attribute), 95
- initializer (GenomeBase.GenomeBase attribute), 78
- initializer (GTree.GTree attribute), 98

- Initializers (module), 71
- initialize() (G1DBinaryString.G1DBinaryString method), 83
- initialize() (G1DList.G1DList method), 91
- initialize() (G2DBinaryString.G2DBinaryString method), 87
- initialize() (G2DList.G2DList method), 95
- initialize() (GenomeBase.GenomeBase method), 78
- initialize() (GPopulation.GPopulation method), 66
- initialize() (GSimpleGA.GSimpleGA method), 61
- initialize() (GTree.GTree method), 98
- initialize() (GTree.GTreeGP method), 101
- insert() (DBAdapters.DBBaseAdapter method), 47
- insert() (DBAdapters.DBFileCSV method), 48
- insert() (DBAdapters.DBMySQLAdapter method), 50
- insert() (DBAdapters.DBSQLite method), 52
- insert() (DBAdapters.DBURLPost method), 53
- insert() (DBAdapters.DBVPythonGraph method), 54
- insert() (DBAdapters.DBXMLRPC method), 55
- Interaction (module), 46
- Interactive mode, 12
- isEmpty() (FunctionSlot.FunctionSlot method), 57
- isLeaf() (GenomeBase.GTreeNodeBase method), 77
- isLeaf() (GTree.GTreeNode method), 103
- isLeaf() (GTree.GTreeNodeGP method), 105
- isReady() (Migration.MigrationScheme method), 43
- isReady() (Migration.WANMigration method), 45
- isReady() (Network.UDPThreadServer method), 41
- isReady() (Network.UDPThreadUnicastClient method), 41
- items() (Statistics.Statistics method), 58

- key_fitness_score() (in module GPopulation), 67
- key_fitness_score() (in module Selectors), 73
- key_raw_score() (in module GPopulation), 67
- key_raw_score() (in module Selectors), 73

- LinearScaling() (in module Scaling), 73
- list2DSwapElement() (in module Util), 38
- listSwapElement() (in module Util), 38

- makeDisplay() (DBAdapters.DBVPythonGraph method), 54
- Migration (module), 42
- MigrationScheme (class in Migration), 43
- minimaxType (in module Consts), 31
- multiprocessing_eval() (in module GPopulation), 67
- multiprocessing_eval_full() (in module GPopulation), 67
- mutate() (G1DBinaryString.G1DBinaryString method), 83
- mutate() (G1DList.G1DList method), 91
- mutate() (G2DBinaryString.G2DBinaryString method), 87
- mutate() (G2DList.G2DList method), 95
- mutate() (GenomeBase.GenomeBase method), 78
- mutate() (GTree.GTree method), 98
- mutate() (GTree.GTreeGP method), 101
- mutator (G1DBinaryString.G1DBinaryString attribute), 84
- mutator (G1DList.G1DList attribute), 91
- mutator (G2DBinaryString.G2DBinaryString attribute), 87
- mutator (G2DList.G2DList attribute), 95
- mutator (GenomeBase.GenomeBase attribute), 78
- mutator (GTree.GTree attribute), 98
- Mutators (module), 68

- Network (module), 39
- newNode() (GTree.GTreeNode method), 103
- newNode() (GTree.GTreeNodeGP method), 105
- nodeType (in module Consts), 31
- Non-terminal node, 13

- open() (DBAdapters.DBBaseAdapter method), 48
- open() (DBAdapters.DBFileCSV method), 49
- open() (DBAdapters.DBMySQLAdapter method), 50
- open() (DBAdapters.DBSQLite method), 52
- open() (DBAdapters.DBURLPost method), 53
- open() (DBAdapters.DBVPythonGraph method), 54
- open() (DBAdapters.DBXMLRPC method), 56

- pickleAndCompress() (in module Network), 42
- plotHistPopScore() (in module Interaction), 46
- plotPopScore() (in module Interaction), 47
- poolLength() (Network.UDPThreadServer method), 41
- poolLength() (Network.UDPThreadUnicastClient method), 42
- popPool() (Network.UDPThreadServer method), 41
- popPool() (Network.UDPThreadUnicastClient method), 42
- PowerLawScaling() (in module Scaling), 73
- printStats() (GPopulation.GPopulation method), 66
- printStats() (GSimpleGA.GSimpleGA method), 61
- printTimeElapsed() (GSimpleGA.GSimpleGA method), 61
- processNodes() (GenomeBase.GTreeBase method), 76
- processNodes() (GTree.GTree method), 98
- processNodes() (GTree.GTreeGP method), 101
- pyevolve (module), 30

- raiseException() (in module Util), 39
- rand_random() (in module Crossovers), 71
- rand_random() (in module Util), 39
- randomFlipCoin() (in module Util), 39
- Raw score, 12
- RawScoreCriteria() (in module GSimpleGA), 64
- RawStatsCriteria() (in module GSimpleGA), 64
- remove() (G1DBinaryString.G1DBinaryString method), 84
- remove() (G1DList.G1DList method), 91

- remove() (GAllele.GAlleleList method), 79
- remove() (GenomeBase.G1DBase method), 74
- replaceChild() (GenomeBase.GTreeNodeBase method), 77
- replaceChild() (GTree.GTreeNode method), 103
- replaceChild() (GTree.GTreeNodeGP method), 105
- reset() (Util.ErrorAccumulator method), 36
- reset() (Util.Graph method), 37
- resetStats() (G1DBinaryString.G1DBinaryString method), 84
- resetStats() (G1DList.G1DList method), 91
- resetStats() (G2DBinaryString.G2DBinaryString method), 87
- resetStats() (G2DList.G2DList method), 95
- resetStats() (GenomeBase.GenomeBase method), 78
- resetStats() (GTree.GTree method), 98
- resetStats() (GTree.GTreeGP method), 101
- resetStructure() (DBAdapters.DBMySQLAdapter method), 50
- resetStructure() (DBAdapters.DBSQLite method), 52
- resetTableIdentify() (DBAdapters.DBMySQLAdapter method), 50
- resetTableIdentify() (DBAdapters.DBSQLite method), 52
- resumeString() (G1DBinaryString.G1DBinaryString method), 84
- resumeString() (G1DList.G1DList method), 91
- resumeString() (G2DBinaryString.G2DBinaryString method), 87
- resumeString() (G2DList.G2DList method), 95
- resumeString() (GenomeBase.G1DBase method), 74
- run() (Network.UDPThreadBroadcastClient method), 40
- run() (Network.UDPThreadServer method), 41
- run() (Network.UDPThreadUnicastClient method), 42

- Sample genome, 12
- SaturatedScaling() (in module Scaling), 73
- scale() (GPopulation.GPopulation method), 66
- Scaling (module), 73
- select() (GSimpleGA.GSimpleGA method), 61
- select() (Migration.MigrationScheme method), 43
- select() (Migration.WANMigration method), 45
- selector (GSimpleGA.GSimpleGA attribute), 61
- selector (Migration.MigrationScheme attribute), 43
- selector (Migration.WANMigration attribute), 45
- Selectors (module), 72
- selectPool() (Migration.MigrationScheme method), 43
- selectPool() (Migration.WANMigration method), 45
- send() (Network.UDPThreadBroadcastClient method), 40
- send() (Network.UDPThreadUnicastClient method), 42
- set() (FunctionSlot.FunctionSlot method), 57
- setBufferSize() (Network.UDPThreadServer method), 41
- setCompressionLevel() (Migration.MigrationScheme method), 43
- setCompressionLevel() (Migration.WANMigration method), 45
- setCrossoverRate() (GSimpleGA.GSimpleGA method), 61
- setData() (GTree.GTreeNode method), 103
- setData() (GTree.GTreeNodeGP method), 105
- setData() (Network.UDPThreadBroadcastClient method), 40
- setDBAdapter() (GSimpleGA.GSimpleGA method), 61
- setElitism() (GSimpleGA.GSimpleGA method), 61
- setElitismReplacement() (GSimpleGA.GSimpleGA method), 61
- setGAEngine() (Migration.MigrationScheme method), 44
- setGAEngine() (Migration.WANMigration method), 45
- setGenerations() (GSimpleGA.GSimpleGA method), 62
- setGPMODE() (GSimpleGA.GSimpleGA method), 62
- setGroupName() (Migration.MigrationScheme method), 44
- setGroupName() (Migration.WANMigration method), 45
- setIdentify() (DBAdapters.DBBaseAdapter method), 48
- setIdentify() (DBAdapters.DBFileCSV method), 49
- setIdentify() (DBAdapters.DBMySQLAdapter method), 50
- setIdentify() (DBAdapters.DBSQLite method), 52
- setIdentify() (DBAdapters.DBURLPost method), 53
- setIdentify() (DBAdapters.DBVPythonGraph method), 54
- setIdentify() (DBAdapters.DBXMLRPC method), 56
- setInteractiveGeneration() (GSimpleGA.GSimpleGA method), 62
- setInteractiveMode() (GSimpleGA.GSimpleGA method), 62
- setInternalList() (G1DBinaryString.G1DBinaryString method), 84
- setInternalList() (G1DList.G1DList method), 91
- setInternalList() (GenomeBase.G1DBase method), 74
- setItem() (G2DBinaryString.G2DBinaryString method), 87
- setItem() (G2DList.G2DList method), 95
- setMigrationAdapter() (GSimpleGA.GSimpleGA method), 62
- setMigrationRate() (Migration.MigrationScheme method), 44
- setMigrationRate() (Migration.WANMigration method), 45
- setMinimax() (GPopulation.GPopulation method), 66
- setMinimax() (GSimpleGA.GSimpleGA method), 62
- setMultipleTargetHost() (Network.UDPThreadUnicastClient method), 42
- setMultiProcessing() (GPopulation.GPopulation method), 66
- setMultiProcessing() (GSimpleGA.GSimpleGA method), 62

- setMutationRate() (GSimpleGA.GSimpleGA method), 62
- setMyself() (Migration.MigrationScheme method), 44
- setMyself() (Migration.WANMigration method), 46
- setNumIndividuals() (Migration.MigrationScheme method), 44
- setNumIndividuals() (Migration.WANMigration method), 46
- setNumReplacement() (Migration.MigrationScheme method), 44
- setNumReplacement() (Migration.WANMigration method), 46
- setParams() (G1DBinaryString.G1DBinaryString method), 84
- setParams() (G1DList.G1DList method), 91
- setParams() (G2DBinaryString.G2DBinaryString method), 87
- setParams() (G2DList.G2DList method), 95
- setParams() (GenomeBase.GenomeBase method), 78
- setParams() (GPopulation.GPopulation method), 67
- setParams() (GSimpleGA.GSimpleGA method), 62
- setParams() (GTree.GTree method), 98
- setParams() (GTree.GTreeGP method), 101
- setParent() (GenomeBase.GTreeNodeBase method), 77
- setParent() (GTree.GTreeNode method), 103
- setParent() (GTree.GTreeNodeGP method), 105
- setPopulationSize() (GPopulation.GPopulation method), 67
- setPopulationSize() (GSimpleGA.GSimpleGA method), 63
- setRandomApply() (FunctionSlot.FunctionSlot method), 57
- setReal() (GAllele.GAlleleRange method), 80
- setRoot() (GenomeBase.GTreeBase method), 76
- setRoot() (GTree.GTree method), 99
- setRoot() (GTree.GTreeGP method), 101
- setSortType() (GPopulation.GPopulation method), 67
- setSortType() (GSimpleGA.GSimpleGA method), 63
- setStatsGenFreq() (DBAdapters.DBBaseAdapter method), 48
- setStatsGenFreq() (DBAdapters.DBFileCSV method), 49
- setStatsGenFreq() (DBAdapters.DBMySQLAdapter method), 50
- setStatsGenFreq() (DBAdapters.DBSQLite method), 52
- setStatsGenFreq() (DBAdapters.DBURLPost method), 53
- setStatsGenFreq() (DBAdapters.DBVPythonGraph method), 54
- setStatsGenFreq() (DBAdapters.DBXMLRPC method), 56
- setTargetHost() (Network.UDPThreadUnicastClient method), 42
- setTopology() (Migration.WANMigration method), 46
- setType() (GTree.GTreeNodeGP method), 105
- shutdown() (Network.UDPThreadServer method), 41
- shutdown() (Network.UDPThreadUnicastClient method), 42
- SigmaTruncScaling() (in module Scaling), 73
- sort() (GPopulation.GPopulation method), 67
- sortType (in module Consts), 31
- Standardized Fitness, 13
- start() (Migration.MigrationScheme method), 44
- start() (Migration.WANMigration method), 46
- Statistics (class in Statistics), 57
- Statistics (module), 57
- statistics() (GPopulation.GPopulation method), 67
- Step callback function, 13
- step() (GSimpleGA.GSimpleGA method), 63
- stepCallback (GSimpleGA.GSimpleGA attribute), 63
- stop() (Migration.MigrationScheme method), 44
- stop() (Migration.WANMigration method), 46
- swapNodeData() (GTree.GTreeNode method), 104
- swapNodeData() (GTree.GTreeNodeGP method), 105
- terminationCriteria (GSimpleGA.GSimpleGA attribute), 63
- traversal() (GenomeBase.GTreeBase method), 76
- traversal() (GTree.GTree method), 99
- traversal() (GTree.GTreeGP method), 101
- UDPThreadBroadcastClient (class in Network), 39
- UDPThreadServer (class in Network), 40
- UDPThreadUnicastClient (class in Network), 41
- unpickleAndDecompress() (in module Network), 42
- Util (module), 35
- WANMigration (class in Migration), 44
- writeDotGraph() (GTree.GTreeGP method), 101
- writeDotImage() (GTree.GTreeGP method), 102
- writeDotRaw() (GTree.GTreeGP method), 102
- writePopulationDot() (GTree.GTreeGP static method), 102
- writePopulationDotRaw() (GTree.GTreeGP static method), 102