# pyeasyga Documentation

***Release 0.3.1***

**Ayodeji Remi-Omosowon**

**May 05, 2019**

# Contents

*A simple and easy-to-use implementation of a Genetic Algorithm library in Python*

**Contents:**

# CHAPTER 1

# pyeasyga


## 1.1 Introduction

A simple and easy-to-use implementation of a Genetic Algorithm library in Python.

`pyeasyga` provides a simple interface to the power of Genetic Algorithms (GAs). You don't have to have expert GA knowledge in order to use it.

- Homepage: https://github.com/remiomosowon/pyeasyga
- PyPI: https://pypi.python.org/pypi/pyeasyga
- Documentation: http://pyeasyga.readthedocs.org.
- Issues / Feedback: https://github.com/remiomosowon/pyeasyga/issues
- Free software: BSD license

### 1.1.1 Installation

At the command line, simply run:

```
$ pip install pyeasyga
```

Or clone this repository and run `python setup.py install` from within the project directory. e.g.:

```
$ git clone https://github.com/remiomosowon/pyeasyga.git
$ cd pyeasyga
$ python setup.py install
```

For alternative install methods, see the INSTALL file or the Installation section in the documentation.

### 1.1.2 Examples

See the Usage section in the documentation for examples. The example files can be found in the `examples` directory.

## 1.2 Note

- Currently under active development

# CHAPTER 2

# Installation

If you have pip installed, at the command line simply run:

```
$ pip install pyeasyga
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv pyeasyga
$ pip install pyeasyga
```

Or, download and extract the compressed archive (or clone the repository) from github, and inside the directory run:

```
$ python setup.py install
```

CHAPTER 3

# Usage

To use pyeasyga in a project:

## 3.1 Simple

1. Import the module

```
from pyeasyga import pyeasyga
```

2. Setup your data e.g.

```
data = [('pear', 50), ('apple', 35), ('banana', 40)]
```

3. Initialise the GeneticAlgorithm class with the only required parameter: `data`

```
ga = pyeasyga.GeneticAlgorithm(data)
```

4. Define a fitness function for the Genetic Algorithm. The function should take two parameters: a candidate soultion (an individual in GA speak), and the data that is used to help determine the individual's fitness

```
def fitness (individual, data):
    fitness = 0
    if individual.count(1) == 2:
        for (selected, (fruit, profit)) in zip(individual, data):
            if selected:
                fitness += profit
    return fitness
```

5. Set the Genetic Algorithm's `fitness_function` attribute to your defined fitness function

```
ga.fitness_function = fitness
```

6. Run the Genetic Algorithm

```
ga.run()
```

7. Print the best solution

```
print ga.best_individual()
```

## 3.2 Advanced

1. Import the module

```
from pyeasyga import pyeasyga
```

2. Setup your data e.g.

```
data = [('pear', 50), ('apple', 35), ('banana', 40)]
```

3. Initialise the GeneticAlgorithm class with the required `data` parameter, and all or some of the optional parameters

```
ga = pyeasyga.GeneticAlgorithm(data,
                               population_size=10,
                               generations=20,
                               crossover_probability=0.8,
                               mutation_probability=0.05,
                               elitism=True,
                               maximise_fitness=True)
```

Or

```
ga = pyeasyga.GeneticAlgorithm(data, 10, 20, 0.8, 0.05, True, True)
```

Or, just initialise the GeneticAlgorithm class with only the required `data` parameter, if you are content with the default parameters

```
ga = pyeasyga.GeneticAlgorithm(data)
```

4. Optionally, define a function to create a representation of a candidate solution (an individual in GA speak). This function should take in the data defined in step 1. as a parameter.

```
def create_individual(data):
    return [random.randint(0, 1) for _ in xrange(len(data))]
```

5. Set the Genetic Algorithm's `create_individual` attribute to your defined function

```
ga.create_individual = create_individual
```

6. Optionally, define and set functions for the Genetic Algorithm's genetic operators (i.e. crossover, mutate, selection)

```
# For the crossover function, supply two individuals (i.e. candidate
# solution representations) as parameters,
def crossover(parent_1, parent_2):
    crossover_index = random.randrange(1, len(parent_1))
    child_1 = parent_1[:index] + parent_2[index:]
```

(continues on next page)

(continued from previous page)

```
    child_2 = parent_2[:index] + parent_1[index:]
    return child_1, child_2

# and set the Genetic Algorithm's ``crossover_function`` attribute to
# your defined function
ga.crossover_function = crossover


# For the mutate function, supply one individual (i.e. a candidate
# solution representation) as a parameter,
def mutate(individual):
    mutate_index = random.randrange(len(individual))
    if individual[mutate_index] == 0:
        individual[mutate_index] = 1
    else:
        individual[mutate_index] = 0

# and set the Genetic Algorithm's ``mutate_function`` attribute to
# your defined function
ga.mutate_function = mutate


# For the selection function, supply a ``population`` parameter
def selection(population):
    return random.choice(population)

# and set the Genetic Algorithm's ``selection_function`` attribute to
# your defined function
ga.selection_function = selection
```

7. Define a fitness function for the Genetic Algorithm. The function should take two parameters: a candidate solution representation (an individual in GA speak), and the data that is used to help determine the individual's fitness

```
def fitness (individual, data):
    fitness = 0
    if individual.count(1) == 2:
        for (selected, (fruit, profit)) in zip(individual, data):
            if selected:
                fitness += profit
    return fitness
```

8. Set the Genetic Algorithm's `fitness_function` attribute to your defined fitness function

```
ga.fitness_function = fitness
```

9. Run the Genetic Algorithm

```
ga.run()
```

10. Print the best solution:

```
print ga.best_individual()
```

11. You can also examine all the individuals in the last generation:

```
for individual in ga.last_generation():
    print individual
```

## 3.3 Example of Simple Usage

This simple example uses the default `pyeasyga.GeneticAlgorithm` parameters.

The problem is to select only two items from a list (the supplied data) while maximising the cost of the selected items. *(Solution: Selecting the pear and apple gives the highest possible cost of 90.)*

```
>>> from pyeasyga.pyeasyga import GeneticAlgorithm
>>>
>>> data = [('pear', 50), ('apple', 35), ('banana', 40)]
>>> ga = GeneticAlgorithm(data)
>>>
>>> def fitness (individual, data):
>>>     fitness = 0
>>>     if individual.count(1) == 2:
>>>         for (selected, (fruit, profit)) in zip(individual, data):
>>>             if selected:
>>>                 fitness += profit
>>>     return fitness
>>>
>>> ga.fitness_function = fitness
>>> ga.run()
>>> print ga.best_individual()
```

## 3.4 Example of Advanced Usage

This example uses both default and optional `pyeasyga.GeneticAlgorithm` parameters.

The problem is to select only two items from a list (the supplied data) while maximising the cost of the selected items. *(Solution: Selecting the pear and apple gives the highest possible cost of 90.)*

```
>>> from pyeasyga.pyeasyga import GeneticAlgorithm
>>>
>>> data = [('pear', 50), ('apple', 35), ('banana', 40)]
>>> ga = GeneticAlgorithm(data, 20, 50, 0.8, 0.2, True, True)
>>>
>>> def create_individual(data):
>>>     return [random.randint(0, 1) for _ in xrange(len(data))]
>>>
>>> ga.create_individual = create_individual
>>>
>>>
>>> def crossover(parent_1, parent_2):
>>>     crossover_index = random.randrange(1, len(parent_1))
>>>     child_1 = parent_1[:index] + parent_2[index:]
>>>     child_2 = parent_2[:index] + parent_1[index:]
>>>     return child_1, child_2
>>>
>>> ga.crossover_function = crossover
>>>
```

(continues on next page)

(continued from previous page)

```
>>>
>>> def mutate(individual):
>>>     mutate_index = random.randrange(len(individual))
>>>     if individual[mutate_index] == 0:
>>>         individual[mutate_index] = 1
>>>     else:
>>>         individual[mutate_index] = 0
>>>
>>> ga.mutate_function = mutate
>>>
>>>
>>> def selection(population):
>>>     return random.choice(population)
>>>
>>> ga.selection_function = selection
>>>
>>> def fitness (individual, data):
>>>     fitness = 0
>>>     if individual.count(1) == 2:
>>>         for (selected, (fruit, profit)) in zip(individual, data):
>>>             if selected:
>>>                 fitness += profit
>>>     return fitness
>>>
>>> ga.fitness_function = fitness
>>> ga.run()
>>> print ga.best_individual()
>>>
>>> for individual in ga.last_generation():
>>>     print individual
```

CHAPTER 4

# Examples

## 4.1 1-Dimensional Knapsack Problem

one_dimensional_knapsack.py

This example solves the one-dimensional knapsack problem used as the example on the Wikipedia page for the Knapsack problem. Here is the problem statement.

```
from pyeasyga import pyeasyga

# setup data
data = [{'name': 'box1', 'value': 4, 'weight': 12},
        {'name': 'box2', 'value': 2, 'weight': 1},
        {'name': 'box3', 'value': 10, 'weight': 4},
        {'name': 'box4', 'value': 1, 'weight': 1},
        {'name': 'box5', 'value': 2, 'weight': 2}]

ga = pyeasyga.GeneticAlgorithm(data)        # initialise the GA with data

# define a fitness function
def fitness(individual, data):
    values, weights = 0, 0
    for selected, box in zip(individual, data):
        if selected:
            values += box.get('value')
            weights += box.get('weight')
    if weights > 15:
        values = 0
    return values

ga.fitness_function = fitness               # set the GA's fitness function
ga.run()                                    # run the GA
print ga.best_individual()                  # print the GA's best solution
```

To run:

```
$ python one_dimensional_knapsack.py
```

Output:

```
(15, [0, 1, 1, 1, 1])
```

i.e. if you select all boxes except the first one, you get a maximum amount of $15 while still keeping the overall weight under or equal to 15kg.

## 4.2 Multi-Dimensional Knapsack Problem

multi_dimensional_knapsack.py

This solves the multidimensional knapsack problem (MKP) seen here. It is a well-known NP-hard combinatorial optimisation problem.

```
from pyeasyga import pyeasyga

# setup data
data = [(821, 0.8, 118), (1144, 1, 322), (634, 0.7, 166), (701, 0.9, 195),
        (291, 0.9, 100), (1702, 0.8, 142), (1633, 0.7, 100), (1086, 0.6,
 ↪145),
        (124, 0.6, 100), (718, 0.9, 208), (976, 0.6, 100), (1438, 0.7, 312),
        (910, 1, 198), (148, 0.7, 171), (1636, 0.9, 117), (237, 0.6, 100),
        (771, 0.9, 329), (604, 0.6, 391), (1078, 0.6, 100), (640, 0.8, 120),
        (1510, 1, 188), (741, 0.6, 271), (1358, 0.9, 334), (1682, 0.7, 153),
        (993, 0.7, 130), (99, 0.7, 100), (1068, 0.8, 154), (1669, 1, 289)]

ga = pyeasyga.GeneticAlgorithm(data)        # initialise the GA with data
ga.population_size = 200                    # increase population size to
 ↪200 (default value is 50)

# define a fitness function
def fitness(individual, data):
    weight, volume, price = 0, 0, 0
    for (selected, item) in zip(individual, data):
        if selected:
            weight += item[0]
            volume += item[1]
            price += item[2]
    if weight > 12210 or volume > 12:
        price = 0
    return price

ga.fitness_function = fitness               # set the GA's fitness function
ga.run()                                    # run the GA
print ga.best_individual()                  # print the GA's best solution
```

To run:

```
$ python multi_dimensional_knapsack.py
```

Output:

```
(3531, [0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1,
 ↪0, 0, 1, 0, 1])
```

i.e. the indicated selection of items satisfies the required weight and volume constraints, and gives a total value of 3531.

## 4.3 8 Queens Puzzle

8_queens.py

This solves the 8 queens puzzle.

```
import random
from pyeasyga import pyeasyga

# setup seed data
seed_data = [0, 1, 2, 3, 4, 5, 6, 7]

# initialise the GA
ga = pyeasyga.GeneticAlgorithm(seed_data,
                               population_size=200,
                               generations=100,
                               crossover_probability=0.8,
                               mutation_probability=0.2,
                               elitism=True,
                               maximise_fitness=False)

# define and set function to create a candidate solution representation
def create_individual(data):
    individual = data[:]
    random.shuffle(individual)
    return individual

ga.create_individual = create_individual

# define and set the GA's crossover operation
def crossover(parent_1, parent_2):
    crossover_index = random.randrange(1, len(parent_1))
    child_1a = parent_1[:crossover_index]
    child_1b = [i for i in parent_2 if i not in child_1a]
    child_1 = child_1a + child_1b

    child_2a = parent_2[crossover_index:]
    child_2b = [i for i in parent_1 if i not in child_2a]
    child_2 = child_2a + child_2b

    return child_1, child_2

ga.crossover_function = crossover

# define and set the GA's mutation operation
def mutate(individual):
    mutate_index1 = random.randrange(len(individual))
    mutate_index2 = random.randrange(len(individual))
    individual[mutate_index1], individual[mutate_index2] = individual[mutate_
↪index2], individual[mutate_index1]

ga.mutate_function = mutate
```

(continues on next page)

(continued from previous page)

```
# define and set the GA's selection operation
def selection(population):
    return random.choice(population)

ga.selection_function = selection

# define a fitness function
def fitness (individual, data):
    collisions = 0
    for item in individual:
        item_index = individual.index(item)
        for elem in individual:
            elem_index = individual.index(elem)
            if item_index != elem_index:
                if item - (elem_index - item_index) == elem \
                    or (elem_index - item_index) + item == elem:
                    collisions += 1
    return collisions

ga.fitness_function = fitness       # set the GA's fitness function
ga.run()                            # run the GA

# function to print out chess board with queens placed in position
def print_board(board_representation):
    def print_x_in_row(row_length, x_position):
        print '',
        for _ in xrange(row_length):
            print '---',
        print '\n|',
        for i in xrange(row_length):
            if i == x_position:
                print '{} |'.format('X'),
            else:
                print '  |',
        print ''

    def print_board_bottom(row_length):
        print '',
        for _ in xrange(row_length):
            print '---',

    num_of_rows = len(board_representation)
    row_length = num_of_rows    #rows == columns in a chessboard

    for row in xrange(num_of_rows):
        print_x_in_row(row_length, board_representation[row])

    print_board_bottom(row_length)
    print '\n'

# print the GA's best solution; a solution is valid only if there are no
↪collisions
if ga.best_individual()[0] == 0:
    print ga.best_individual()
    print_board(ga.best_individual()[1])
else:
    print None
```

To run:

```
$ python 8_queens.py
```

Output:

```
(0, [2, 5, 7, 0, 3, 6, 4, 1])

 --- --- --- --- --- --- --- ---
|   |   | X |   |   |   |   |   |
 --- --- --- --- --- --- --- ---
|   |   |   |   |   | X |   |   |
 --- --- --- --- --- --- --- ---
|   |   |   |   |   |   |   | X |
 --- --- --- --- --- --- --- ---
| X |   |   |   |   |   |   |   |
 --- --- --- --- --- --- --- ---
|   |   |   | X |   |   |   |   |
 --- --- --- --- --- --- --- ---
|   |   |   |   |   |   | X |   |
 --- --- --- --- --- --- --- ---
|   |   |   |   | X |   |   |   |
 --- --- --- --- --- --- --- ---
|   | X |   |   |   |   |   |   |
 --- --- --- --- --- --- --- ---
```

CHAPTER 5

# Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

## 5.1 Types of Contributions

### 5.1.1 Report Bugs

Report bugs at https://github.com/remiomosowon/pyeasyga/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

### 5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

### 5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

### 5.1.4 Write Documentation

pyeasyga could always use more documentation, whether as part of the official pyeasyga docs, in docstrings, or even on the web in blog posts, articles, and such.

### 5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/remiomosowon/pyeasyga/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

## 5.2 Get Started!

Ready to contribute? Here's how to set up *pyeasyga* for local development.

1. Fork the *pyeasyga* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/pyeasyga.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

   ```
   $ mkvirtualenv pyeasyga
   $ cd pyeasyga/
   $ python setup.py develop
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

   ```
   $ flake8 pyeasyga tests
   $ python setup.py test
   $ tox
   ```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

## 5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7 and 3.4. Please make sure you don't break compatibility. Check https://travis-ci.org/remiomosowon/pyeasyga/pull_requests and make sure that the tests pass for all supported Python versions.

## 5.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_pyeasyga
```

# CHAPTER 6

# Credits

## 6.1 Development Lead

- Ayodeji Remi-Omosowon <remiomosowon@gmail.com>

## 6.2 Contributors

- Yasser Gonzalez <yasserglez@gmail.com>

CHAPTER 7

# History

## 7.1 v0.3.0

**2015-04-07**

- Added Python 3.4 support without breaking Python 2 compatibility (thanks to yasserglez)

## 7.2 v0.2.5

**2014-07-09**

- Added an example that solves the 8 Queens Puzzle

**2014-07-09**

- Modified the GeneticAlgorithm class initialisation parameters
- Made changes to USAGE documentation
- Added EXAMPLE documentation as a separate section

## 7.3 v0.2.4

**2014-07-07**

- Refactored most of the code; Made GeneticAlgorithm class more OOP
- Made changes to INSTALLATION documentation

## 7.4 v0.2.3

**2014-07-05**

- Fixed breaking python 2.6 build

## 7.5 v0.2.2

**2014-07-05**

- Removed duplicate 'Example' documentation; now maintaining only one copy in examples/README.rst
- Added link to jeffknupp's sandman repo in HISTORY
- Modified release option in Makefile to also upload project documentation
- Added INSTALLATION and EXAMPLE sections to README.rst
- Removed easy_install installation step from documentation (pip is sufficient)
- Added a simple example of usage to docs/usage.rst
- Reduced the default GA population and generation size (to allow applications that use the different parameters to run quickly)
- Modified tests to account for the new default population, generation size
- Added docstrings to all methods

## 7.6 v0.2.0

**2014-07-04**

- First upload to pypi.
- Added changes made to HISTORY (pypi upload, new version)

## 7.7 v0.1.0

**2014-06-23**

- Start of `pyeasyga` development.

**2014-07-03**

- Implemented all of basic GA functionality
- Fix issue with odd-numbered population that causes an off-by-one error in the population size
- Set default ga selection function to tournament_selection
- Created examples to show how to use the library
- Start versioning (better late than never); copied jeffknupp's update_version.sh from sandman

  **selected versioning standard:** major.minor.micro (e.g. 2.1.5)

  – major => big changes that can break compatibility

- minor => new features
- micro => bug fixes

CHAPTER 8

# Indices and tables

- genindex
- modindex