The Extended Selfish Gene

Release 1.0

Giovanni Squillero, Alberto Tonda, Stella Stakiadi

Jan 29, 2021

CONTENTS

1 What is SGX? 3

2 Audience 5

3 Installation 7
3.1 Source Code 7
3.2 Importance of FOSS 7

4 Fitness Function 9
4.1 How do we handle a different scenario 9
4.2 Multi-Objective Evolutionary Algorithm 9

5 Allele 11
5.1 Base Allele class 11
5.2 Boolean Allele class 12
5.3 Categorical Allele class 12

6 Fitness 15
6.1 Base fitness class 15
6.2 Fitness Function class 16
6.3 Multi-Objective class 16
6.4 Simple class 17

7 Utils 19
7.1 CPU_time 19
7.2 Jupyter Support 19
7.3 Logging 19
7.4 Random class 19
7.5 Archive class 19
7.6 Base class 20
7.7 Species class 21

8 SGX Algorithm 23
8.1 Simple Algorithm class 23

9 Modular Design Rationale 25

10 Authors 27
10.1 Giovanni Squillero 27
10.2 Alberto Tonda 27
10.3 Stella Stakiadi 27

11 License 29

12 Acknowledgements 31

13 Appendix - Terms 33

Python Module Index 35

Index 37

Welcome to Extended Selfish Gene's documentation!

CHAPTER

ONE

WHAT IS SGX?

The Selfish Gene optimization algorithm (SG) is a population-less evolutionary algorithm loosely inspired by the interpretation of the Darwinian theory given by the English biologist Richard Dawkins and popularized as the Selfish Gene theory. It enables a user to efficiently find the list parameters, either discrete symbols or real numbers, that maximizes a given target function.

The original SG was an almost straightforward implementation of a though experiment, only able to handle binary values; it was published in SAC98 as "The selfish gene algorithm: a new evolutionary optimization strategy" (F. Corno, M.S. Reorda, G. Squillero, 1998) and few month later, with some modifications, in ICEC98 as "A new evolutionary algorithm inspired by the selfish gene theory" (F. Corno, M.S. Reorda, G. Squillero, 1998). The base algorithm was later discovered surprisingly similar to the Equilibrium Genetic Algorithm, developed by Ari Juels, Shumeet Baluja, and Alistair Sinclair in 1993 and never published – see "Lost gems of EC: the equilibrium genetic algorithm and the role of crossover" (Fernando G. Lobo, 2007). Even more surprisingly, Georges Harik, Fernando Lobo, and David Golberg proposed a quite similar, yet completely unrelated, algorithm in the very same ICEC98: "The Compact Genetic Algorithm" (G.R. Harik, F.G. Lobo, D.E. Goldberg, 1998). Comprehensive background information on "Estimation of Distribution Algorithms (EDAs) (Martin Pelikan, Mark W. Hauschild, Fernando G. Lobo, 2015) can be found in an introduction by Lobo et al.

Since its appearance, the SG was demonstrated more robust than pure hill climbing, reasonably efficient, and quite easy to implement. It was immediately exploited by practitioners in many real-world applications, CAD problems; and by scholars for various test benches. Moreover, the SG framework enabled the inclusions of tricks that made it effective in quite a wider range of situations. The enhanced SG-Clans added to the basic SG a sort of recursive evolution, inspired by the concept of allopatric speciation, to escape local optima. Results were published in "Optimizing deceptive functions with the SG-Clans algorithm" (F. Corno, M.S. Reorda, G. Squillero, 1999). Non-binary encodings were eventually added in 2000s. Indeed, real-valued parameters was never included as they never worked properly, although a draft paper titled "A population-less evolutionary algorithm for real and integer optimization" mysteriously crawled its way up to semantic scholar.

Over the years, the algorithm was reimplemented by different researchers in different languages, and a few brand new approaches derived from it (see Google Scholar's up-to-date references). In 2016, a comprehensive review was published on IOPScience "Selfish Gene Algorithm Vs Genetic Algorithm: A Review" (Ariff, Norharyati Md, Khalid, Noor Elaiza Abdul, Hashim, Rathiah, Noor, Noorhayati Mohamed, 2016).

CHAPTER TWO

AUDIENCE

The expected audience for SGX, a *'Quick n' Dirty'* numerical optimization, includes computer scientists, engineers and practitioners.

- This evolutionary algorithm provides a Sub-Optimal result, which is better than a *"hill-climbing"* algorithm.
- It is a **real, industrial application** where fitness function is computationally intensive.
- **Real-time** application.
- SGX also provides an easy-to-use and **standard interface**.
- The code is **parallelizable**, which means that it can run in parallel multiple threads. There is no need to wait for everything to be completed. Some implementations might be Embarrassingly parallel (see https://en.wikipedia.org/wiki/Embarrassingly_parallel).
- SGX is available as a PyPi package and it can be easily installed using pip.
- The modular design allows scholars to extend SGX for custom application.

CHAPTER

THREE

INSTALLATION

3.1 Source Code

SGX is available as a PyPi package from https://pypi.org/project/sgx/ and installing it is as simple as

```
pip install sgx
```

and then

```
>>> import sgx
```

Caveat: on some systems the package manager is `pip3`.

3.2 Importance of FOSS

Personal control, customizability and freedom:

Users of FOSS benefit from the Four Essential Freedoms to make unrestricted use of, and to study, copy, modify, and redistribute such software with or without modification. If they would like to change the functionality of software they can bring about changes to the code and, if they wish, distribute such modified versions of the software or often depending on the software's decision making model and its other users even push or request such changes to be made via updates to the original software.

Privacy and security:

Manufacturers of proprietary, closed-source software are sometimes pressured to building in backdoors or other covert, undesired features into their software. Instead of having to trust software vendors, users of FOSS can inspect and verify the source code themselves and can put trust on a community of volunteers and users. As proprietary code is typically hidden from public view, only the vendors themselves and hackers may be aware of any vulnerabilities in them while FOSS involves as many people as possible for exposing bugs quickly.

Low costs or no costs:

FOSS is often free of charge although donations are often encouraged. This also allows users to better test and compare software.

Quality, collaboration and efficiency:

FOSS allows for better collaboration among various parties and individuals with the goal of developing the most efficient software for its users or use-cases while proprietary software is typically meant to generate profits. Furthermore, in many cases more organizations and individuals contribute to such projects than to proprietary software. It has been shown that technical superiority is typically the primary reason why companies choose open source software.

The default branch is always the more stable and the only one tested through Travis CI. The experimental branches `exp/*` contain code and comments that some programmers may find disturbing — Viewers discretion advised. Before trying to contribute read this paper and this style guide. It may be wise to send Giovanni an email [@] before digging into the project.

CHAPTER

FOUR

FITNESS FUNCTION

Fitness class also redefines the relational operator in order to handle different types of optimization (eg. maximization, minimization) and to provide limited support to more complex scenarios (eg. multi-objective optimization).

4.1 How do we handle a different scenario

When subclassing, one's fitness should only redefine 'is_fitter', and optionally 'is_distinguishable' and 'is_dominant'; 'is_dominant' must be changed if 'is_fitter' is randomized (the result is uncertain).

The idea of several, different scenarios is the following:

`a == b` In this case, fitness `a` cannot be distinguished from fitness `b`.

`a != b` In this case, fitness `a` is distinguishable from fitness `b`.

`a > b` In this case, fitness `a` is fitter than fitness `b`. (may not always be the case, see lexicographic)

`a >= b` In this case, fitness `a` is fitter or not distinguishable from fitness `b`.

`a < b` In this case, fitness `b` is fitter than fitness `a`, respectively. (may not always be the case, see lexicographic)

`a <= b` In this case, fitness `b` is fitter or not distinguishable from fitness `a`, respectively.

`a >> b` In this case, fitness `a` dominates fitness `b` which is a certain case.

`a << b` In this case, fitness `a` is dominated by fitness `b`, accordingly.

4.2 Multi-Objective Evolutionary Algorithm

The problem becomes more interesting in case that there exist more than one characteristic that should be compared in order to decide which individual is "better". It is very important to find rules that describe characteristics with respect to a property of interest. The MOEA approach, a method of combining the traditional genetic algorithm (TGA) with the multi-objective method, can consider the relation between the parameters and the objective spaces in the same time then explore the optimum solution. Our multi-objective evolutionary algorithm uses a 'helper' function which can decide the best individual when there are two comparable characteristics.

This special Multi-Objective scenario can be illustrated with an airplaine ticket purchase. Let us consider the example of buying a flight ticket where the price of ticket and travel time are the decision-making criteria.

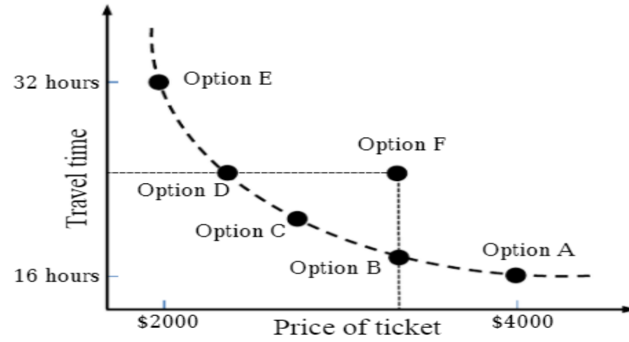

Figure 1. Illustration of decision-making process in buying a flight ticket

The points A,B,C,D,E and F represent the options for flying between two cities. We assume that difference in travel time is due to the waiting time for connecting flight at transit. *Option A* is the most expensive with ticket price of $4000, but with least travel time of 16 hours. The cheapest ticket is of $2000 with travel time of 32 hours if one takes flying *Option E*. Here the decision-making process of flight booking is not a single objective of either price or travel time. The traveler has few options to choose from with some trade-off between travel time and price. If one selects *Option B* instead of *Option A*, he will be saving on his ticket price by spending more time on transit. Again, if the traveler selects *Option D* instead of *Option E*, he has to sell out more money for buying the ticket, but he can save a few hours. If someone chooses *Option F*, he is definitely losing. He can go for *Option B* at same price with less travel time, or *Option D* of same travel duration at lower price. The points A,B,C,D and E are called *Pareto Optimal Points*, named after the famous Italian economist Vilfredo Pareto. They are also called **Non-Dominated** solutions. The example of flight options is for a **two-objective optimization**. In multi-objective optimization there can be more than two objectives.

CHAPTER

FIVE

ALLELE

5.1 Base Allele class

sgx.allele.base

class sgx.allele.base.**Allele**

Abstract class for Allele.

An allele must be Hashable (ie. non modifiable)

abstract describe() → str

Pretty describes the current allele.

property mode

Returns the most frequent allele.

property possible_values

Possible values of the allele. None if not reasonably applicable (eg. a float)

abstract sample(*sample_type: Optional[str] = 'sample'*) → Hashable

Sample.

Parameters sample_type – 'sample' (default): random value according to the current probability distribution 'uniform': random value according to a uniform probability distribution (ie. completely random) 'mode': most common value according to the current probability distribution

abstract update(*winner: Hashable*, *loser: Hashable*) → None

Updates the winner and the loser Genotype, so as to modify the new Learning Rate.

Parameters

- **winner** – The genotype of the better solution.
- **loser** – The genotype of the worse solution.

5.2 Boolean Allele class

sgx.allele.boolean

class sgx.allele.boolean.**Boolean**(*learning_rate: float = 0.001*)

describe() → str
Pretty describes the current boolean allele.

is_valid(*value: Hashable*) → bool
Checks if the allele is boolean.

run_paranoia_checks() → bool
Returns True, if all tests are passed.

sample(*sample_type: Optional[str] = 'sample'*) → Hashable
Sample.

Parameters **sample_type** – The type of sample (sample (default), uniform, mode).

static sigmoid(*x: float*, *k: Optional[float] = 1*) → float
Logistic function with given logistic growth (k). See https://en.wikipedia.org/wiki/Logistic_function

Parameters

- **x** – A float number.
- **k** – Logistic Growth.

Returns The result probability of the sigmoid function.

update(*winner: Hashable*, *loser: Hashable*) → None
Updates the winner and the loser Genotype, so as to modify the new Learning Rate.

Parameters

- **winner** – The genotype of the better solution.
- **loser** – The genotype of the worse solution.

5.3 Categorical Allele class

sgx.allele.categorical

class sgx.allele.categorical.**Categorical**(*alternatives: Sequence[Hashable]*, *weights: Optional[Union[Sequence[float], dict]] = None*, *learning_rate: Optional[float] = None*)

describe() → str
Pretty describes the current categorical allele.

is_valid(*value: Hashable*) → bool
Checks if the allele is categorical.

property possible_values
Possible values of the allele. None if not reasonably applicable (eg. a float)

run_paranoia_checks() → bool
Returns True, if all tests are passed.

sample (*sample_type: Optional[str] = 'sample'*) → Hashable
Sample.

Parameters `sample_type` – The type of sample (sample (default), uniform, mode).

update (*winner: Hashable*, *loser: Hashable*) → None
Updates the winner and the loser Genotype, so as to modify the new Learning Rate.

Parameters

- `winner` – The genotype of the better solution.
- `loser` – The genotype of the worse solution.

CHAPTER

SIX

FITNESS

6.1 Base fitness class

sgx.fitness.base

class sgx.fitness.base.**Fitness**

Fitness of a phenotype, handle multiple formats (eg. scalar, tuple).

The class also redefines the relational operator in order to handle different types of optimization (eg. maximization, minimization) and to provide limited support to more complex scenarios (eg. multi-objective optimization)

Equalities ('==' and '!=') are based on *is_distinguishable*.

Single angular-bracket operators ('>', '<', '>=', and '<=') are based on *is_fitter* and may be randomized (the result may not be reproducible).

Double angular-bracket operators ('>>' and '<<') are based on *is_dominant* and the result is stable. By default *is_dominant* is defined as *is_fitter*.

When subclassing, one should only redefine *is_fitter*, and optionally *is_distinguishable* and *is_dominant*; *is_dominant* must be changed if *is_fitter* is randomized (the result is uncertain).

Additional sanity checks should be added to *check_comparable*. Subclasses may redefine the *decorate* method to change the values appearance.

__eq__(*other:* sgx.fitness.base.Fitness) → bool

Returns True if one fitness is equal (==) to the other.

__ge__(*other:* sgx.fitness.base.Fitness) → bool

Returns True if one fitness is greater or equal (>=) to the other.

__gt__(*other:* sgx.fitness.base.Fitness) → bool

Returns True if one fitness is greater (>) than the other.

__le__(*other:* sgx.fitness.base.Fitness) → bool

Returns True if one fitness is less or equal (<=) to the other.

__lshift__(*other:* sgx.fitness.base.Fitness) → bool

Returns True if one fitness does not dominates (<<) the other.

__lt__(*other:* sgx.fitness.base.Fitness) → bool

Returns True if one fitness is less (<) than the other.

__ne__(*other:* sgx.fitness.base.Fitness) → bool

Returns True if one fitness is not equal (!=) to the other.

__rshift__(*other:* sgx.fitness.base.Fitness) → bool

Returns True if one fitness dominates (>>) the other.

check_comparable(*other: sgx.fitness.base.Fitness*)
Checks if the fitness is able to be compared.

decorate() → str
Represent the individual fitness value with a nice string.

is_distinguishable(*other: sgx.fitness.base.Fitness*) → bool
Check whether some differences from the other Fitness may be perceived.

is_dominant(*other: sgx.fitness.base.Fitness*) → bool
Check whether dominates the other (result is certain).

is_fitter(*other: sgx.fitness.base.Fitness*) → bool
Check whether fitter than the other (result may be accidental).

is_valid(*fitness: sgx.fitness.base.Fitness*) → bool
Returns True if the fitness is able to be compared, otherwise Raises an Assertion Error.

run_paranoia_checks() → bool
Returns True if checks are successful.

sgx.fitness.base.**reversed**(*fitness_class: sgx.fitness.base.Fitness*) → *sgx.fitness.base.Fitness*
Reverse fitness class turning a maximization problem into a minimization one.

6.2 Fitness Function class

sgx.fitness.function

6.3 Multi-Objective class

sgx.fitness.multi_objective

class sgx.fitness.multi_objective.**Lexicase**(*value: Sequence, fitness_type: Type[sgx.fitness.base.Fitness] = <class 'sgx.fitness.simple.Scalar'>, **kwargs*)
Pseudo-MO through Lexicase selection (DOI:10.1109/TEVC.2014.2362729).

is_fitter(*other: sgx.fitness.multi_objective.Lexicase*) → bool
Check whether fitter than the other.

class sgx.fitness.multi_objective.**MultiObjective**(*value: Sequence, fitness_type: Type[sgx.fitness.base.Fitness] = <class 'sgx.fitness.simple.Scalar'>, **kwargs*)
Abstract class for handling Multi-Objective problems.

is_dominant(*other: sgx.fitness.multi_objective.Lexicase*) → bool
Check whether dominates the other (result is certain).

abstract is_fitter(*other: sgx.fitness.multi_objective.Lexicase*) → bool
Check whether fitter than the other.

6.4 Simple class

sgx.fitness.simple

class sgx.fitness.simple.**Approximate**(*argument*, *rel_tol: float = 1e-09*, *abs_tol: float = 0*)
A single, floating-point value with approximate equality – Larger is better.

check_comparable(*other:* sgx.fitness.simple.Approximate)
Checks if the fitness is able to be compared.

decorate() → str
Represent the individual fitness value with a nice string.

is_distinguishable(*other:* sgx.fitness.base.Fitness) → bool
Check whether some differences from the other Fitness may be perceived.

is_fitter(*other:* sgx.fitness.base.Fitness) → bool
Check whether fitter than the other.

class sgx.fitness.simple.**Integer**
A single numeric value – Larger is better.

class sgx.fitness.simple.**Scalar**(*x=0*, */*)
A single numeric value – Larger is better.

class sgx.fitness.simple.**Vector**(*value: Sequence*, *fitness_type: Type[sgx.fitness.base.Fitness] = <class 'sgx.fitness.simple.Scalar'>*, ***kwargs*)
A generic vector of Fitness values.

fitness_type is the subtype, **kwargs are passed to fitness init

Examples

f1 = sgx.fitness.Vector([23, 10], fitness_type=Approximate, abs_tol=.1) f2 = sgx.fitness.Vector([23, 10], fitness_type=Approximate, abs_tol=.001)

f1 > sgx.fitness.Vector([23, 9.99], fitness_type=Approximate, abs_tol=.1) is False f2 > sgx.fitness.Vector([23, 9.99], fitness_type=Approximate, abs_tol=.001) is True

check_comparable(*other:* sgx.fitness.simple.Vector)
Checks if the fitness is able to be compared.

static compare_vectors(*v1: Sequence[*sgx.fitness.base.Fitness*]*, *v2: Sequence[*sgx.fitness.base.Fitness*]*) → int
Compare Fitness values in v1 and v2.

Parameters

- **v1** – The first fitness vector.
- **v2** – The second fitness vector.

Returns -1 if v1 < v2; +1 if v1 > v2; 0 if v1 == v2

decorate() → str
Represent the individual fitness value with a nice string.

is_distinguishable(*other:* sgx.fitness.simple.Vector) → bool
Check whether some differences from the other Fitness may be perceived.

is_fitter(*other:* sgx.fitness.base.Fitness) → bool
Check whether fitter than the other.

CHAPTER

SEVEN

UTILS

7.1 CPU_time

sgx.utils.cpu_time

7.2 Jupyter Support

sgx.utils.jupyter_support

`sgx.utils.jupyter_support.`**`is_notebook`**`()` → bool

Check if running inside a notebooks

Credits: https://stackoverflow.com/questions/15411967/

7.3 Logging

sgx.utils.logging

`sgx.utils.logging.`**`log_cpu`**`(`*level: int = 20, msg: str = '', *args, **kwargs*`)` → None

Like log(), but including cpu time.

7.4 Random class

sgx.utils.random

7.5 Archive class

sgx.archive

7.6 Base class

sgx.base

class sgx.base.**Genome**(**args*)

A tuple of Alleles, each one specifying a set of alternative genes.

is_valid(*genotype: sgx.base.Genotype*) → bool

Check an object against a specification.

The function may be used to check a value against a parameter definition, a node against a section definition).

Returns True if the object is valid, False otherwise.

run_paranoia_checks() → bool

Check the internal consistency of a "paranoid" object.

The function should be overridden by the sub-classes to implement the required, specific checks. It always returns `True`, but throws an exception whenever an inconsistency is detected.

Notez bien: Sanity checks may be computationally intensive, paranoia checks are not supposed to be used in production environments (i.e., when `-O` is used for compiling). Their typical usage is: `assert foo.run_paranoia_checks()`

Returns True (always)

Raises **AssertionError if some internal data structure is incoherent** –

class sgx.base.**Genotype**(**args*)

A tuple containing the organism's actual genes (their values).

run_paranoia_checks() → bool

Check the internal consistency of a "paranoid" object.

The function should be overridden by the sub-classes to implement the required, specific checks. It always returns `True`, but throws an exception whenever an inconsistency is detected.

Notez bien: Sanity checks may be computationally intensive, paranoia checks are not supposed to be used in production environments (i.e., when `-O` is used for compiling). Their typical usage is: `assert foo.run_paranoia_checks()`

Returns True (always)

Raises **AssertionError if some internal data structure is incoherent** –

class sgx.base.**Paranoid**

Abstract class: Paranoid classes do implement *run_paranoia_checks()*.

run_paranoia_checks() → bool

Check the internal consistency of a "paranoid" object.

The function should be overridden by the sub-classes to implement the required, specific checks. It always returns `True`, but throws an exception whenever an inconsistency is detected.

Notez bien: Sanity checks may be computationally intensive, paranoia checks are not supposed to be used in production environments (i.e., when `-O` is used for compiling). Their typical usage is: `assert foo.run_paranoia_checks()`

Returns True (always)

Raises **AssertionError if some internal data structure is incoherent** –

class sgx.base.**Pedantic**

Abstract class: Pedantic classes do implement *is_valid()*.

abstract is_valid(*obj: Any*) → bool

Check an object against a specification.

The function may be used to check a value against a parameter definition, a node against a section definition).

Returns True if the object is valid, False otherwise.

7.7 Species class

sgx.species

class sgx.species.**Species**(*genome: Sequence[Any], fitness_function: sgx.fitness.function.FitnessFunction, mutation_rate: Optional[float] = None*)

Missing

evaluate(*genotype:* sgx.base.Genotype) → *sgx.fitness.base.Fitness*

Evaluates a genotype according to a specific Fitness Function.

Parameters **genotype** – The Genotype which is going to be evaluated.

Returns The fitness calculated from a Fitness Function.

sample(*sample_type: Optional[str] = 'sample'*) → *sgx.base.Genotype*

Sampling the genotypes with a specific method.

Parameters **sample_type** – The type of sampling is going to be used.

Returns The candidate's Genotype after sampling.

update(*winner:* sgx.base.Genotype, *loser:* sgx.base.Genotype)

Updates the winner and the loser Genotype, so as to modify the new Learning Rate.

Parameters

- **winner** – The genotype of the better solution.
- **loser** – The genotype of the worse solution.

CHAPTER

EIGHT

SGX ALGORITHM

8.1 Simple Algorithm class

`sgx.algorithms.simple`

CHAPTER

NINE

MODULAR DESIGN RATIONALE

. . . ?

CHAPTER

TEN

AUTHORS

10.1 Giovanni Squillero

Politecnico di Torino
Department of Control and Computer Engineering
Corso Duca degli Abruzzi 24
10129 Torino — Italy
E-mail: giovanni.squillero@polito.it

10.2 Alberto Tonda

Génie et Microbiologie des Procédés Alimentaires (GMPA)
French National Institute for Agricultural Research
AgroParisTech, Université Paris - Saclay
1 av. Brétignières
78850 Thiverval-Grignon — France
E-mail: alberto.tonda@inrae.fr

10.3 Stella Stakiadi

Politecnico di Torino
Department of Control and Computer Engineering
Corso Duca degli Abruzzi 24
10129 Torino — Italy
E-mail: stellastak@gmail.com

CHAPTER

ELEVEN

LICENSE

Copyright © 2021 Giovanni Squillero

The Extended Selfish Gene (SGX) is free and open-source software , and it is distributed under the permissive Apache License 2.0.

CHAPTER

TWELVE

ACKNOWLEDGEMENTS

This Documentation was built with Sphinx using a theme provided by Read the Docs.

CHAPTER THIRTEEN

APPENDIX - TERMS

`**Allele**`: An allele is one of two, or more, forms of a given gene variant. An allele is one of two, or more, versions of the same gene at the same place on a chromosome.

`**Locus**`: A locus (plural loci) is a specific, fixed position on a chromosome where a particular gene or genetic marker is located. Each chromosome carries many genes, with each gene occupying a different position or locus;

`**Gene**`: A gene is a basic unit of heredity and a sequence of nucleotides in DNA or RNA that encodes the synthesis of a gene product, either RNA or protein.

`**Genome**`: A genome is all genetic material of an organism. It consists of DNA (or RNA in RNA viruses). The genome includes both the genes (the coding regions) and the noncoding DNA, as well as mitochondrial DNA and chloroplast DNA.

`**Genotype**`: A genotype is an organism's complete set of genetic material. Often though, genotype is used to refer to a single gene or set of genes, such as the genotype for eye color. The genes take part in determining the characteristics that are observable (phenotype) in an organism, such as hair color, height, etc.

`**Phenotype**`: Phenotype is the term used in genetics for the composite observable characteristics or traits of an organism. The term covers the organism's morphology or physical form and structure, its developmental processes, its biochemical and physiological properties, its behavior, and the products of behavior.

`**Individual**`: An individual is that which exists as a distinct entity (see Population).

`**Population**`: A population is defined as a group of individuals of the same species living and interbreeding within a given area.

PYTHON MODULE INDEX

S

sgx.allele.base, 11
sgx.allele.boolean, 12
sgx.allele.categorical, 12
sgx.base, 20
sgx.fitness.base, 15
sgx.fitness.multi_objective, 16
sgx.fitness.simple, 17
sgx.species, 21
sgx.utils.cpu_time, 19
sgx.utils.jupyter_support, 19
sgx.utils.logging, 19
sgx.utils.random, 19

INDEX

Symbols

__eq__() (*sgx.fitness.base.Fitness method*), 15
__ge__() (*sgx.fitness.base.Fitness method*), 15
__gt__() (*sgx.fitness.base.Fitness method*), 15
__le__() (*sgx.fitness.base.Fitness method*), 15
__lshift__() (*sgx.fitness.base.Fitness method*), 15
__lt__() (*sgx.fitness.base.Fitness method*), 15
__ne__() (*sgx.fitness.base.Fitness method*), 15
__rshift__() (*sgx.fitness.base.Fitness method*), 15

A

Allele (*class in sgx.allele.base*), 11
Approximate (*class in sgx.fitness.simple*), 17

B

Boolean (*class in sgx.allele.boolean*), 12

C

Categorical (*class in sgx.allele.categorical*), 12
check_comparable() (*sgx.fitness.base.Fitness method*), 15
check_comparable() (*sgx.fitness.simple.Approximate method*), 17
check_comparable() (*sgx.fitness.simple.Vector method*), 17
compare_vectors() (*sgx.fitness.simple.Vector static method*), 17

D

decorate() (*sgx.fitness.base.Fitness method*), 16
decorate() (*sgx.fitness.simple.Approximate method*), 17
decorate() (*sgx.fitness.simple.Vector method*), 17
describe() (*sgx.allele.base.Allele method*), 11
describe() (*sgx.allele.boolean.Boolean method*), 12
describe() (*sgx.allele.categorical.Categorical method*), 12

E

evaluate() (*sgx.species.Species method*), 21

F

Fitness (*class in sgx.fitness.base*), 15

G

Genome (*class in sgx.base*), 20
Genotype (*class in sgx.base*), 20

I

Integer (*class in sgx.fitness.simple*), 17
is_distinguishable() (*sgx.fitness.base.Fitness method*), 16
is_distinguishable() (*sgx.fitness.simple.Approximate method*), 17
is_distinguishable() (*sgx.fitness.simple.Vector method*), 17
is_dominant() (*sgx.fitness.base.Fitness method*), 16
is_dominant() (*sgx.fitness.multi_objective.MultiObjective method*), 16
is_fitter() (*sgx.fitness.base.Fitness method*), 16
is_fitter() (*sgx.fitness.multi_objective.Lexicase method*), 16
is_fitter() (*sgx.fitness.multi_objective.MultiObjective method*), 16
is_fitter() (*sgx.fitness.simple.Approximate method*), 17
is_fitter() (*sgx.fitness.simple.Vector method*), 17
is_notebook() (*in module sgx.utils.jupyter_support*), 19
is_valid() (*sgx.allele.boolean.Boolean method*), 12
is_valid() (*sgx.allele.categorical.Categorical method*), 12
is_valid() (*sgx.base.Genome method*), 20
is_valid() (*sgx.base.Pedantic method*), 21
is_valid() (*sgx.fitness.base.Fitness method*), 16

L

Lexicase (*class in sgx.fitness.multi_objective*), 16
log_cpu() (*in module sgx.utils.logging*), 19

M

mode() (*sgx.allele.base.Allele property*), 11

module
  sgx.allele.base, 11
  sgx.allele.boolean, 12
  sgx.allele.categorical, 12
  sgx.base, 20
  sgx.fitness.base, 15
  sgx.fitness.multi_objective, 16
  sgx.fitness.simple, 17
  sgx.species, 21
  sgx.utils.cpu_time, 19
  sgx.utils.jupyter_support, 19
  sgx.utils.logging, 19
  sgx.utils.random, 19
MultiObjective (*class in sgx.fitness.multi_objective*), 16

P

Paranoid (*class in sgx.base*), 20
Pedantic (*class in sgx.base*), 21
possible_values() (*sgx.allele.base.Allele property*), 11
possible_values() (*sgx.allele.categorical.Categorical property*), 12

R

reversed() (*in module sgx.fitness.base*), 16
run_paranoia_checks() (*sgx.allele.boolean.Boolean method*), 12
run_paranoia_checks() (*sgx.allele.categorical.Categorical method*), 12
run_paranoia_checks() (*sgx.base.Genome method*), 20
run_paranoia_checks() (*sgx.base.Genotype method*), 20
run_paranoia_checks() (*sgx.base.Paranoid method*), 20
run_paranoia_checks() (*sgx.fitness.base.Fitness method*), 16

S

sample() (*sgx.allele.base.Allele method*), 11
sample() (*sgx.allele.boolean.Boolean method*), 12
sample() (*sgx.allele.categorical.Categorical method*), 12
sample() (*sgx.species.Species method*), 21
Scalar (*class in sgx.fitness.simple*), 17
sgx.allele.base
  module, 11
sgx.allele.boolean
  module, 12
sgx.allele.categorical
  module, 12
sgx.base
  module, 20
sgx.fitness.base
  module, 15
sgx.fitness.multi_objective
  module, 16
sgx.fitness.simple
  module, 17
sgx.species
  module, 21
sgx.utils.cpu_time
  module, 19
sgx.utils.jupyter_support
  module, 19
sgx.utils.logging
  module, 19
sgx.utils.random
  module, 19
sigmoid() (*sgx.allele.boolean.Boolean static method*), 12
Species (*class in sgx.species*), 21

U

update() (*sgx.allele.base.Allele method*), 11
update() (*sgx.allele.boolean.Boolean method*), 12
update() (*sgx.allele.categorical.Categorical method*), 13
update() (*sgx.species.Species method*), 21

V

Vector (*class in sgx.fitness.simple*), 17