# DEAP Documentation

***Release 0.9.0***

**DEAP Project**

March 01, 2013

# CONTENTS

CHAPTER

ONE

# TUTORIAL

Although this tutorial doesn't make reference directly to the complete API of the framework, we think it is the place to start to understand the principles of DEAP. The core of the architecture is based on the `creator` and the `Toolbox`. Their usage to create types is shown in the first part of this tutorial. Then, a next step is taken in the direction of building generic algorithms by showing how to use the different tools present in the framework. Finally, we conclude on how those algorithms can be made parallel with minimal changes to the overall code (generally, adding a single line of code will do the job).

## 1.1 Where to Start?

If your are used to an other evolutionary algorithm framework, you'll notice we do things differently with DEAP. Instead of limiting you with predefined types, we provide ways of creating the appropriate ones. Instead of providing closed initializers, we enable you to customize them as you wish. Instead of suggesting unfit operators, we explicitly ask you to choose them wisely. Instead of implementing many sealed algorithms, we allow you to write the one that fit all your needs. This tutorial will present a quick overview of what DEAP is all about along with what every DEAP program is made of.

### 1.1.1 Types

The first thing to do is to think of the appropriate type for your problem. Then, instead of looking in the list of available types, DEAP enables you to build your own. This is done with the `creator` module. Creating an appropriate type might seems overwhelming but the creator makes it very easy. In fact, this is usually done in a single line. For example, the following creates a `FitnessMin` class for a minimization problem and an `Individual` class that is derived from a list with a fitness attribute set to the just created fitness.

```
from deap import base, creator
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)
```

That's it. More on creating types can be found in the *Creating Types* tutorial.

### 1.1.2 Initialization

Once the types are created you need to fill them with sometimes random values, sometime guessed ones. Again, DEAP provides an easy mechanism to do just that. The `Toolbox` is a container for tools of all sorts including initializers that can do what is needed of them. The following takes on the last lines of code to create the initializers for individuals containing random floating point numbers and for a population that contains them.

```
import random
from deap import tools

IND_SIZE = 10

toolbox = base.Toolbox()
toolbox.register("attribute", random.random)
toolbox.register("individual", tools.initRepeat, creator.Individual,
                 toolbox.attribute, n=IND_SIZE)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

This creates functions to initialize populations from individuals that are themselves initialized with random float numbers. The functions are registered in the toolbox with there default arguments under the given name. For example, it will be possible to call the function `toolbox.population()` to instantly create a population. More initialization methods are found in the *Creating Types* tutorial and the various *Examples*.

### 1.1.3 Operators

Operators are just like initalizers, excepted that some are already implemented in the `tools` module. Once you've chose the perfect ones simply register them in the toolbox. In addition you must create your evaluation function. This is how it is done in DEAP.

```
def evaluate(individual):
    return sum(individual),

toolbox.register("mate", tools.cxTwoPoints)
toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=1, indpb=0.1)
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("evaluate", evaluate)
```

The registered functions are renamed by the toolbox to allows genericity so that the algorithm does not depend on operators name. Note also that fitness values must be iterable, that is why we return tuple in the evaluate function. More on this in the *Next Step Toward Evolution* tutorial and *Examples*.

### 1.1.4 Algorithms

Now that everything is ready, we can start to write our own algorithm. It is usually done in a main function. For the purpose of completeness we will develop the complete generational algorithm.

```
def main():
    pop = toolbox.population(n=50)
    CXPB, MUTPB, NGEN = 0.5, 0.2, 40

    # Evaluate the entire population
    fitnesses = map(toolbox.evaluate, pop)
    for ind, fit in zip(pop, fitnesses):
        ind.fitness.values = fit

    for g in range(NGEN):
        # Select the next generation individuals
        offspring = toolbox.select(pop, len(pop))
        # Clone the selected individuals
        offspring = map(toolbox.clone, offspring)

        # Apply crossover and mutation on the offspring
```

```
        for child1, child2 in zip(offspring[::2], offspring[1::2]):
            if random.random() < CXPB:
                toolbox.mate(child1, child2)
                del child1.fitness.values
                del child2.fitness.values

        for mutant in offspring:
            if random.random() < MUTPB:
                toolbox.mutate(mutant)
                del mutant.fitness.values

        # Evaluate the individuals with an invalid fitness
        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        # The population is entirely replaced by the offspring
        pop[:] = offspring

    return pop
```

It is also possible to use one of the four algorithms readily available in the `algorithms` module, or build from some building blocks called variations also available in this module.

## 1.2 Creating Types

This tutorial shows how types are created using the creator and initialized using the toolbox.

### 1.2.1 Fitness

The provided `Fitness` class is an abstract class that needs a `weights` attribute in order to be functional. A minimizing fitness is built using negatives weights, while a maximizing fitness has positive weights. For example, the following line creates, in the `creator`, a ready to use single objective minimizing fitness named `FitnessMin`.

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

The `weights` argument must be a tuple so that multi objective and single objective fitnesses can be treated the same way. A `FitnessMulti` would be created the same way but using

```
creator.create("FitnessMulti", base.Fitness, weights=(-1.0, 1.0))
```

rendering a fitness that minimize the first objective and maximize the second one. The weights can also be used to variate the importance of each objective one against another. This means that the weights can be any real number and only the sign is used to determine if a maximization or minimization is done. An example of where the weights can be useful is in the crowding distance sort made in the NSGA-II selection algorithm.

### 1.2.2 Individual

Simply by thinking of the different flavours of evolutionary algorithms (GA, GP, ES, PSO, DE, ...), we notice that an extremely large variety of individuals are possible reinforcing the assumption that all types cannot be made available by developers. Here is a guide on how to create some of those individuals using the `creator` and initializing them using a `Toolbox`.

### List of Floats

The first individual created will be a simple list containing floats. In order to produce this kind of individual, we need to create an `Individual` class, using the creator, that will inherit from the standard `list` and have a `fitness` attribute. Then we will initialize this list using the `initRepeat()` helper function that will repeat `n` times the float generator that has been registered under the `attr_float()` alias of the toolbox. Note that the `attr_float()` is a direct reference to the `random()` function.

```
import random

from deap import base
from deap import creator
from deap import tools

creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)

IND_SIZE=10

toolbox = base.Toolbox()
toolbox.register("attr_float", random.random)
toolbox.register("individual", tools.initRepeat, creator.Individual,
                 toolbox.attr_float, n=IND_SIZE)
```

Calling `toolbox.individual()` will readily return a complete individual composed of `IND_SIZE` floating point numbers with a maximizing single objective fitness attribute.

Variations of this type are possible by inheriting from `array.array` or `numpy.ndarray` as following.

```
creator.create("Individual", array.array, typecode="d", fitness=creator.FitnessMax)
creator.create("Individual", numpy.ndarray, fitness=creator.FitnessMax)
```

The type inheriting from array needs a *typecode* on initialization just as the original class.

### Permutation

An individual for the permutation representation is almost similar to the general list individual. In fact they both inherit from the basic `list` type. The only difference is that instead of filling the list with a series of floats, we need to generate a random permutation and provide that permutation to the individual. First, the individual class is created the exact same way as the previous one. Then, an `indices()` function is added to the toolbox referring to the `sample()` function, sample is used instead of `shuffle()` because this last one does not return the shuffled list. The indices function returns a complete permutation of the numbers between `0` and `IND_SIZE - 1`. Finally, the individual is initialized with the `initIterate()` function which gives to the individual an iterable of what is produced by the call to the indices function.

```
import random

from deap import base
from deap import creator
from deap import tools

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

IND_SIZE=10

toolbox = base.Toolbox()
```

```
toolbox.register("indices", random.sample, range(IND_SIZE), IND_SIZE)
toolbox.register("individual", tools.initIterate, creator.Individual,
                 toolbox.indices)
```

Calling `toolbox.individual()` will readily return a complete individual that is a permutation of the integers `0` to `IND_SIZE` with a minimizing single objective fitness attribute.

### Arithmetic Expression

The next individual that is commonly used is a prefix tree of mathematical expressions. This time a `PrimitiveSet` must be defined containing all possible mathematical operators that our individual can use. Here the set is called `MAIN` and has a single variable defined by the arity. Operators `add()`, `sub()`, and `mul()` are added to the primitive set with each an arity of 2. Next, the `Individual` class is created as before but having an additional static attribute `pset` set to remember the global primitive set. This time the content of the individuals will be generated by the `genRamped()` function that generate trees in a list format based on a ramped procedure. Once again, the individual is initialised using the `initIterate()` function to give the complete generated iterable to the individual class.

```
import operator

from deap import base
from deap import creator
from deap import gp
from deap import tools

pset = gp.PrimitiveSet("MAIN", arity=1)
pset.addPrimitive(operator.add, 2)
pset.addPrimitive(operator.sub, 2)
pset.addPrimitive(operator.mul, 2)

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMin,
               pset=pset)

toolbox = base.Toolbox()
toolbox.register("expr", gp.genRamped, pset=pset, min_=1, max_=2)
toolbox.register("individual", tools.initIterate, creator.Individual,
                 toolbox.expr)
```

Calling `toolbox.individual()` will readily return a complete individual that is an arithmetic expression in the form of a prefix tree with a minimizing single objective fitness attribute.

### Evolution Strategy

Evolution strategies individuals are slightly different as they contain generally two list, one for the actual individual and one for its mutation parameters. This time instead of using the list base class we will inherit from an `array.array` for both the individual and the strategy. Since there is no helper function to generate two different vectors in a single object we must define this function our-self. The `initES()` function receives two classes and instantiate them generating itself the random numbers in the intervals provided for individuals of a given size.

```
import array
import random

from deap import base
from deap import creator
from deap import tools
```

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", array.array, typecode="d",
               fitness=creator.FitnessMin, strategy=None)
creator.create("Strategy", array.array, typecode="d")

def initES(icls, scls, size, imin, imax, smin, smax):
    ind = icls(random.uniform(imin, imax) for _ in range(size))
    ind.strategy = scls(random.uniform(smin, smax) for _ in range(size))
    return ind

IND_SIZE = 10
MIN_VALUE, MAX_VALUE = -5., 5.
MIN_STRAT, MAX_STRAT = -1., 1.

toolbox = base.Toolbox()
toolbox.register("individual", initES, creator.Individual,
                 creator.Strategy, IND_SIZE, MIN_VALUE, MAX_VALUE, MIN_STRAT,
                 MAX_STRAT)
```

Calling `toolbox.individual()` will readily return a complete evolution strategy with a strategy vector and a minimizing single objective fitness attribute.

### Particle

A particle is another special type of individual as it usually has a speed and generally remember its best position. This type of individual is created (once again) the same way inheriting from a list. This time `speed`, `best` and speed limits attributes are added to the object. Again, an initialization function `initParticle()` is also registered to produce the individual receiving the particle class, size, domain, and speed limits as arguments.

```
import random

from deap import base
from deap import creator
from deap import tools

creator.create("FitnessMax", base.Fitness, weights=(1.0, 1.0))
creator.create("Particle", list, fitness=creator.FitnessMax, speed=None,
               smin=None, smax=None, best=None)

def initParticle(pcls, size, pmin, pmax, smin, smax):
    part = pcls(random.uniform(pmin, pmax) for _ in xrange(size))
    part.speed = [random.uniform(smin, smax) for _ in xrange(size)]
    part.smin = smin
    part.smax = smax
    return part

toolbox = base.Toolbox()
toolbox.register("particle", initParticle, creator.Particle, size=2,
                 pmin=-6, pmax=6, smin=-3, smax=3)
```

Calling `toolbox.individual()` will readily return a complete particle with a speed vector and a maximizing two objectives fitness attribute.

#### A Funky One

Supposing your problem have very specific needs. It is also possible to build custom individuals very easily. The next individual created is a list of alternating integers and floating point numbers, using the `initCycle()` function.

```
import random

from deap import base
from deap import creator
from deap import tools

creator.create("FitnessMax", base.Fitness, weights=(1.0, 1.0))
creator.create("Individual", list, fitness=creator.FitnessMax)

toolbox = base.Toolbox()

INT_MIN, INT_MAX = 5, 10
FLT_MIN, FLT_MAX = -0.2, 0.8
N_CYCLES = 4

toolbox.register("attr_int", random.randint, INT_MIN, INT_MAX)
toolbox.register("attr_flt", random.uniform, FLT_MIN, FLT_MAX)
toolbox.register("individual", tools.initCycle, creator.Individual,
                 (toolbox.attr_int, toolbox.attr_flt), n=N_CYCLES)
```

Calling `toolbox.individual()` will readily return a complete individual of the form `[int float int float ... int float]` with a maximizing two objectives fitness attribute.

### 1.2.3 Population

Population are much like individuals, instead of being initialized with attributes it is filled with individuals, strategies or particles.

#### Bag

A bag population is the most commonly used type, it has no particular ordering although it is generally implemented using a list. Since the bag has no particular attribute it does not need any special class. The population is initialized using directly the toolbox and the `initRepeat()` function.

```
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

Calling `toolbox.population()` will readily return a complete population in a list providing a number of times the repeat helper must be repeated as an argument of the population function. The following example produces a population with 100 individuals.

```
toolbox.population(n=100)
```

#### Grid

A grid population is a special case of structured population where neighbouring individuals have a direct effect on each other. The individuals are distributed in grid where each cell contains a single individual. (Sadly?) It is no different than the list implementation of the bag population other that it is composed of lists of individuals.

```
toolbox.register("row", tools.initRepeat, list, toolbox.individual, n=N_COL)
toolbox.register("population", tools.initRepeat, list, toolbox.row, n=N_ROW)
```

Calling `toolbox.population()` will readily return a complete population where the individuals are accessible using two indices for example `pop[r][c]`. For the moment there is no algorithm specialized for structured population, we are waiting your submissions.

### Swarm

A swarm is used in particle swarm optimization, it is different in the sens that it contains a network of communication. The simplest network is the complete connection where each particle knows the best position of that ever been visited by any other particle. This is generally implemented by copying that global best position to a `gbest` attribute and the global best fitness to a `gbestfit` attribute.

```
creator.create("Swarm", list, gbest=None, gbestfit=creator.FitnessMax)
toolbox.register("swarm", tools.initRepeat, creator.Swarm, toolbox.particle)
```

Calling `toolbox.population()` will readily return a complete swarm. After each evaluation the `gbest` and `gbestfit` should be set by the algorithm to reflect the best found position and fitness.

### Demes

A deme is a sub-population that is contained in a population. It is similar has an island in the island model. Demes being only sub-population are in fact no different than population other than by their names. Here we create a population containing 3 demes each having a different number of individuals using the *n* argument of the `initRepeat()` function.

```
toolbox.register("deme", tools.initRepeat, list, toolbox.individual)

DEME_SIZES = 10, 50, 100
population = [toolbox.deme(n=i) for i in DEME_SIZES]
```

### Seeding a Population

Sometimes, a first guess population can be used to initialize an evolutionary algorithm. The key idea to initialize a population with not random individuals is to have an individual initializer that takes a content as argument.

```
import json

from deap import base
from deap import creator

creator.create("FitnessMax", base.Fitness, weights=(1.0, 1.0))
creator.create("Individual", list, fitness=creator.FitnessMax)

def initIndividual(icls, content):
    return icls(content)

def initPopulation(pcls, ind_init, filename):
    contents = json.load(open(filename, "r"))
    return pcls(ind_init(c) for c in contents)

toolbox = base.Toolbox()

toolbox.register("individual_guess", initIndividual, creator.Individual)
toolbox.register("population_guess", initPopulation, list, toolbox.individual_guess, "my_guess.json")

population = toolbox.population_guess()
```

The population will be initialized from the file `my_guess.json` that shall contain a list of first guess individuals. This initialization can be combined with a regular initialization to have part random and part not random individuals. Note that the definition of `initIndividual()` and the registration of `individual_guess()` are optional as the default constructor of a list is similar. Removing those lines leads to the following.

```
toolbox.register("population_guess", initPopulation, list, creator.Individual, "my_guess.json")
```

## 1.3 Next Step Toward Evolution

Before starting with complex algorithms, we will see some basis of DEAP. First, we will start by creating simple individuals (as seen in the *Creating Types* tutorial) and make them interact with each other using different operators. Afterwards, we will learn how to use the algorithms and other tools.

### 1.3.1 A First Individual

First import the required modules and register the different functions required to create individuals that are a list of floats with a minimizing two objectives fitness.

```
import random

from deap import base
from deap import creator
from deap import tools

IND_SIZE = 5

creator.create("FitnessMin", base.Fitness, weights=(-1.0, -1.0))
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()
toolbox.register("attr_float", random.random)
toolbox.register("individual", tools.initRepeat, creator.Individual,
                 toolbox.attr_float, n=IND_SIZE)
```

The first individual can now be built by adding the appropriate line to the script.

```
ind1 = toolbox.individual()
```

Printing the individual `ind1` and checking if its fitness is valid will give something like this

```
print ind1               # [0.86..., 0.27..., 0.70..., 0.03..., 0.87...]
print ind1.fitness.valid # False
```

The individual is printed as its base class representation (here a list) and the fitness is invalid because it contains no values.

### 1.3.2 Evaluation

The evaluation is the most personal part of an evolutionary algorithm, it is the only part of the library that you must write your-self. A typical evaluation function takes one individual as argument and return its fitness as a `tuple`. As shown in the in the *Core Architecture* section, a fitness is a list of floating point values and has a property `valid` to know if this individual shall be re-evaluated. The fitness is set by setting the `values` to the associated `tuple`. For example, the following evaluates the previously created individual `ind1` and assign its fitness to the corresponding values.

```
def evaluate(individual):
    # Do some hard computing on the individual
    a = sum(individual)
    b = len(individual)
    return a, 1. / b

ind1.fitness.values = evaluate(ind1)
print ind1.fitness.valid    # True
print ind1.fitness          # (2.73, 0.2)
```

Dealing with single objective fitness is not different, the evaluation function **must** return a tuple because single-objective is treated as a special case of multi-objective.

### 1.3.3 Mutation

The next kind of operator that we will present is the mutation operator. There is a variety of mutation operators in the `deap.tools` module. Each mutation has its own characteristics and may be applied to different type of individual. Be careful to read the documentation of the selected operator in order to avoid undesirable behaviour.

The general rule for mutation operators is that they **only** mutate, this means that an independent copy must be made prior to mutating the individual if the original individual has to be kept or is a *reference* to an other individual (see the selection operator).

In order to apply a mutation (here a gaussian mutation) on the individual `ind1`, simply apply the desired function.

```
mutant = toolbox.clone(ind1)
ind2, = tools.mutGaussian(mutant, mu=0.0, sigma=0.2, indpb=0.2)
del mutant.fitness.values
```

The fitness' values are deleted because they not related to the individual anymore. As stated above, the mutation does mutate and only mutate an individual it is not responsible of invalidating the fitness nor anything else. The following shows that `ind2` and `mutant` are in fact the same individual.

```
print ind2 is mutant    # True
print mutant is ind1    # False
```

### 1.3.4 Crossover

The second kind of operator that we will present is the crossover operator. There is a variety of crossover operators in the `deap.tools` module. Each crossover has its own characteristics and may be applied to different type of individuals. Be careful to read the documentation of the selected operator in order to avoid undesirable behaviour.

The general rule for crossover operators is that they **only** mate individuals, this means that an independent copies must be made prior to mating the individuals if the original individuals have to be kept or is are *references* to other individuals (see the selection operator).

Lets apply a crossover operation to produce the two children that are cloned beforehand.

```
child1, child2 = [toolbox.clone(ind) for ind in (ind1, ind2)]
tools.cxBlend(child1, child2, 0.5)
del child1.fitness.values
del child2.fitness.values
```

---

**Note:** Just as a remark on the language, the form `toolbox.clone([ind1, ind2])` cannot be used because if `ind1` and `ind2` are referring to the same location in memory (the same individual) there will be a single independent copy of the individual and the second one will be a reference to this same independent copy. This is caused by the

mechanism that prevents recursive loops. The first time the individual is seen, it is put in the "memo" dictionary, the next time it is seen the deep copy stops for that object and puts a reference to that previously created deep copy. Care should be taken when deep copying containers.

### 1.3.5 Selection

Selection is made among a population by the selection operators that are available in the `deap.operators` module. The selection operator usually takes as first argument an iterable container of individuals and the number of individuals to select. It returns a list containing the references to the selected individuals. The selection is made as follow.

```
selected = tools.selBest([child1, child2], 2)
print child1 in selected        # True
```

**Warning:** It is **very** important here to note that the selection operators does not duplicate any individual during the selection process. If an individual is selected twice and one of either object is modified, the other will also be modified. Only a reference to the individual is copied. Just like every other operator it selects and only selects.

Usually duplication of the entire population will be made after selection or before variation.

```
selected = toolbox.select(population, LAMBDA)
offspring = [toolbox.clone(ind) for ind in selected]
```

### 1.3.6 Using the Toolbox

The toolbox is intended to contain all the evolutionary tools, from the object initializers to the evaluation operator. It allows easy configuration of each algorithms. The toolbox has basically two methods, `register()` and `unregister()`, that are used to add or remove tools from the toolbox.

This part of the tutorial will focus on registration of the evolutionary tools in the toolbox rather than the initialization tools. The usual names for the evolutionary tools are `mate()`, `mutate()`, `evaluate()` and `select()`, however, any name can be registered as long as it is unique. Here is how they are registered in the toolbox.

```
from deap import base
from deap import tools

toolbox = base.Toolbox()

def evaluateInd(individual):
    # Do some computation
    return result,

toolbox.register("mate", tools.cxTwoPoints)
toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=1, indpb=0.2)
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("evaluate", evaluateInd)
```

Using the toolbox for registering tools helps keeping the rest of the algorithms independent from the operator set. Using this scheme makes it very easy to locate and change any tool in the toolbox if needed.

#### Using the Tools

When building evolutionary algorithms the toolbox is used to contain the operators, which are called using their generic name. For example, here is a very simple generational evolutionary algorithm.

```
for g in range(NGEN):
    # Select the next generation individuals
    offspring = toolbox.select(pop, len(pop))
    # Clone the selected individuals
    offspring = map(toolbox.clone, offspring)

    # Apply crossover on the offspring
    for child1, child2 in zip(offspring[::2], offspring[1::2]):
        if random.random() < CXPB:
            toolbox.mate(child1, child2)
            del child1.fitness.values
            del child2.fitness.values

    # Apply mutation on the offspring
    for mutant in offspring:
        if random.random() < MUTPB:
            toolbox.mutate(mutant)
            del mutant.fitness.values

    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
    fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    # The population is entirely replaced by the offspring
    pop[:] = offspring
```

This is a complete algorithm. It is generic enough to accept any kind of individual and any operator, as long as the operators are suitable for the chosen individual type. As shown in the last example, the usage of the toolbox allows to write algorithms that are as close as possible to the pseudo code. Now it is up to you to write and experiment your own.

### Tool Decoration

Tool decoration is a very powerful feature that helps to control very precise thing during an evolution without changing anything in the algorithm or operators. A decorator is a wrapper that is called instead of a function. It is asked to make some initialization and termination work before and after the actual function is called. For example, in the case of a constrained domain, one can apply a decorator to the mutation and crossover in order to keep any individual from being out-of-bound. The following defines a decorator that checks if any attribute in the list is out-of-bound and clips it if it is the case. The decorator is defined using three functions in order to receive the *min* and *max* arguments. Whenever the mutation or crossover is called, bounds will be check on the resulting individuals.

```
def checkBounds(min, max):
    def decorator(func):
        def wrapper(*args, **kargs):
            offspring = func(*args, **kargs)
            for child in offspring:
                for i in xrange(len(child)):
                    if child[i] > max:
                        child[i] = max
                    elif child[i] < min:
                        child[i] = min
            return offspring
        return wrapper
    return decorator
```

```
toolbox.register("mate", tools.cxBlend, alpha=0.2)
toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=2)

toolbox.decorate("mate", checkBounds(MIN, MAX))
toolbox.decorate("mutate", checkBounds(MIN, MAX))
```

This will work on crossover and mutation because both return a tuple of individuals. The mutation is often considered to return a single individual but again like for the evaluation, the single individual case is a special case of the multiple individual case.

For more information on decorators, see Introduction to Python Decorators and Python Decorator Libary.

### 1.3.7 Variations

Variations allows to build simple algorithms using predefined small building blocks. In order to use a variation, the toolbox must be set to contain the required operators. For example in the lastly presented complete algorithm, the crossover and mutation are regrouped in the `varAnd()` function, this function requires the toolbox to contain the `mate()` and `mutate()` functions. The variations can be used to simplify the writing of an algorithm as follow.

```
from deap import algorithms

for g in range(NGEN):
    # Select and clone the next generation individuals
    offspring = map(toolbox.clone, toolbox.select(pop, len(pop)))

    # Apply crossover and mutation on the offspring
    offspring = algorithms.varAnd(offspring, toolbox, CXPB, MUTPB)

    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
    fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    # The population is entirely replaced by the offspring
    pop[:] = offspring
```

This last example shows that using the variations makes it straight forward to build algorithms that are very close to the pseudo-code.

### 1.3.8 Algorithms

There is several algorithms implemented in the `algorithms` module. They are very simple and reflect the basic types of evolutionary algorithms present in the literature. The algorithms use a `Toolbox` as defined in the last sections. In order to setup a toolbox for an algorithm, you must register the desired operators under a specified names, refer to the documentation of the selected algorithm for more details. Once the toolbox is ready, it is time to launch the algorithm. The simple evolutionary algorithm takes 5 arguments, a *population*, a *toolbox*, a probability of mating each individual at each generation (*cxpb*), a probability of mutating each individual at each generation (*mutpb*) and a number of generations to accomplish (*ngen*).

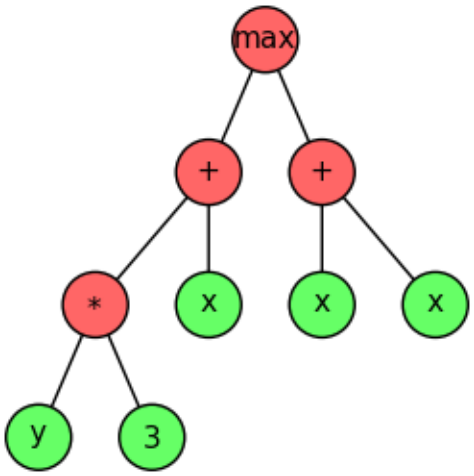
```
from deap import algorithms

algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=50)
```

The best way to understand what the simple evolutionary algorithm does, it to take a look at the documentation or the source code

Now that you built your own evolutionary algorithm in python, you are welcome to gives us feedback and appreciation. We would also really like to hear about your project and success stories with DEAP.

## 1.4 Genetic Programming

Genetic programming is a special field of evolutionary computation that aims at building programs automatically to solve problems independently of their domain. Although there exists diverse representations used to evolve programs, the most common is the syntax tree.




For example, the above figure presents the program $\max(x + 3 * y, x + x)$. For this tree and further examples, the leaves of the tree, in green, are called terminals, while the internal nodes, in red, are called primitves. The terminals are divided in two subtypes, the constants and the arguments. The constants remain the same for the entire evolution while the arguments are the program inputs. For the last presented tree, the arguments are the variables $x$ and $y$, and the constant is the number 3.

In DEAP, user defined primitives and terminals are contained in a primitive set. For now, there exists two kinds of primitive set, the loosely and the strongly typed.

### 1.4.1 Loosely Typed GP

Loosely typed GP does not enforce specific type between the nodes. More specifically, primitives' arguments can be any primitives or terminals present in the primitive set.

The following code define a loosely typed `PrimitiveSet` for the previous tree

```
pset = PrimitiveSet("main", 2)
pset.addPrimitive(max, 2)
pset.addPrimitive(operator.add, 2)
pset.addPrimitive(operator.mul, 2)
pset.addTerminal(3)
```

The first line creates a primitive set. Its argument are the name of the procedure it will generate `"main"` and its number of inputs, 2. The next three lines add function as primitives. The first argument is the function to add and the second argument the function arity. The last line adds a constant terminal. Currently the default names for the arguments are `"ARG0"` and `"ARG1"`. To change it to `"x"` and `"y"`, simply call

```
pset.renameArguments(ARG0="x")
pset.renameArguments(ARG1="y")
```

In this case, all functons take two arguments. Having a 1 argument negation function, for example, could be done with

```
pset.addPrimitive(operator.neg, 1)
```

Our primitive set is now ready to generate some trees. The `gp` module contains three prefix expression generation functions `genFull()`, `genGrow()`, and `genRamped()`. Their first argument is a primitive set and they return a valid prefix expression in the form of a list of primitives. The content of this list can be read by the `PrimitiveTree` class to create a prefix tree.

```
expr = genFull(pset, min_=1, max_=3)
tree = PrimitiveTree(expr)
```

The last code produces a valid full tree with height randomly chosen between 1 and 3.

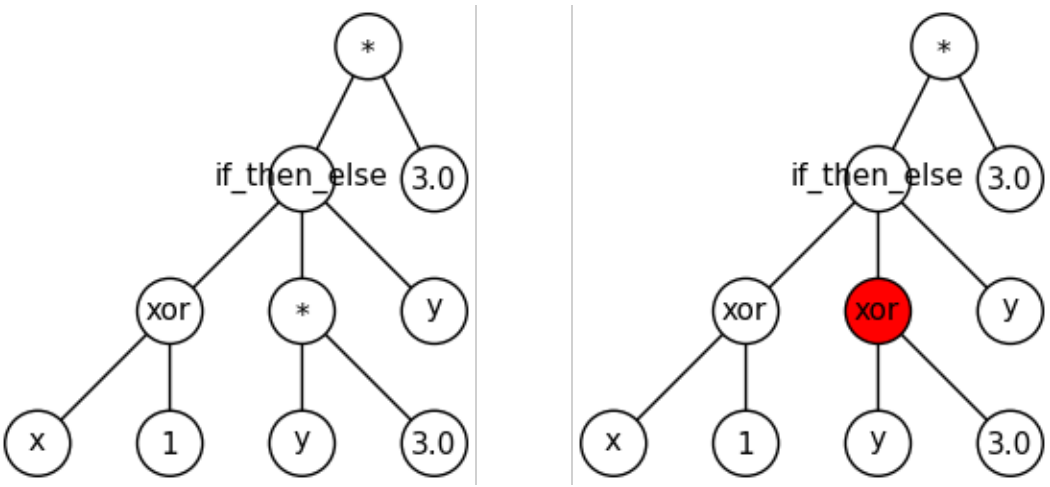
### 1.4.2 Strongly Typed GP

In strongly typed GP, every primitive and terminal is assigned a specific type. The output type of a primitive must match the input type of another one for them to be connected. For example, a primitive can return a boolean and this value is guaranteed to not be multipled with a float if the multiplication operator operates only on floats.

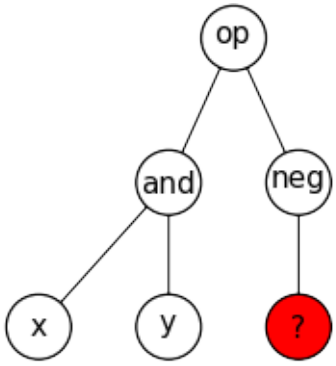
```
def if_then_else(input, output1, output2):
    return output1 if input else output2

pset = PrimitiveSetTyped("main", [bool, float], float)
pset.addPrimitive(operator.xor, [bool, bool], bool)
pset.addPrimitive(operator.mul, [float, float], float)
pset.addPrimitive(if_then_else, [bool, float, float], float)
pset.addTerminal(3.0, float)
pset.addTerminal(1, bool)

pset.renameArguments(ARG0="x")
pset.renameArguments(ARG1="y")
```

In the last code sample, we first define an *if then else* function, that returns the second argument if the first argument is true and the third one otherwise. Then, we define our `PrimitiveSetTyped`. Again, the procedure is named `"main"`. The second argument defines the input types of the program, here `"x"` is a `bool` and `"y"` is a `float`. The third argument defines the output type of the program, a `float`. Adding primitives to this primitive now requires to set the input and output types of the primitives and terminal. For example, we define our `"if_then_else"` function first argument as a boolean, the second and third argument have to be floats. The function is defined as returning a float. We now understand that the multiplication primitive can only have the terminal `3.0`, the `if_then_else` function or the `"y"` as input, which are the only floats defined.

The preceding code can produce the tree on the left but not the one on the right because the type restrictions.




**Note:** The generation of trees is done randomly while making sure type constraints are respected. If any primitive as an input type that no primitive and terminal can provide, chances are that this primitive will be picked and placed in the tree, resulting in the imposibility to complete the tree within the limit fixed by the generator. For example, when generating a full tree of height 2, suppose `"op"` takes a boolean and a float, `"and"` takes 2 boolean and `"neg"` takes a float, no terminal is defined and the arguments are booleans. The following situation will occur where no terminal can be placed to complete the tree.




In this case, DEAP raises an `IndexError` with the message `"The gp.generate function tried to add a terminal of type float, but there is none available."`

### 1.4.3 Ephemeral Constants

An ephemeral constant is a terminal encapsulating a value that is generated from a given function at run time. Ephemeral constants allow to have terminals that don't have all the same values. For example, to create an ephemeral constant that takes its value in $[-1, 1)$ we use

```
pset.addEphemeralConstant(lambda: random.uniform(-1, 1))
```

The ephemeral constant value is determined when it is inserted in the tree and never changes unless it is replaced by another ephemeral constant. Since it is a terminal, ephemeral constant can also be typed

```
pset.addEphemeralConstant(lambda: random.randint(-10, 10), int)
```

### 1.4.4 Generation of Tree Individuals

The code presented in the last two sections produce valid trees. However, as in the *Next Step Toward Evolution* tutorial, these trees are yet valid individuals for evolution. One must combine the creator and the toolbox to produce valid individuals. We need to create the `Fitness` and the `Individual` classes. To the `Individual`, in addition to the fitness we add a reference to the primitive set. This is used by some of the gp operators to modify the individuals.

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMin,
               pset=pset)
```

We then register the generation functions into a `Toolbox`.

```
toolbox = base.Toolbox()
toolbox.register("expr", gp.genFull, pset=pset, min_=1, max_=3)
toolbox.register("individual", tools.initIterate, creator.Individual,
                 toolbox.expr)
```

Calling `toolbox.individual()` readily returns an individual of type `PrimitiveTree`.

### 1.4.5 Evaluation of Trees

In DEAP, trees can be translated to readable Python code and compiled to Python code object using functions provided by the `gp` module. The first function, `stringify()` takes an expression or a PrimitiveTree and translates it into readable Python code. For example, the following lines generate a tree and output the code from the first example primitive set

```
>>> expr = genFull(pset, min_=1, max_=3)
>>> tree = PrimitiveTree(expr)
>>> stringify(tree)
'mul(add(x, x), max(y, x))'
```

Now, this string represents the program we just generated, but it cannot yet be executed. To do so, we have to compile the expression to Python code object. Since this function has two inputs, we wish to compile the code into a callable object. This is possible with `lambdify()`. The term *lambdify* comes from the fact that it returns a `lambda` function corresponding to the code. `lambdify()` takes two arguments, the expression to compile and the associated primitive set. The following example compiles the preceding tree and evaluates the resulting function for $x = 1$ and $y = 2$.

```
>>> function = lambdify(tree, pset)
>>> function(1, 2)
4
```

Finally, when the generated program has no input argument and terminals are functions, the expression can be compiled to byte code using the function `evaluate()`. An example of this sort of problem is the *Artificial Ant Problem*.

### 1.4.6 Tree Size Limit and Bloat Control

Since DEAP uses the Python parser to compile the code represented by the trees, it inherits from its limitations. The most commonly encountered restriction is the parsing stack limit. Python interpreter parser stack limit is commonly fixed between 92 and 99. This means that an expression can at most be composed of 91 succeeding primitives. In other words, a tree can have a maximum depth of 91. When the limit is exceeded, Python raises the following error

```
s_push: parser stack overflow
Traceback (most recent call last):
[...]
MemoryError
```

Since this limit is hardcoded in the interpreter, there exists no easy way to increase it. Furthermore, in GP this error commonly stems from a phenomema known as bloat. That is, the produced individuals have reached a point where they contain too much primitives to effectively solved the problem and the evolution stagnates. To counter this, DEAP provides different functions that can effectively maintain the size and height of the trees under an acceptable limit. These operators are listed in the GP section of *Operators*.

### 1.4.7 How to Evolve Programs

The different ways to evolve program trees are presented through the *Genetic Programming (GP)* examples.

## 1.5 Speed up the evolution

Although pure performance is not one of the most important objectives of DEAP, it may be good to know some ways to speed things up. This section will present various ways to improve performance without loosing too much of the DEAP ease of use. To show an approximation of the speedup you can achieve with each technique, an example is provided (despite the fact that every problem is different and will obviously produce different speedups).

---

**Note:** The benchmarks were run on a Linux workstation (Core i7 920, 8 GB), with a standard Python distribution (2.7) and DEAP 0.8.

---

### 1.5.1 The comparison problem : Sorting Networks

From Wikipedia : *A sorting network is an abstract mathematical model of a network of wires and comparator modules that is used to sort a sequence of numbers. Each comparator connects two wires and sort the values by outputting the smaller value to one wire, and a larger value to the other.*

Genetic algorithms are an interesting way to generate arbitrary sorting networks, as the fitness function of a network can easily be defined. Moreover, this problem is a good benchmark for evolutionnary framework : its complexity (and therefore the resources needed) can be tuned by changing the number of inputs. A 6-inputs problems is fairly simple and will be solved in a few seconds, while a 20-inputs one will take several minutes for each *generation*.

For this benchmark, we used an input size of 12, with 500 individuals which are allowed to evolve for 20 generations. Times reported are the total duration of the program, including initialization.

### 1.5.2 Normal run

*Time taken : 147 seconds (Reference time)*

With no optimizations at all, the level of performance reached is not impressive, yet sufficient for most users, especially for prototyping.

### 1.5.3 Using PyPy

*Time taken : 36 seconds (4.1x faster)*

PyPy (here in version 1.7) is an alternative Python interpreter which includes a Just-In-Time (JIT) compiler to greatly improve performances, especially when facing several rehearsals of the same loop or function. No change at all is needed in the program, and except for external C modules and advanced properties, PyPy is fully compliant with the Python 2.7 API.

To run your evolution under PyPy, simply download it, install it, and launch :

```
pypy your_program.py
```

Considering the ease of use, the speedup reached is substantial, and therefore makes a good first possibility to accelerate the execution.

### 1.5.4 Using the C++ version of the NSGA II algorithm

*Time taken : 121 seconds (1.2x faster)*

Starting with DEAP 0.8, a module named *cTools* is provided. This module includes a subset of the deap.tools module, but is implemented in C to improve performances. Only the bottleneck functions (in term of computational effort) are provided in *cTools*, with the exact same API and behaviour as their pure-Python implementation. The NSGA-II selection algorithm is one of them, and can be used here to improve performances, by only adding one line (the module inclusion) and changing a second (the registration of the selection algorithm).

```
from deap import cTools

toolbox.register("select", cTools.selNSGA2)
```

As one can see, the speedup reached is somewhat modest, since the main bottleneck remains the evaluation function. However, the improvement remains, and the coding effort needed is minimal; we will also see that it can be combined with other techniques to reach a better speedup.

---

**Note:** The cTools module is built at installation time (i.e. when executing the setup.py file). If no compiler is available, or if the building process failed for some reason, the cTools module will not be available.

---

### 1.5.5 Using an home-made C++ evaluation function

*Time taken : 33 seconds (4.5x faster)* This time, we look at an heavier optimization : replacement of the evaluation function by its C equivalent. CPython provides a C/C++ API to Python objects, and allows the writing of a C extension module relatively easily. However, this is problem specific, and can not be used with an other Python interpreter than CPython (like PyPy).

In this case, the extension code has approximately 130 lines of C++ code, from which about 100 are the evaluation function (the other parts are declarations needed by the Python interpreter to build and use the extension). The module is compiled with easy_install, and can thereafter be used as a normal Python module :

```
import SNC as snc

def evalEvoSN(individual, dimension):
    fit,depth,length= snc.evalNetwork(dimension, individual)
    return fit, length, depth
```

```
toolbox.register("evaluate", evalEvoSN, dimension=INPUTS)
```

The speedup obtained is notable, up to 5 times faster. At this point, the part of the computational effort taken by the evaluation drop from 80% to 10%. But what makes the other 90%?

### 1.5.6 Combining C++ version of NSGA II and evaluation function

*Time taken : 11 seconds (13.4x faster)*

For our last try, we use both the C version of NSGA-II and the C version of the evaluation function. This time, we clearly see an impressive improvement in term of computation speed, it is almost 15 times faster. This speed brings DEAP in the same range of performances as compiled (static) programs (like OpenBeagle): a small overhead is still produced by the systematic deep copy of the individuals and the use of some pure Python functions, but this is clearly not a bad performance at all considering that the program did not changed that much.

### 1.5.7 Speedups summary

It should be noted that, apart the evaluation function, all the other steps of the evolution (crossovers, mutations, copy, initialization, etc.) are still programmed in Python, and thus benefit from its ease of use. Adding a statistical measure or a sorting network viewer, trying other complicated mutations operators, reading new individuals from a database or an XML file and checkpointing the evolution at any generation is still far easier than with any compiled evolution framework, thanks to the power of Python. So, by adding a minimal complexity to the critical parts, one can still achieve excellent performances without sacrificing the beauty of the code and its clarity.

| Method | Time (s) | Speedup |
|---|---|---|
| Pure Python | 147 | 1.0x |
| PyPy 1.7 | 36 | 4.1x |
| C++ NSGA II | 121 | 1.2x |
| Custom C++ evaluation function | 33 | 4.5x |
| C++ NSGA II and C++ evaluation | 11 | 13.4x |

To complete this test, we also ran the problem with an harder parametrization (16 inputs instead of 12). It took *1997 seconds* with standard python interpreter, compared to *469 seconds* with PyPy (4.3x faster) and *124 seconds* when using C++ version for both NSGA II and evaluator, that is a speedup of *16.1x*. In other terms, we reduced the computation time from more than half an hour to a small 2 minutes...

| Method | Time (s) | Speedup |
|---|---|---|
| Pure Python | 1997 | 1.0x |
| PyPy 1.7 | 469 | 4.3x |
| C++ NSGA II and C++ evaluation | 124 | 16.1x |

### 1.5.8 Distribution

The previous optimizations were done by improving the execution speed itself. To speed up the execution further, distribution might be a good solution, especially if the computational effort is concentrated in a specific part of the program (in evolutionnary algorithms, this is often the evaluation function). DEAP offers some simple ways to distribute your code without effort, look at the specific page *Using Multiple Processors* to learn more about it.

## 1.6 Using Multiple Processors

This section of the tutorial shows all the work that is needed to distribute operations in deap. Distribution relies on serialization of objects and serialization is usually done by pickling, thus all objects that are distributed (functions and arguments, e.g. individuals and parameters) must be pickleable. This means modifications made to an object on a distant processing unit will not be made available to the other processing units (including the master one) if it is not explicitly communicated through function arguments and return values.

### 1.6.1 Scalable Concurent Operations in Python (SCOOP)

SCOOP is a distributed task module allowing concurrent parallel programming on various environments, from heterogeneous grids to supercomputers. It has an interface similar to the `concurrent.futures` module introduced in Python 3.2. Its two simple functions `submit()` and `map()` allow to distribute computation efficiently and easily over a grid of computers.

In the *last section* a complete algorithm was exposed with the `toolbox.map()` left to the default `map()`. In order to distribute the evaluations, we will replace this map by the one from SCOOP.

```
from scoop import futures

toolbox.register("map", futures.map)
```

Once this line is added, your program absolutly needs to be run from a `main()` function as mentionned in the scoop documentation. To run your program, use scoop as the main module.

```
$ python -m scoop your_program.py
```

That is it, your program has been run in parallel on all available processors.

### 1.6.2 Distributed Task Manager

Deprecated since version 0.9: Use SCOOP instead, it is a fork of DTM and developped by the same group. Distributing tasks on multiple computers is taken care of by the distributed task manager module `dtm`. Its API similar to the multiprocessing module allows it to be very easy to use. In the *last section* a complete algorithm was exposed with the `toolbox.map()` left to the default `map()`. In order to distribute the evaluations, simply replace this map with the one provided by the dtm module and tell to dtm which function is the main program here it is the `main()` function.

```
from deap import dtm

toolbox.register("map", dtm.map)

def main():
    # My evolutionary algorithm
    pass

if __name__ == "__main__":
    dtm.start(main)
```

That's it. The map operation contained in the toolbox will now be distributed. The next time you run the algorithm, it will run on the number of cores specified to the `mpirun` command used to run the python script. The usual bash command to use dtm will be :

```
$ mpirun [options] python my_script.py
```

### 1.6.3 Multiprocessing Module

Using the `multiprocessing` module is exactly similar to using the distributed task manager. Again in the toolbox, replace the appropriate function by the distributed one.

```
import multiprocessing

pool = multiprocessing.Pool()
toolbox.register("map", pool.map)

# Continue on with the evolutionary algorithm
```

**Warning:** As stated in the `multiprocessing` guidelines, under Windows, a process pool must be protected in a `if __name__ == "__main__"` section because of the way processes are initialized.

---

**Note:** While Python 2.6 is required for the multiprocessing module, the pickling of partial function is possible only since Python 2.7 (or 3.1), earlier version of Python may throw some strange errors when using partial function in the multiprocessing `multiprocessing.Pool.map()`. This may be avoided by creating local function outside of the toolbox (in Python version 2.6).

---

---

**Note:** The pickling of lambda function is not yet available in Python.

---

## 1.7 Benchmarking Against the Best (BBOB)

Once you've created your own algorithm, the structure of DEAP allows you to benchmark it against the best algorithms very easily. The interface of the Black-Box Optimization Benchmark (BBOB) is compatible with the toolbox. In fact, once your new algorithm is encapsulated in a main function there is almost nothing else to do on DEAP's side. This tutorial will review the essential steps to bring everything to work with the very basic *One Fifth Rule*.

### 1.7.1 Preparing the Algorithm

The BBOB makes use of many continuous functions on which will be tested the algorithm. These function are given as argument to the algorithm, thus the toolbox shall register the evaluation in the main function.

The evaluation functions provided by BBOB returns a fitness as a single value. The first step is then to transform them in a single element tuple as required by DEAP philosophy on single objective optimization. We will use a decorator for this.

```
def tupleize(func):
    """A decorator that tuple-ize the result of a function. This is usefull
    when the evaluation function returns a single value.
    """
    def wrapper(*args, **kargs):
        return func(*args, **kargs),
    return wrapper
```

The algorithm is encapsulated in a main function that receives four arguments, the evaluation function, the dimensionality of the problem, the maximum number of evaluations and the target value to reach. As stated earlier, the toolbox is initialized in the main function with the `update()` function (described in the example) and the evaluation function received, which is decorated by our tuple-izer.

Then, the target fitness value is encapsulated in a `FitnessMin` object so that we can easily compare the individuals with it. Following is simply the algorithm, which is explained in the *One Fifth Rule* example.

```
def main(func, dim, maxfuncevals, ftarget=None):
    toolbox = base.Toolbox()
    toolbox.register("update", update)
    toolbox.register("evaluate", func)
    toolbox.decorate("evaluate", tupleize)

    # Create the desired optimal function value as a Fitness object
    # for later comparison
    opt = creator.FitnessMin((ftarget,))

    # Interval in which to initialize the optimizer
    interval = -5, 5
    sigma = (interval[1] - interval[0])/2.0
    alpha = 2.0**(1.0/dim)

    # Initialize best randomly and worst as a place holder
    best = creator.Individual(random.uniform(interval[0], interval[1]) for _ in range(dim))
    worst = creator.Individual([0.0] * dim)

    # Evaluate the first individual
    best.fitness.values = toolbox.evaluate(best)

    # Evolve until ftarget is reached or the number of evaluation
    # is exausted (maxfuncevals)
    for g in range(1, maxfuncevals):
        toolbox.update(worst, best, sigma)
        worst.fitness.values = toolbox.evaluate(worst)
        if best.fitness <= worst.fitness:
            # Incease mutation strength and swap the individual
            sigma = sigma * alpha
            best, worst = worst, best
        else:
            # Decrease mutation strength
            sigma = sigma * alpha**(-0.25)

        # Test if we reached the optimum of the function
        # Remember that ">" for fitness means better (not greater)
        if best.fitness > opt:
            return best

    return best
```

### 1.7.2 Running the Benchmark

Now that the algorithm is ready, it is time to run it under the BBOB. The following code is taken from the BBOB example with added comments. The `fgeneric` module provides a `LoggingFunction`, which take care of outputting all necessary data to compare the tested algorithm with the other ones published and to be published.

This logger contains the current problem instance and provides the problem target. Since it is responsible of logging each evaluation function call, there is even no need to save the best individual found by our algorithm (call to the `main()` function). The single line that is related to the provided algorithm in the call to the `main()` function.

```
from deap import benchmarks

import fgeneric
```

```
    return best

if __name__ == "__main__":
    # Maximum number of restart for an algorithm that detects stagnation
    maxrestarts = 1000

    # Create a COCO experiment that will log the results under the
    # ./output directory
    e = fgeneric.LoggingFunction("output")

    # Iterate over all desired test dimensions
    for dim in (2, 3, 5, 10, 20, 40):
        # Set the maximum number function evaluation granted to the algorithm
        # This is usually function of the dimensionality of the problem
        maxfuncevals = 100 * dim**2
        minfuncevals = dim + 2

        # Iterate over a set of benchmarks (noise free benchmarks here)
        for f_name in bn.nfreeIDs:

            # Iterate over all the instance of a single problem
            # Rotation, translation, etc.
            for instance in chain(range(1, 6), range(21, 31)):

                # Set the function to be used (problem) in the logger
                e.setfun(*bn.instantiate(f_name, iinstance=instance))

                # Independent restarts until maxfunevals or ftarget is reached
                for restarts in range(0, maxrestarts + 1):
                    if restarts > 0:
                        # Signal the experiment that the algorithm restarted
                        e.restart('independent restart')  # additional info

                    # Run the algorithm with the remaining number of evaluations
                    revals = int(math.ceil(maxfuncevals - e.evaluations))
                    main(e.evalfun, dim, revals, e.ftarget)

                    # Stop if ftarget is reached
                    if e.fbest < e.ftarget or e.evaluations + minfuncevals > maxfuncevals:
                        break

                e.finalizerun()

                print('f%d in %d-D, instance %d: FEs=%d with %d restarts, '
                      'fbest-ftarget=%.4e'
                      % (f_name, dim, instance, e.evaluations, restarts,
                         e.fbest - e.ftarget))
```

Once these experiments are done, the data contained in the `ouput` directory can be used to build the results document. See the BBOB web site on how to build the document.

The complete example : [source code].

CHAPTER

TWO

# EXAMPLES

This section contains some documented examples of common toy problems often encountered in the evolutionary computation community. Note that there are several other examples in the `deap/examples` sub-directory of the framework. These can be used has ground work for implementing your own flavour of evolutionary algorithms.

## 2.1 Genetic Algorithm (GA)

### 2.1.1 One Max Problem

This is the first complete example built with DEAP. It will help new users to overview some of the framework possibilities. The problem is very simple, we search for a `1` filled list individual. This problem is widely used in the evolutionary computation community since it is very simple and it illustrates well the potential of evolutionary algorithms.

#### Setting Things Up

Here we use the one max problem to show how simple can be an evolutionary algorithm with DEAP. The first thing to do is to elaborate the structures of the algorithm. It is pretty obvious in this case that an individual that can contain a series of *booleans* is the most interesting kind of structure available. DEAP does not contain any explicit individual structure since it is simply a container of attributes associated with a fitness. Instead, it provides a convenient method for creating types called the creator.

First of all, we need to import some modules.

```
import random

from deap import base
from deap import creator
from deap import tools
```

##### Creator

The creator is a class factory that can build at run-time new classes that inherit from a base classe. It is very useful since an individual can be any type of container from list to n-ary tree. The creator allows build complex new structures convenient for evolutionary computation.

Let see an example of how to use the creator to setup an individual that contains an array of booleans and a maximizing fitness. We will first need to import the `deap.base` and `deap.creator` modules.

The creator defines at first a single function `create()` that is used to create types. The `create()` function takes at least 2 arguments plus additional optional arguments. The first argument *name* is the actual name of the type that we want to create. The second argument *base* is the base classe that the new type created should inherit from. Finally the optional arguments are members to add to the new type, for example a `fitness` for an individual or `speed` for a particle.

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)
```

The first line creates a maximizing fitness by replacing, in the base type `Fitness`, the pure virtual `weights` attribute by `(1.0,)` that means to maximize a single objective fitness. The second line creates an `Individual` class that inherits the properties of `list` and has a `fitness` attribute of the type `FitnessMax` that was just created.

In this last step, two things are of major importance. The first is the comma following the `1.0` in the weights declaration, even when implementing a single objective fitness, the weights (and values) must be iterable. We won't repeat it enough, in DEAP single objective is a special case of multiobjective. The second important thing is how the just created `FitnessMax` can be used directly as if it was part of the `creator`. This is not magic.

#### Toolbox

A `Toolbox` can be found in the base module. It is intended to store functions with their arguments. The toolbox contains two methods, `register()` and `unregister()` that are used to do the tricks.

```
toolbox = base.Toolbox()
# Attribute generator
toolbox.register("attr_bool", random.randint, 0, 1)
# Structure initializers
toolbox.register("individual", tools.initRepeat, creator.Individual,
    toolbox.attr_bool, 100)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

In this code block we registered a generation function and two initialization functions. The generator `toolbox.attr_bool()` when called, will draw a random integer between 0 and 1. The two initializers for their part will produce respectively initialized individuals and populations.

Again, looking a little closer shows that their is no magic. The registration of tools in the toolbox only associates an *alias* to an already existing function and freezes part of its arguments. This allows to call the alias as if the majority of the (or every) arguments have already been given. For example, the `attr_bool()` generator is made from the `randint()` that takes two arguments *a* and *b*, with `a <= n <= b`, where *n* is the returned integer. Here, we fix `a = 0` and `b = 1`.

It is the same thing for the initializers. This time, the `initRepeat()` is frozen with predefined arguments. In the case of the `individual()` method, `initRepeat()` takes 3 arguments, a class that is a container – here the `Individual` is derived from a `list` –, a function to fill the container and the number of times the function shall be repeated. When called, the `individual()` method will thus return an individual initialized with what would be returned by 100 calls to the `attr_bool()` method. Finally, the `population()` method uses the same paradigm, but we don't fix the number of individuals that it should contain.

### The Evaluation Function

The evaluation function is pretty simple in this case, we need to count the number of ones in the individual. This is done by the following lines of code.

```
def evalOneMax(individual):
    return sum(individual),
```

The returned value must be an iterable of length equal to the number of objectives (weights).

### The Genetic Operators

There is two way of using operators, the first one, is to simply call the function from the `tools` module and the second one is to register them with their argument in a toolbox as for the initialization methods. The most convenient way is to register them in the toolbox, because it allows to easily switch between operators if desired. The toolbox method is also used in the algorithms, see the one max short version for an example.

Registering the operators and their default arguments in the toolbox is done as follow.

```
toolbox.register("evaluate", evalOneMax)
toolbox.register("mate", tools.cxTwoPoints)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
toolbox.register("select", tools.selTournament, tournsize=3)
```

The evaluation is given the alias evaluate. Having a single argument being the individual to evaluate we don't need to fix any, the individual will be given later in the algorithm. The mutation, for its part, needs an argument to be fixed (the independent probability of each attribute to be mutated *indpb*). In the algorithms the `mutate()` function is called with the signature `mutant, = toolbox.mutate(mutant)`. This is the most convenient way because each mutation takes a different number of arguments, having those arguments fixed in the toolbox leave open most of the possibilities to change the mutation (or crossover, or selection, or evaluation) operator later in your researches.

### Evolving the Population

Once the representation and the operators are chosen, we have to define an algorithm. A good habit to take is to define the algorithm inside a function, generally named `main()`.

#### Creating the Population

Before evolving it, we need to instantiate a population. This step is done effortless using the method we registered in the toolbox.

```
def main():
    pop = toolbox.population(n=300)
```

`pop` will be a `list` composed of 300 individuals, *n* is the parameter left open earlier in the toolbox. The next thing to do is to evaluate this brand new population.

```
    # Evaluate the entire population
    fitnesses = list(map(toolbox.evaluate, pop))
    for ind, fit in zip(pop, fitnesses):
        ind.fitness.values = fit
```

We first `map()` the evaluation function to every individual, then assign their respective fitness. Note that the order in `fitnesses` and `population` are the same.

#### The Appeal of Evolution

The evolution of the population is the last thing to accomplish. Let say that we want to evolve for a fixed number of generation `NGEN`, the evolution will then begin with a simple for statement.

```
    # Begin the evolution
    for g in range(NGEN):
        print("-- Generation %i --" % g)
```

Is that simple enough? Lets continue with more complicated things, selecting, mating and mutating the population. The crossover and mutation operators provided within DEAP usually take respectively 2 and 1 individual(s) on input and return 2 and 1 modified individual(s), they also modify inplace these individuals.

In a simple GA, the first step is to select the next generation.

```
# Select the next generation individuals
offspring = toolbox.select(pop, len(pop))
# Clone the selected individuals
offspring = list(map(toolbox.clone, offspring))
```

This step creates an offspring list that is an exact copy of the selected individuals. The `toolbox.clone()` method ensure that we don't own a reference to the individuals but an completely independent instance.

Next, a simple GA would replace the parents by the produced children directly in the population. This is what is done by the following lines of code, where a crossover is applied with probability `CXPB` and a mutation with probability `MUTPB`. The `del` statement simply invalidate the fitness of the modified individuals.

```
# Apply crossover and mutation on the offspring
for child1, child2 in zip(offspring[::2], offspring[1::2]):
    if random.random() < CXPB:
        toolbox.mate(child1, child2)
        del child1.fitness.values
        del child2.fitness.values

for mutant in offspring:
    if random.random() < MUTPB:
        toolbox.mutate(mutant)
        del mutant.fitness.values
```

The population now needs to be re-evaluated, we then apply the evaluation as seen earlier, but this time only on the individuals with an invalid fitness.

```
# Evaluate the individuals with an invalid fitness
invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
fitnesses = map(toolbox.evaluate, invalid_ind)
for ind, fit in zip(invalid_ind, fitnesses):
    ind.fitness.values = fit
```

And finally, last but not least, we replace the old population by the offspring.

```
pop[:] = offspring
```

This is the end of the evolution part, it will continue until the predefined number of generation are accomplished.

Although, some statistics may be gathered on the population, the following lines print the min, max, mean and standard deviation of the population.

```
# Gather all the fitnesses in one list and print the stats
fits = [ind.fitness.values[0] for ind in pop]

length = len(pop)
mean = sum(fits) / length
sum2 = sum(x*x for x in fits)
std = abs(sum2 / length - mean**2)**0.5

print("  Min %s" % min(fits))
print("  Max %s" % max(fits))
print("  Avg %s" % mean)
print("  Std %s" % std)
```

A `Statistics` object has been defined to facilitate how statistics are gathered. It is not presented here so that we can focus on the core and not the gravitating helper objects of DEAP.

The complete example : [source code].

### 2.1.2 One Max Problem: Short Version

The short one max genetic algorithm example is very similar to one max example. The only difference is that it makes use of the `algorithms` module that implements some basic evolutionary algorithms. The initialization are the same so we will skip this phase. The algorithms impemented use specific functions from the toolbox, in this case `evaluate()`, `mate()`, `mutate()` and `select()` must be registered.

```
toolbox.register("evaluate", evalOneMax)
toolbox.register("mate", tools.cxTwoPoints)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
toolbox.register("select", tools.selTournament, tournsize=3)
```

The toolbox is then passed to the algorithm and the algorithm uses the registered function.

```
def main():
    pop = toolbox.population(n=300)
    hof = tools.HallOfFame(1)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", tools.mean)
    stats.register("std", tools.std)
    stats.register("min", min)
    stats.register("max", max)

    algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=40, stats=stats,
                        halloffame=hof, verbose=True)
```

The short GA One max example makes use of a `HallOfFame` in order to keep track of the best individual to appear in the evolution (it keeps it even in the case it extinguishes), and a `Statistics` object to compile the population statistics during the evolution.

Every algorithms from the `algorithms` module can take these objects. Finally, the *verbose* keyword indicate wheter we want the algorithm to output the results after each generation or not.

The complete example : [source code]

### 2.1.3 Knapsack Problem: Inheriting from Set

Again for this example we will use a very simple problem, the 0-1 Knapsack. The purpose of this example is to show the simplicity of DEAP and the ease to inherit from anyting else than a simple list or array.

Many evolutionary algorithm textbooks mention that the best way to have an efficient algorithm to have a representation close the problem. Here, what can be closer to a bag than a set? Lets make our individuals inherit from the `set` class.

```
creator.create("Fitness", base.Fitness, weights=(-1.0, 1.0))
creator.create("Individual", set, fitness=creator.Fitness)
```

That's it. You now have individuals that are in fact sets, they have the usual attribute `fitness`. The fitness is a minimization of the first objective (the weight of the bag) and a maximization of the second objective (the value of the bag). We will now create a dictionary of 100 random items to map the values and weights.

```
# Create the item dictionary: item name is an integer, and value is
# a (weight, value) 2-uple.
items = {}
# Create random items and store them in the items' dictionary.
for i in range(NBR_ITEMS):
    items[i] = (random.randint(1, 10), random.uniform(0, 100))
```

We now need to initialize a population and the individuals therein. For this we will need a `Toolbox` to register our generators since sets can also be created with an input iterable.

```
creator.create("Fitness", base.Fitness, weights=(-1.0, 1.0))
creator.create("Individual", set, fitness=creator.Fitness)

toolbox = base.Toolbox()

# Attribute generator
toolbox.register("attr_item", random.randrange, NBR_ITEMS)
```

Voilà! The *last* thing to do is to define our evaluation function.

```
def evalKnapsack(individual):
    weight = 0.0
    value = 0.0
    for item in individual:
        weight += items[item][0]
        value += items[item][1]
    if len(individual) > MAX_ITEM or weight > MAX_WEIGHT:
        return 10000, 0             # Ensure overweighted bags are dominated
    return weight, value
```

Everything is ready for evolution. Ho no wait, since DEAP's developers are lazy, there is no crossover and mutation operators that can be applied directly on sets. Lets define some. For example, a crossover, producing two child from two parents, could be that the first child is the intersection of the two sets and the second child their absolute difference.

```
def cxSet(ind1, ind2):
    """Apply a crossover operation on input sets. The first child is the
    intersection of the two sets, the second child is the difference of the
    two sets.
    """
    temp = set(ind1)                # Used in order to keep type
    ind1 &= ind2                    # Intersection (inplace)
    ind2 ^= temp                    # Symmetric Difference (inplace)
    return ind1, ind2
```

A mutation operator could randomly add or remove an element from the set input individual.

```
def mutSet(individual):
    """Mutation that pops or add an element."""
    if random.random() < 0.5:
        if len(individual) > 0:     # We cannot pop from an empty set
            individual.remove(random.choice(sorted(tuple(individual))))
    else:
        individual.add(random.randrange(NBR_ITEMS))
    return individual,
```

---

**Note:** The outcome of this mutation is dependent of the python you use. The `set.pop()` function is not consistent between versions of python. See the sources of the actual example for a version that will be stable but more complicated.

We then register these operators in the toolbox. Since it is a multi-objective problem, we have selected the SPEA-II selection scheme : `selSPEA2()`

```
    return individual,

toolbox.register("evaluate", evalKnapsack)
toolbox.register("mate", cxSet)
```

From here, everything left to do is either write the algorithm or use provided in `algorithms`. Here we will use the `eaMuPlusLambda()` algorithm.

```
toolbox.register("select", tools.selNSGA2)
def main():
    random.seed(64)
    NGEN = 50
    MU = 50
    LAMBDA = 100
    CXPB = 0.7
    MUTPB = 0.2

    pop = toolbox.population(n=MU)
    hof = tools.ParetoFront()
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", tools.mean)
    stats.register("std", tools.std)
    stats.register("min", min)
    stats.register("max", max)

    algorithms.eaMuPlusLambda(pop, toolbox, MU, LAMBDA, CXPB, MUTPB, NGEN, stats,
                              halloffame=hof)
```

Finally, a `ParetoFront` may be used to retrieve the best non dominated individuals of the evolution and a `Statistics` object is created for compiling four different statistics over the generations.

The complete example : [source code].

### 2.1.4 Cooperative Coevolution

This example explores cooperative coevolution using DEAP. This tutorial is not as complete as previous examples concerning type creation and other basic stuff. Instead, we cover the concepts of coevolution as they would be applied in DEAP. Assume that if a function from the toolbox is used, it has been properly registered. This example makes a great template for implementing your own coevolutionary algorithm, it is based on the description of cooperative coevolution by [Potter2001].

Coevolution is, in fact, just an extension of how algorithms works in deap. Multiple populations are evolved in turn (or simultaneously on multiple processors) just like in traditional genetic algorithms. The implementation of the coevolution is thus straightforward. A first loop acts for iterating over the populations and a second loop iterates over the individuals of these population.

The first step is to create a bunch of species that will evolve in our population.

```
    species = [toolbox.species() for _ in range(NUM_SPECIES)]
```

Cooperative coevolution works by sending the best individual of each species (called representative) to help in the evaluation of the individuals of the other species. Since the individuals are not yet evaluated we select randomly the individuals that will be in the set of representatives.

```
    representatives = [random.choice(species[i]) for i in range(NUM_SPECIES)]
```

The evaluation function takes a list of individuals to be evaluated including the representatives of the other species and possibly some other arguments. It is not presented in detail for scope reasons, the structure would be, as usual, something like this

```
def evaluate(individuals):
    # Compute the collaboration fitness
    return fitness,
```

The evolution can now begin.

```
    while g < ngen:
        # Initialize a container for the next generation representatives
        next_repr = [None] * len(species)
        for (i, s), j in zip(enumerate(species), species_index):
            # Variate the species individuals
            s = algorithms.varAnd(s, toolbox, 0.6, 1.0)

            # Get the representatives excluding the current species
            r = representatives[:i] + representatives[i+1:]
            for ind in s:
                # Evaluate and set the individual fitness
                ind.fitness.values = toolbox.evaluate([ind] + r, target_set)

            # Select the individuals
            species[i] = toolbox.select(s, len(s))  # Tournament selection
            next_repr[i] = toolbox.get_best(s)[0]   # Best selection
        representatives = next_repr
```

The last lines evolve each species once before sharing their representatives. The common parts of an evolutionary algorithm are all present, variation, evaluation and selection occurs for each species. The species index is simply a unique number identifying each species, it can be used to keep independent statistics on each new species added.

After evolving each species, steps described in [Potter2001] are achieved to add a species and remove useless species on stagnation. These steps are not covered in this example but are present in the complete source code of the coevolution examples.

- Coevolution Base
- Coevolution Niching
- Coevolution Generalization
- Coevolution Adaptation
- Coevolution Final

## 2.2 Genetic Programming (GP)

### 2.2.1 Symbolic Regression Problem: Introduction to GP

Symbolic regression is one of the best known problems in GP (see *Reference*). It is commonly used as a tuning problem for new algorithms, but is also widely used with real-life distributions, where other regression methods may not work. It is conceptually a simple problem, and therefore makes a good introductory example for the GP framework in DEAP.

All symbolic regression problems use an arbitrary data distribution, and try to fit the most accurately the data with a symbolic formula. Usually, a measure like the RMSE (Root Mean Square Error) is used to measure an individual's fitness.

In this example, we use a classical distribution, the quartic polynomial $(x^4+x^3+x^2+x)$, a one-dimension distribution. *20* equidistant points are generated in the range [-1, 1], and are used to evaluate the fitness.

### Creating the primitives set

One of the most crucial aspect of a GP program is the choice of the primitives set. They should make good building blocks for the individuals so the evolution can succeed. In this problem, we use a classical set of primitives, which are basic arithmetic functions :

```
def safeDiv(left, right):
    try:
        return left / right
    except ZeroDivisionError:
        return 0

pset = gp.PrimitiveSet("MAIN", 1)
pset.addPrimitive(operator.add, 2)
pset.addPrimitive(operator.sub, 2)
pset.addPrimitive(operator.mul, 2)
pset.addPrimitive(safeDiv, 2)
pset.addPrimitive(operator.neg, 1)
pset.addPrimitive(math.cos, 1)
pset.addPrimitive(math.sin, 1)
pset.addEphemeralConstant(lambda: random.randint(-1,1))
```

The redefinition of the division is made to protect it against a zero division error (which would crash the program). The other functions are simply a mapping from the Python `operator` module. The number following the function is the *arity* of the primitive, that is the number of entries it takes (this will be used by DTM to build the individuals from the primitives).

On the last line, we declare an `Ephemeral` constant. This is a special terminal type, which does not have a fixed value. When the program appends an ephemeral constant terminal to a tree, the function it contains is executed, and its result is inserted as a constant terminal. In this case, those constant terminals can take the values -1, 0 or 1.

The second argument of `PrimitiveSet` is the number of inputs. Here, as we have only a one dimension regression problem, there is only one input, but it could have as many as you want. By default, those inputs are named "ARGx", where "x" is a number, but you can easily rename them :

```
pset.renameArguments(ARG0='x')
```

### Creator

As any evolutionary program, symbolic regression needs (at least) two object types : an individual containing the genotype and a fitness. We can easily create them with the creator :

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMin, pset=pset)
```

The first line creates the fitness object (this is a minimization problem, so the weight is negative). The *weights* argument must be an iterable of weights, even if there is only one fitness measure. The second line create the individual object itself. Very straightforward, we can see that it will be based upon a tree, to which we add two attributes : a fitness and the primitive set. If, for any reason, the user would want to add any other attribute (for instance, a file in which the

individual will be saved), it would be as easy as adding this attribute of any type to this line. After this declaration, any individual produced will contain those wanted attributes.

### Toolbox

Now, we want to register some parameters specific to the evolution process. In DEAP, this is done through the toolbox :

```
toolbox = base.Toolbox()
toolbox.register("expr", gp.genRamped, pset=pset, min_=1, max_=2)
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.expr)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("lambdify", gp.lambdify, pset=pset)

def evalSymbReg(individual):
    # Transform the tree expression in a callable function
    func = toolbox.lambdify(expr=individual)
    # Evaluate the sum of squared difference between the expression
    # and the real function : x**4 + x**3 + x**2 + x
    values = (x/10. for x in range(-10,10))
    diff_func = lambda x: (func(x)-(x**4 + x**3 + x**2 + x))**2
    diff = sum(map(diff_func, values))
    return diff,

toolbox.register("evaluate", evalSymbReg)
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("mate", gp.cxOnePoint)
toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)
toolbox.register('mutate', gp.mutUniform, expr=toolbox.expr_mut)
```

First, a toolbox instance is created (in some problem types like coevolution, you may consider creating more than one toolbox). Then, we can register any parameters. The first lines register how to create an individual (by calling gp.genRamped with the previously defined primitive set), and how to create the population (by repeating the individual initialization).

We may now introduce the evaluation function, which will receive an individual as input, and return the corresponding fitness. This function uses the *lambdify* function previously defined to transform the individual into its executable form – that is, a program. After that, the evaluation is only simple maths, where the difference between the values produced by the evaluated individual and the real values are squared and summed to compute the RMSE, which is returned as the fitness of the individual.

> **Warning:** Even if the fitness only contains one measure, keep in mind that DEAP stores it as an iterable. Knowing that, you can understand why the evaluation function must return a tuple value (even if it is a 1-tuple) :
>
> ```
> def evalSymbReg(individual):
>     # Transform the tree expression in a callable function
>     func = toolbox.lambdify(expr=individual)
>     # Evaluate the sum of squared difference between the expression
>     # and the real function : x**4 + x**3 + x**2 + x
>     values = (x/10. for x in range(-10,10))
>     diff_func = lambda x: (func(x)-(x**4 + x**3 + x**2 + x))**2
>     diff = sum(map(diff_func, values))
>     return diff,
> ```
>
> Returning only the value would produce strange behaviors and errors, as the selection and stats functions relies on the fact that the fitness is always an iterable.

Afterwards, we register the evaluation function. We also choose the selection method (a tournament of size 3), the mate method (one point crossover with uniform probability over all the nodes), the mutation method (an uniform probability mutation which may append a new full sub-tree to a node).

At this point, any structure with an access to the toolbox instance will also have access to all of those registered parameters. Of course, the user could register other parameters basing on his needs.

#### Statistics

Although optional, statistics are often useful in evolutionary programming. DEAP offers a simple class which can handle most of the "boring work". In this case, we want to keep four measures over the fitness distribution : the mean, the standard deviation, the minimum and the maximum.

```
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("Avg", tools.mean)
stats.register("Std", tools.std)
stats.register("Min", min)
stats.register("Max", max)
```

#### Launching the evolution

At this point, DEAP has all the information needed to begin the evolutionary process, but nothing has been initialized. We can start the evolution by creating the population and then call a pre-made algorithm, as eaSimple :

```
def main():
    pop = toolbox.population(n=300)
    hof = tools.HallOfFame(1)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", tools.mean)
    stats.register("std", tools.std)
    stats.register("min", min)
    stats.register("max", max)

    algorithms.eaSimple(pop, toolbox, 0.5, 0.1, 40, stats, halloffame=hof)

    return pop, stats, hof

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

The hall of fame is a specific structure which contains the *n* best individuals (here, the best one only).

The complete example : [source code].

#### Reference

*John R. Koza, "Genetic Programming: On the Programming of Computers by Means of Natural Selection", MIT Press, 1992, pages 162-169.*

### 2.2.2 Even-Parity Problem

Parity is one of the classical GP problems. The goal is to find a program that produces the value of the Boolean even parity given n independent Boolean inputs. Usually, 6 Boolean inputs are used (Parity-6), and the goal is to match the good parity bit value for each of the $2^6 = 64$ possible entries. The problem can be made harder by increasing the

number of inputs (in the DEAP implementation, this number can easily be tuned, as it is fixed by a constant named PARITY_FANIN_M).

For more information about this problem, see *Reference*.

#### Primitives set used

Parity uses standard Boolean operators as primitives, available in the Python operator module :

```
pset = gp.PrimitiveSet("MAIN", PARITY_FANIN_M, "IN")
pset.addPrimitive(operator.and_, 2)
pset.addPrimitive(operator.or_, 2)
pset.addPrimitive(operator.xor, 2)
pset.addPrimitive(operator.not_, 1)
pset.addTerminal(1)
pset.addTerminal(0)
```

In addition to the *n* inputs, we add two constant terminals, one at 0, one at 1.

---

**Note:** As Python is a dynamic typed language, you can mix Boolean operators and integers without any issue.

---

#### Evaluation function

In this implementation, the fitness of a Parity individual is simply the number of successful cases. Thus, the fitness is maximized, and the maximum value is 64 in the case of a 6 inputs problems.

```
def evalParity(individual):
    func = toolbox.lambdify(expr=individual)
    return sum(func(*in_) == out for in_, out in zip(inputs, outputs)),
```

*inputs* and *outputs* are two pre-generated lists, to speedup the evaluation, mapping a given input vector to the good output bit. The Python `sum()` function works on booleans (false is interpreted as 0 and true as 1), so the evaluation function boils down to sum the number of successful tests : the higher this sum, the better the individual.

#### Conclusion

The other parts of the program are mostly the same as the *Symbolic Regression algorithm*.

The complete example: [source code]

#### Reference

*John R. Koza, "Genetic Programming II: Automatic Discovery of Reusable Programs", MIT Press, 1994, pages 157-199.*

### 2.2.3 Multiplexer 3-8 Problem

The multiplexer problem is another extensively used GP problem. Basically, it trains a program to reproduce the behavior of an electronic multiplexer (mux). Usually, a 3-8 multiplexer is used (3 address entries, from A0 to A2, and 8 data entries, from D0 to D7), but virtually any size of multiplexer can be used.

This problem was first defined by Koza (see *Reference*).

### Primitives set used

The primitive set is almost the same as the set used in *Parity*. Three Boolean operators (and, or and not), imported from `operator`, and a specific if-then-else primitive, which return either its second or third argument depending on the value of the first one.

```
pset = gp.PrimitiveSet("MAIN", MUX_TOTAL_LINES, "IN")
pset.addPrimitive(operator.and_, 2)
pset.addPrimitive(operator.or_, 2)
pset.addPrimitive(operator.not_, 1)
pset.addPrimitive(if_then_else, 3)
pset.addTerminal(1)
pset.addTerminal(0)
```

As usual, we also add two terminals, a Boolean true and a Boolean false.

### Evaluation function

To speed up the evaluation, the computation of the input/output pairs is done at start up, instead of at each evaluation call. This pre-computation also allows to easily tune the multiplexer size, by changing the value of *MUX_SELECT_LINES*.

```
MUX_SELECT_LINES = 3
MUX_IN_LINES = 2 ** MUX_SELECT_LINES
MUX_TOTAL_LINES = MUX_SELECT_LINES + MUX_IN_LINES

# input : [A0 A1 A2 D0 D1 D2 D3 D4 D5 D6 D7] for a 8-3 mux
inputs = [[0] * MUX_TOTAL_LINES for i in range(2 ** MUX_TOTAL_LINES)]
outputs = [None] * (2 ** MUX_TOTAL_LINES)

for i in range(2 ** MUX_TOTAL_LINES):
    value = i
    divisor = 2 ** MUX_TOTAL_LINES
    # Fill the input bits
    for j in range(MUX_TOTAL_LINES):
        divisor /= 2
        if value >= divisor:
            inputs[i][j] = 1
            value -= divisor

    # Determine the corresponding output
    indexOutput = MUX_SELECT_LINES
    for j, k in enumerate(inputs[i][:MUX_SELECT_LINES]):
        indexOutput += k * 2**j
    outputs[i] = inputs[i][indexOutput]
```

After that, the evaluation function is trivial, as we have both inputs and output values. The fitness is then the number of well predicted outputs over the 2048 cases (for a 3-8 multiplexer).

```
def evalMultiplexer(individual):
    func = toolbox.lambdify(expr=individual)
    return sum(func(*in_) == out for in_, out in zip(inputs, outputs)),
```

The complete example: [source code]

#### Reference

*John R. Koza, "Genetic Programming I: On the Programming of Computers by Means of Natural Selection", MIT Press, 1992, pages 170-187.*

### 2.2.4 Artificial Ant Problem

The Artificial Ant problem is a more sophisticated yet classical GP problem, in which the evolved individuals have to control an artificial ant so that it can eat all the food located in a given environment. This example shows how DEAP can easily deal with more complex problems, including an intricate system of functions and resources (including a small simulator).

For more information about this problem, see *Reference*.

#### Primitives set used

We use the standard primitives set for the Artificial Ant problem :

```
pset = gp.PrimitiveSet("MAIN", 0)
pset.addPrimitive(ant.if_food_ahead, 2)
pset.addPrimitive(prog2, 2)
pset.addPrimitive(prog3, 3)
pset.addTerminal(ant.move_forward)
pset.addTerminal(ant.turn_left)
pset.addTerminal(ant.turn_right)
```

- `if_food_ahead` is a primitive which executes its first argument if there is food in front of the ant; else, it executes its second argument.
- `prog2()` and `prog3()` are the equivalent of the lisp PROGN2 and PROGN3 functions. They execute their children in order, from the first to the last. For instance, prog2 will first execute its first argument, then its second.
- `move_forward()` makes the artificial ant move one front. This is a terminal.
- `turn_right()` and `turn_left()` makes the artificial ant turning clockwise and counter-clockwise, without changing its position. Those are also terminals.

---

**Note:** There is no external input as in symbolic regression or parity.

---

Although those functions are obviously not already built-in in Python, it is very easy to define them :

```
def progn(*args):
    for arg in args:
        arg()

def prog2(out1, out2):
    return partial(progn,out1,out2)

def prog3(out1, out2, out3):
    return partial(progn,out1,out2,out3)

def if_then_else(condition, out1, out2):
    out1() if condition() else out2()

class AntSimulator(object):
```

```
    def if_food_ahead(self, out1, out2):
        return partial(if_then_else, self.sense_food, out1, out2)
```

Partial functions are a powerful feature of Python which allow to create functions on the fly. For more detailed information, please refer to the Python documentation of `functools.partial()`.

### Evaluation function

The evaluation function use an instance of a simulator class to evaluate the individual. Each individual is given 600 moves on the simulator map (obtained from an external file). The fitness of each individual corresponds to the number of pieces of food picked up. In this example, we are using a classical trail, the *Santa Fe trail*, in which there is 89 pieces of food. Therefore, a perfect individual would achieve a fitness of 89.

```
def evalArtificialAnt(individual):
    # Transform the tree expression to functionnal Python code
    routine = gp.evaluate(individual, pset)
    # Run the generated routine
    ant.run(routine)
    return ant.eaten,
```

Where *ant* is the instance of the simulator used. The `evaluate()` function is a convenience one provided by DEAP and returning an executable Python program from a GP individual and its primitives function set.

### Complete example

Except for the simulator code (about 75 lines), the code does not fundamentally differ from the *Symbolic Regression example*. Note that as the problem is harder, improving the selection pressure by increasing the size of the tournament to 7 allows to achieve better performance.

```
from deap import base
from deap import creator
from deap import tools
from deap import gp

def progn(*args):
    for arg in args:
        arg()

def prog2(out1, out2):
    return partial(progn,out1,out2)

def prog3(out1, out2, out3):
    return partial(progn,out1,out2,out3)

def if_then_else(condition, out1, out2):
    out1() if condition() else out2()

class AntSimulator(object):
    direction = ["north","east","south","west"]
    dir_row = [1, 0, -1, 0]
    dir_col = [0, 1, 0, -1]

    def __init__(self, max_moves):
        self.max_moves = max_moves
        self.moves = 0
        self.eaten = 0
```

```
        self.routine = None

    def _reset(self):
        self.row = self.row_start
        self.col = self.col_start
        self.dir = 1
        self.moves = 0
        self.eaten = 0
        self.matrix_exc = copy.deepcopy(self.matrix)

    @property
    def position(self):
        return (self.row, self.col, self.direction[self.dir])

    def turn_left(self):
        if self.moves < self.max_moves:
            self.moves += 1
            self.dir = (self.dir - 1) % 4

    def turn_right(self):
        if self.moves < self.max_moves:
            self.moves += 1
            self.dir = (self.dir + 1) % 4

    def move_forward(self):
        if self.moves < self.max_moves:
            self.moves += 1
            self.row = (self.row + self.dir_row[self.dir]) % self.matrix_row
            self.col = (self.col + self.dir_col[self.dir]) % self.matrix_col
            if self.matrix_exc[self.row][self.col] == "food":
                self.eaten += 1
            self.matrix_exc[self.row][self.col] = "passed"

    def sense_food(self):
        ahead_row = (self.row + self.dir_row[self.dir]) % self.matrix_row
        ahead_col = (self.col + self.dir_col[self.dir]) % self.matrix_col
        return self.matrix_exc[ahead_row][ahead_col] == "food"

    def if_food_ahead(self, out1, out2):
        return partial(if_then_else, self.sense_food, out1, out2)

    def run(self,routine):
        self._reset()
        while self.moves < self.max_moves:
            routine()

    def parse_matrix(self, matrix):
        self.matrix = list()
        for i, line in enumerate(matrix):
            self.matrix.append(list())
            for j, col in enumerate(line):
                if col == "#":
                    self.matrix[-1].append("food")
                elif col == ".":
                    self.matrix[-1].append("empty")
                elif col == "S":
                    self.matrix[-1].append("empty")
                    self.row_start = self.row = i
```

```
                    self.col_start = self.col = j
                    self.dir = 1
        self.matrix_row = len(self.matrix)
        self.matrix_col = len(self.matrix[0])
        self.matrix_exc = copy.deepcopy(self.matrix)

ant = AntSimulator(600)

pset = gp.PrimitiveSet("MAIN", 0)
pset.addPrimitive(ant.if_food_ahead, 2)
pset.addPrimitive(prog2, 2)
pset.addPrimitive(prog3, 3)
pset.addTerminal(ant.move_forward)
pset.addTerminal(ant.turn_left)
pset.addTerminal(ant.turn_right)

creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMax, pset=pset)

toolbox = base.Toolbox()

# Attribute generator
toolbox.register("expr_init", gp.genFull, pset=pset, min_=1, max_=2)

# Structure initializers
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.expr_init)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

def evalArtificialAnt(individual):
    # Transform the tree expression to functionnal Python code
    routine = gp.evaluate(individual, pset)
    # Run the generated routine
    ant.run(routine)
    return ant.eaten,

toolbox.register("evaluate", evalArtificialAnt)
toolbox.register("select", tools.selTournament, tournsize=7)
toolbox.register("mate", gp.cxOnePoint)
toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)
toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut)

def main():
    random.seed(69)

    trail_file = open("ant/santafe_trail.txt")
    ant.parse_matrix(trail_file)

    pop = toolbox.population(n=300)
    hof = tools.HallOfFame(1)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", tools.mean)
    stats.register("std", tools.std)
    stats.register("min", min)
    stats.register("max", max)

    algorithms.eaSimple(pop, toolbox, 0.5, 0.2, 40, stats, halloffame=hof)

    return pop, hof, stats
```

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

---

**Note:** The import of the Python standard library modules are not shown.

---

[source code]

#### Reference

*John R. Koza, "Genetic Programming I: On the Programming of Computers by Means of Natural Selection", MIT Press, 1992, pages 147-161.*

### 2.2.5 Spambase Problem: Strongly Typed GP

This problem is a classification example using STGP (Strongly Typed Genetic Programming). The evolved programs work on floating-point values AND Booleans values. The programs must return a Boolean value which must be true if e-mail is spam, and false otherwise. It uses a base of emails (saved in *spambase.csv*, see *Reference*), from which it randomly chooses 400 emails to evaluate each individual.

> **Warning:** Don't expect too much from this program as it is quite basic and not oriented toward performance. It is given to illustrate the use of strongly-typed GP with DEAP. From a machine learning perspective, it is mainly wrong.

#### Primitives set

Strongly-typed GP is a more generic GP where each primitive, in addition to have an arity and a corresponding function, has also a specific return type and specific parameter(s) type. In this way, each primitive is someway describe as a pure C function, where each parameter has to be one of the good type, and where the return value type is specified before run time.

---

**Note:** Actually, when the user does not specify return or parameters type, a default type is selected by DEAP. On standard GP, because all the primitives use this default type, this behaves as there was no type requirement.

---

We define a typed primitive set almost the same way than a normal one, but we have to specify the types used.

```
pset = gp.PrimitiveSetTyped("MAIN", itertools.repeat("float", 57), "bool", "IN")

# boolean operators
pset.addPrimitive(operator.and_, ["bool", "bool"], "bool")
pset.addPrimitive(operator.or_, ["bool", "bool"], "bool")
pset.addPrimitive(operator.not_, ["bool"], "bool")

# floating point operators
# Define a safe division function
def safeDiv(left, right):
    try: return left / right
    except ZeroDivisionError: return 0

pset.addPrimitive(operator.add, ["float","float"], "float")
pset.addPrimitive(operator.sub, ["float","float"], "float")
```

```
pset.addPrimitive(operator.mul, ["float","float"], "float")
pset.addPrimitive(safeDiv, ["float","float"], "float")

# logic operators
# Define a new if-then-else function
def if_then_else(input, output1, output2):
    if input: return output1
    else: return output2

pset.addPrimitive(operator.lt, ["float", "float"], "bool")
pset.addPrimitive(operator.eq, ["float", "float"], "bool")
pset.addPrimitive(if_then_else, ["bool", "float", "float"], "float")

# terminals
pset.addEphemeralConstant(lambda: random.random() * 100, "float")
pset.addTerminal(0, "bool")
pset.addTerminal(1, "bool")
```

On the first line, we see the declaration of a typed primitive set with `PrimitiveSetTyped`. The first argument remains the set name, but the next ones are the type of the entries (in this case, we have 57 float entries and one Boolean output; we could have written *float* 57 times, but it is fairly quicker and more understandable to use the `itertools.repeat()` function). The last argument remains the entries prefix.

After that, we define the primitives themselves. The definition of a typed primitive has two additional parameters : a list containing the parameters type, in order, and the return type.

---

**Note:** The types specified do not have to be real Python or C types. In the above example, we may rename "float" in "type1" and "bool" in "type2" without any issue. For the same reason, DEAP nor Python actually check if a given parameter has the good type.

---

The terminals set is then filled, with at least one terminal of each type, and that is for the primitive set declaration.

### Evaluation function

The evaluation function is very simple : it picks 400 mails at random in the spam database, and then checks if the prediction made by the individual matches the expected Boolean output. The count of well predicted emails is returned as the fitness of the individual (which is so, at most, 400).

```
def evalSpambase(individual):
    # Transform the tree expression in a callable function
    func = toolbox.lambdify(expr=individual)
    # Randomly sample 400 mails in the spam database
    spam_samp = random.sample(spam, 400)
    # Evaluate the sum of correctly identified mail as spam
    result = sum(bool(func(*mail[:57])) is bool(mail[57]) for mail in spam_samp)
    return result,
```

### Toolbox

The toolbox used is very similar to the one presented in the symbolic regression example, but notice that we now use specific STGP operators for crossovers and mutations :

```
toolbox.register("evaluate", evalSpambase)
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("mate", gp.cxOnePoint)
```

```
toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)
toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut)
```

#### Conclusion

Although it does not really differ from the other problems, it is interesting to note how Python can decrease the programming time. Indeed, the spam database is in csv form : with many frameworks, you would have to manually read it, or use a non-standard library, but with Python, you can use the built-in module `csv` and, within 2 lines, it is done! The data is now in the matrix *spam* and can easily be used through all the program :

The complete example: [source code]

#### Reference

Data are from the Machine learning repository, http://www.ics.uci.edu/~mlearn/MLRepository.html

## 2.3 Evolution Strategy (ES)

### 2.3.1 Evolution Strategies Basics

Evolution strategies are special types of evolutionary computation algorithms where the mutation strength is learnt during the evolution. A first type of strategy (endogenous) includes directly the mutation strength for each attribute of an individual inside the individual. This mutation strength is subject to evolution similarly to the individual in a classic genetic algorithm. For more details, [Beyer2002] presents a very good introduction to evolution strategies.

In order to have this kind of evolution we'll need a type of individual that contains a `strategy` attribute. We'll also minimize the objective function, which gives the following classes creation.

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", array.array, typecode="d", fitness=creator.FitnessMin, strategy=None)
creator.create("Strategy", array.array, typecode="d")
```

The initialization function for an evolution strategy is not defined by DEAP. The following generation function takes as argument the class of individual to instantiate, *icls*. It also takes the class of strategy to use as strategy, *scls*. The next arguments are the minimum and maximum values for the individual and strategy attributes. The strategy is added in the `strategy` member of the returned individual.

```
def generateES(icls, scls, size, imin, imax, smin, smax):
    ind = icls(random.uniform(imin, imax) for _ in range(size))
    ind.strategy = scls(random.uniform(smin, smax) for _ in range(size))
    return ind
```

This generation function is registered in the toolbox like any other initializer.

```
toolbox.register("individual", generateES, creator.Individual, creator.Strategy,
    IND_SIZE, MIN_VALUE, MAX_VALUE, MIN_STRATEGY, MAX_STRATEGY)
```

The strategy controls the standard deviation of the mutation. It is common to have a lower bound on the values so that the algorithm don't fall in exploitation only. This lower bound is added to the variation operator by the following decorator.

```
def checkStrategy(minstrategy):
    def decorator(func):
        def wrappper(*args, **kargs):
```

```
            children = func(*args, **kargs)
            for child in children:
                for i, s in enumerate(child.strategy):
                    if s < minstrategy:
                        child.strategy[i] = minstrategy
            return children
        return wrappper
    return decorator
```

The variation operators are decorated via the `decorate()` method of the toolbox and the evaluation function is taken from the `benchmarks` module.

```
toolbox.decorate("mate", checkStrategy(MIN_STRATEGY))
toolbox.decorate("mutate", checkStrategy(MIN_STRATEGY))

toolbox.register("evaluate", benchmarks.sphere)
```

From here, everything left to do is either write the algorithm or use one provided in `algorithms`. Here we will use the `eaMuCommaLambda()` algorithm.

```
def main():
    MU, LAMBDA = 10, 100
    pop = toolbox.population(n=MU)
    hof = tools.HallOfFame(1)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", tools.mean)
    stats.register("std", tools.std)
    stats.register("min", min)
    stats.register("max", max)

    algorithms.eaMuCommaLambda(pop, toolbox, mu=MU, lambda_=LAMBDA,
                               cxpb=0.6, mutpb=0.3, ngen=500,
                               stats=stats, halloffame=hof)

    return pop, stats, hof
```

The complete example : [source code].

### 2.3.2 One Fifth Rule

Soon!

### 2.3.3 Covariance Matrix Adaptation Evolution Strategy

The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [Hansen2001] implemented in the `cma` module makes use of the generate-update paradigm where a population is generated from a strategy and the strategy is updated from the population. It is then straight forward to use it for continuous problem optimization.

As usual the first thing to do is to create the types and as usual we'll need a minimizing fitness and an individual that is a `list`. A toolbox is then created with the desired evaluation function.

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()
toolbox.register("evaluate", benchmarks.rastrigin)
```

Then, it does not get any harder. Once a `Strategy` is instantiated, its `generate()` and `update()` methods are registered in the toolbox for uses in the `eaGenerateUpdate()` algorithm. The `generate()` method is set to produce the created `Individual` class. The random number generator from numpy is seeded because the `cma` module draws all its number from it.

```
def main():
    numpy.random.seed(128)

    strategy = cma.Strategy(centroid=[5.0]*N, sigma=5.0, lambda_=20*N)
    toolbox.register("generate", strategy.generate, creator.Individual)
    toolbox.register("update", strategy.update)

    hof = tools.HallOfFame(1)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", tools.mean)
    stats.register("std", tools.std)
    stats.register("min", min)
    stats.register("max", max)

    algorithms.eaGenerateUpdate(toolbox, ngen=250, stats=stats, halloffame=hof)
```

### 2.3.4 Controlling the Stopping Criteria: BI-POP CMA-ES

A variant of the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [Hansen2001] implies to control very specifically the termination criteria in the generational loop. This can be achieved by writing the algorithm partially invoking manually the `generate()` and `update()` inside a loop with specific stop criteria. In fact, the BI-POP CMA-ES [Hansen2009] has 9 different stop criteria, which are used to control the independent restarts, with different population sizes, of a standard CMA-ES.

As usual, the first thing to do is to create the types and as usual, we'll need a minimizing fitness and an individual that is a `list`.

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)
```

The main function includes the setting of some parameters, namely the number of increasing population restarts and the initial sigma value. Then, the instanciation of the `Toolbox` is done in the main function because it will change with the restarts. Next are initialized the `HallOfFame` and the `Statistics` objects.

```
def main(verbose=True):
    NRESTARTS = 10  # Initialization + 9 I-POP restarts
    SIGMA0 = 2.0    # 1/5th of the domain [-5 5]

    toolbox = base.Toolbox()
    toolbox.register("evaluate", benchmarks.rastrigin)

    halloffame = tools.HallOfFame(1)

    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", tools.mean)
    stats.register("std", tools.std)
    stats.register("min", min)
    stats.register("max", max)
```

Then the first loop controlling the restart is set up. It encapsulates the generational loop with its many stop criteria. The content of this last loop is simply the generate-update loop as presented in the `deap.algorithms.eaGenerateUpdate()` function.

```
    while i < (NRESTARTS + nsmallpopruns):

        strategy = cma.Strategy(centroid=numpy.random.uniform(-4, 4, N), sigma=sigma, lambda_=lambda_
        toolbox.register("generate", strategy.generate, creator.Individual)
        toolbox.register("update", strategy.update)

        conditions = {"MaxIter" : False, "TolHistFun" : False, "EqualFunVals" : False,
                      "TolX" : False, "TolUpSigma" : False, "Stagnation" : False,
                      "ConditionCov" : False, "NoEffectAxis" : False, "NoEffectCoor" : False}

        while not any(conditions.values()):
            # Generate a new population
            population = toolbox.generate()

            # Evaluate the individuals
            fitnesses = toolbox.map(toolbox.evaluate, population)
            for ind, fit in zip(population, fitnesses):
                ind.fitness.values = fit

            # Update the strategy with the evaluated individuals
            toolbox.update(population)

            if t >= MAXITER:
                # The maximum number of iteration per CMA-ES ran
                conditions["MaxIter"] = True

            if (len(stats.min) > i and not len(stats.min[i]) < TOLHISTFUN_ITER) and \
               max(stats.min[i][-TOLHISTFUN_ITER:])[0] - min(stats.min[i][-TOLHISTFUN_ITER:])[0] < TO
                # The range of the best values is smaller than the threshold
                conditions["TolHistFun"] = True

            if t > N and sum(equalfunvalues[-N:]) / float(N) > EQUALFUNVALS:
                # In 1/3rd of the last N iterations the best and k'th best solutions are equal
                conditions["EqualFunVals"] = True

            if all(strategy.pc < TOLX) and all(numpy.sqrt(numpy.diag(strategy.C)) < TOLX):
                # All components of pc and sqrt(diag(C)) are smaller than the threshold
                conditions["TolX"] = True

            if strategy.sigma / sigma > strategy.diagD[-1]**2 * TOLUPSIGMA:
                # The sigma ratio is bigger than a threshold
                conditions["TolUpSigma"] = True

            if len(bestvalues) > STAGNATION_ITER and len(medianvalues) > STAGNATION_ITER and \
               numpy.median(bestvalues[-20:]) >= numpy.median(bestvalues[-STAGNATION_ITER:-STAGNATION
               numpy.median(medianvalues[-20:]) >= numpy.median(medianvalues[-STAGNATION_ITER:-STAGNA
                # Stagnation occured
                conditions["Stagnation"] = True

            if strategy.cond > 10**14:
                # The condition number is bigger than a threshold
                conditions["ConditionCov"] = True

            if all(strategy.centroid == strategy.centroid + 0.1 * strategy.sigma * strategy.diagD[-NC
                # The coordinate axis std is too low
                conditions["NoEffectAxis"] = True

            if any(strategy.centroid == strategy.centroid + 0.2 * strategy.sigma * numpy.diag(strateg
```

```
                # The main axis std has no effect
                conditions["NoEffectCoor"] = True

        i += 1
```

Some variables have been omited for clarity, refer to the complete example for more details [source code].

### 2.3.5 Plotting Important Data: Visualizing the CMA Strategy

With this example we show one technique for plotting the data of an evolution. As developpers of DEAP we cannot make a choice on what data is important to plot and this part is left to the user. Although, plotting would all occur the same way. First the data is gathered during the evolution and at the end the figures are created from the data. This model is the simplest possible. One could also write all data to a file and read those file again to plot the figures. This later model would be more fault tolerant as if the evolution does not terminate normally, the figures could still be plotted. But, we want to keep this example as simpe as possible and thus we will present the former model.

#### Evolution Loop

The beginning of this example is exactly the same as the *CMA-ES* example. The general evolution loop of function `eaGenerateUpdate()` is somewhat insufficient for our purpose. We need to gather the required data on each generation. So instead of using the `eaGenerateUpdate()` function, we'll develop it to get a grip on what is recorded. First, we'll created objects to record our data. Here we want to plot, in addition to what the `Statistics` and `HallOfFame` objects contain, the step size, the axis ratio and the major axis of the covariace matrix, the best value so far, the best coordinates so far and the standard deviation of the all coordinates at each generation.

```
sigma = numpy.ndarray((NGEN,1))
axis_ratio = numpy.ndarray((NGEN,1))
diagD = numpy.ndarray((NGEN,N))
fbest = numpy.ndarray((NGEN,1))
best = numpy.ndarray((NGEN,N))
std = numpy.ndarray((NGEN,N))
```

Once the objects are created, the evolution loop, based on a generational stopping criterion, calls repeatedly the `generate()`, `evaluate()` and `update()` methods registered in the toolbox.

```
for gen in range(NGEN):
    # Generate a new population
    population = toolbox.generate()
    # Evaluate the individuals
    fitnesses = toolbox.map(toolbox.evaluate, population)
    for ind, fit in zip(population, fitnesses):
        ind.fitness.values = fit

    # Update the strategy with the evaluated individuals
    toolbox.update(population)
```

Then, the previoulsy created objects start to play their role. The according data is recorded in each object on each generation.

```
        halloffame.update(population)
        stats.update(population)

        # Save more data along the evolution for latter plotting
        # diagD is sorted and sqrooted in the update method
        sigma[gen] = strategy.sigma
        axis_ratio[gen] = max(strategy.diagD)**2/min(strategy.diagD)**2
```

```
        diagD[gen, :N] = strategy.diagD**2
        fbest[gen] = halloffame[0].fitness.values
        best[gen, :N] = halloffame[0]
```

Now that the data is recorded the only thing left to do is to plot it. We'll use matplotlib to generate the graphics from the recorded data.

```
    # The x-axis will be the number of evaluations
    x = list(range(0, strategy.lambda_ * NGEN, strategy.lambda_))

    plt.figure()
    plt.subplot(2, 2, 1)
    plt.semilogy(x, stats.avg[0], "--b")
    plt.semilogy(x, stats.max[0], "--b")
    plt.semilogy(x, stats.min[0], "-b")
    plt.semilogy(x, fbest, "-c")
    plt.semilogy(x, sigma, "-g")
    plt.semilogy(x, axis_ratio, "-r")
    plt.grid(True)
    plt.title("blue: f-values, green: sigma, red: axis ratio")

    plt.subplot(2, 2, 2)
    plt.plot(x, best)
    plt.grid(True)
    plt.title("Object Variables")

    plt.subplot(2, 2, 3)
    plt.semilogy(x, diagD)
    plt.grid(True)
    plt.title("Scaling (All Main Axes)")

    plt.subplot(2, 2, 4)
    plt.semilogy(x, std)
    plt.grid(True)
    plt.title("Standard Deviations in All Coordinates")
```

Which gives the following result.

## 2.4 Particle Swarm Optimization (PSO)

### 2.4.1 Particle Swarm Optimization Basics

The implementation presented here is the original PSO algorithm as presented in [Poli2007]. From Wikipedia definition of PSO

> *PSO optimizes a problem by having a population of candidate solutions, here dubbed particles, and moving these particles around in the search-space according to simple mathematical formulae. The movements of the particles are guided by the best found positions in the search-space which are updated as better positions are found by the particles.*

#### Modules

Before writing functions and algorithms, we need to import some module from the standard library and from DEAP.

```
import operator
import random


from deap import base
from deap import benchmarks
from deap import creator
from deap import tools
```

### Representation

The particle's goal is to maximize the return value of the fonction at its position.

PSO particles are essentially described as a positions in a search-space of D dimensions. Each particle also has a vector representing the speed of the particle in each dimension. Finally, each particle keeps a reference to the best state in which it has been so far.

This translates in DEAP by the following two lines of code :

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Particle", list, fitness=creator.FitnessMax, speed=list,
    smin=None, smax=None, best=None)
```

Here we create two new objects in the `creator` space. First, we create a `FitnessMax` object, and we specify the `weights` to be `(1.0,)`, this means we want to maximise the value of the fitness of our particles. The second object we create represent our particle. We defined it as a `list` to which we add five attributes. The first attribute is the fitness of the particle, the second is the speed of the particle which is also goind to be a list, the third and fourth are the limit of the speed value, and the fifth attribute will be a reference to a copy of the best state the particle has been so far. Since the particle has no final state until at has been evaluated, the reference is set to `None`. The speed limits are also set to `None` to allow configuration via the function `generate()` presented in the next section.

### Operators

PSO original algorithm use three operators : initializer, updater and evaluator. The initialization consist in generating a random position and a random speed for a particle. The next function create a particle and initialize its attributes, except for the attribute `best`, which will be set only after evaluation :

```
def generate(size, pmin, pmax, smin, smax):
    part = creator.Particle(random.uniform(pmin, pmax) for _ in range(size))
    part.speed = [random.uniform(smin, smax) for _ in range(size)]
    part.smin = smin
    part.smax = smax
    return part
```

The function `updateParticle()` first computes the speed, then limits the speed values between `smin` and `smax`, and finally computes the new particle position.

```
def updateParticle(part, best, phi1, phi2):
    u1 = (random.uniform(0, phi1) for _ in range(len(part)))
    u2 = (random.uniform(0, phi2) for _ in range(len(part)))
    v_u1 = map(operator.mul, u1, map(operator.sub, part.best, part))
    v_u2 = map(operator.mul, u2, map(operator.sub, best, part))
    part.speed = list(map(operator.add, part.speed, map(operator.add, v_u1, v_u2)))
    for i, speed in enumerate(part.speed):
        if speed < part.smin:
            part.speed[i] = part.smin
```

```
        elif speed > part.smax:
            part.speed[i] = part.smax
    part[:] = list(map(operator.add, part, part.speed))
```

The operators are registered in the toolbox with their parameters. The particle value at the beginning are in the range `[-100, 100]` (`pmin` and `pmax`), and the speed is limited in the range `[-50, 50]` through all the evolution.

The evaluation function `h1()` is from [Knoek2003]. The function is already defined in the benchmarks module, so we can register it directly.

```
toolbox = base.Toolbox()
toolbox.register("particle", generate, size=2, pmin=-6, pmax=6, smin=-3, smax=3)
toolbox.register("population", tools.initRepeat, list, toolbox.particle)
toolbox.register("update", updateParticle, phi1=2.0, phi2=2.0)
toolbox.register("evaluate", benchmarks.h1)
```

### Algorithm

Once the operators are registered in the toolbox, we can fire up the algorithm by firstly creating a new population, and then apply the original PSO algorithm. The variable *best* contains the best particle ever found (it known as gbest in the original algorithm).

```
def main():
    pop = toolbox.population(n=5)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("Avg", tools.mean)
    stats.register("Std", tools.std)
    stats.register("Min", min)
    stats.register("Max", max)

    column_names = ["gen", "evals"]
    column_names.extend(stats.functions.keys())
    logger = tools.EvolutionLogger(column_names)
    logger.logHeader()

    GEN = 1000
    best = None

    for g in range(GEN):
        for part in pop:
            part.fitness.values = toolbox.evaluate(part)
            if not part.best or part.best.fitness < part.fitness:
                part.best = creator.Particle(part)
                part.best.fitness.values = part.fitness.values
            if not best or best.fitness < part.fitness:
                best = creator.Particle(part)
                best.fitness.values = part.fitness.values
        for part in pop:
            toolbox.update(part, best)

        # Gather all the fitnesses in one list and print the stats
        stats.update(pop)
        logger.logGeneration(gen=g, evals=len(pop), stats=stats)

    return pop, stats, best
```

#### Conclusion

The full PSO basic example can be found here : [source code].

This is a video of the algorithm in action, plotted with matplotlib. The red dot represents the best solution found so far.

#### References

## 2.5 Estimation of Distribution Algorithms (EDA)

### 2.5.1 Making Your Own Strategy : A Simple EDA

As seen in the *Covariance Matrix Adaptation Evolution Strategy* example, the `eaGenerateUpdate()` algorithm is suitable for algorithms learning the problem distribution from the population. Here we'll cover how to implement a strategy that generates individuals based on an updated sampling function learnt from the sampled population.

#### Estimation of distribution

The basic concept concept behind EDA is to sample $\lambda$ individuals with a certain distribution and estimate the problem distribution from the $\mu$ best individuals. This really simple concept adhere to the generate-update logic. The strategy contains a random number generator which is adapted from the population. The following `EDA` class do just that.

```
class EDA(object):
    def __init__(self, centroid, sigma, mu, lambda_):
        self.dim = len(centroid)
        self.loc = numpy.array(centroid)
        self.sigma = numpy.array(sigma)
        self.lambda_ = lambda_
        self.mu = mu

    def generate(self, ind_init):
        # Generate lambda_ individuals and put them into the provided class
        arz = self.sigma * numpy.random.randn(self.lambda_, self.dim) + self.loc
        return list(map(ind_init, arz))

    def update(self, population):
        # Sort individuals so the best is first
        sorted_pop = sorted(population, key=attrgetter("fitness"), reverse=True)

        # Compute the average of the mu best individuals
        z = sorted_pop[:self.mu] - self.loc
        avg = numpy.mean(z, axis=0)

        # Adjust variances of the distribution
        self.sigma = numpy.sqrt(numpy.sum((z - avg)**2, axis=0) / (self.mu - 1.0))
        self.loc = self.loc + avg
```

A normal random number generator is initialized with a certain mean (*centroid*) and standard deviation (*sigma*) for each dimension. The `generate()` method uses numpy to generate *lambda_* sequences in *dim* dimensions, then the sequences are used to initialize individuals of class given in the *ind_init* argument. Finally, the `update()` computes the average (centre) of the *mu* best individuals and estimates the variance over all attributes of each individual. Once `update()` is called the distributions parameters are changed and a new population can be generated.

#### Objects Needed

Two classes are needed, a minimization fitness and a individual that will combine the fitness and the real values. Moreover, we will use `numpy.ndarray` as base class for our individuals.

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", numpy.ndarray, fitness=creator.FitnessMin)
```

#### Operators

The `eaGenerateUpdate()` algorithm requires to set in a toolbox an evaluation function, an generation method and an update method. We will use the method of an initialized `EDA`. For the generate method, we set the class that the individuals are transferred in to our `Individual` class containing a fitness.

```
def main():
    N, LAMBDA = 30, 1000
    MU = int(LAMBDA/4)
    strategy = EDA(centroid=[5.0]*N, sigma=[5.0]*N, mu=MU, lambda_=LAMBDA)

    toolbox = base.Toolbox()
    toolbox.register("evaluate", benchmarks.rastrigin)
    toolbox.register("generate", strategy.generate, creator.Individual)
    toolbox.register("update", strategy.update)

    hof = tools.HallOfFame(1)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", tools.mean)
    stats.register("std", tools.std)
    stats.register("min", min)
    stats.register("max", max)

    algorithms.eaGenerateUpdate(toolbox, ngen=150, stats=stats, halloffame=hof)

    return hof[0].fitness.values[0]
```

The complete example : [source code].

## 2.6 Distributed Task Manager (DTM)

### 2.6.1 A Pi Calculation with DTM

A simple yet interesting use of DTM is the calculation of $\pi$ with a Monte Carlo approach. This approach is quite straightforward : if you randomly throw $n$ darts on a unit square, approximately $\frac{n*\pi}{4}$ will be inside a quadrant delimited by (0,1) and (1,0). Therefore, if a huge quantity of darts are thrown, one could estimate $\pi$ simply by computing the ratio between the number of darts inside and outside the quadrant. A comprehensive explanation of the algorithm can be found here

---

**Note:** This example is intended to show a simple parallelization of an actual algorithm. It should not be taken as a good $\pi$ calculation algorithm (it is not).

---

A possible serial Python code reads as follow :

```
from random import random
from math import hypot

def test(tries):
    # Each run of this function makes some tries
    # and return the number of darts inside the quadrant (r < 1)
    return sum(hypot(random(), random()) < 1 for i in xrange(tries))

def calcPi(n, t):
    expr = (test(t) for i in range(n))
    pi2 = 4. * sum(expr) / (n*t)
    print("pi = " + str(pi2))
    return pi2

piVal = calcPi(1000, 10000)
```

With DTM, you can now take advantage of the parallelization, and distribute the calls to the function *test()*. There are many ways to do so, but a mere one is to use `repeat()`, which repeats a function an arbitrary number of times, and returns a results list. In this case, the program may look like this :

```
from math import hypot
from random import random
from deap import dtm

def test(tries):
    # Each run of this function makes some tries
    # and return the number of darts inside the quadrant (r < 1)
    return sum(hypot(random(), random()) < 1 for i in xrange(tries))

def calcPi(n, t):
    expr = dtm.repeat(test, n, t)
    pi2 = 4. * sum(expr) / (n*t)
    print("pi = " + str(pi2))
    return pi2

piVal = dtm.start(calcPi, 1000, 10000)
```

And so, without any major changes (and not at all in the *test()* function), this computation can be distributed.

### 2.6.2 DTM + EAP = DEAP : a Distributed Evolution

As part of the DEAP framework, EAP offers an easy DTM integration. As the EAP algorithms use a map function stored in the toolbox to spawn the individuals evaluations (by default, this is simply the traditional Python `map()`), the parallelization can be made very easily, by replacing the map operator in the toolbox :

```
from deap import dtm
tools.register("map", dtm.map)
```

Thereafter, ensure that your main code is in enclosed in a Python function (for instance, main), and just add the last line :

```
dtm.start(main)
```

For instance, take a look at the short version of the onemax. This is how it may be parallelized :

```
from deap import dtm

creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

```
creator.create("Individual", array.array, typecode='b', fitness=creator.FitnessMax)

toolbox = base.Toolbox()

# Attribute generator
toolbox.register("attr_bool", random.randint, 0, 1)

# Structure initializers
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_bool, 100)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

def evalOneMax(individual):
    return sum(individual),

toolbox.register("evaluate", evalOneMax)
toolbox.register("mate", tools.cxTwoPoints)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("map", dtm.map)

def main():
    random.seed(64)

    pop = toolbox.population(n=300)
    hof = tools.HallOfFame(1)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("Avg", tools.mean)
    stats.register("Std", tools.std)
    stats.register("Min", min)
    stats.register("Max", max)

    algorithms.eaSimple(toolbox, pop, cxpb=0.5, mutpb=0.2, ngen=40, stats=stats, halloffame=hof)
    logging.info("Best individual is %s, %s", hof[0], hof[0].fitness.values)

    return pop, stats, hof

dtm.start(main)      # Launch the first task
```

As one can see, parallelization requires almost no changes at all (an import, the selection of the distributed map and the starting instruction), even with a non-trivial program. This program can now be run on a multi-cores computer, on a small cluster or on a supercomputer, without any changes, as long as those environments provide a MPI implementation.

---

**Note:** In this specific case, the distributed version would be actually *slower* than the serial one, because of the extreme simplicity of the evaluation function (which takes *less than 0.1 ms* to execute), as the small overhead generated by the serialization, load-balancing, treatment and transfer of the tasks and the results is not balanced by a gain in the evaluation time. In more complex, real-life problems (for instance sorting networks), the benefit of a distributed version is fairly noticeable.

---

CHAPTER

THREE

# API

The API of DEAP is built around a clean and lean core composed of two simple structures: a `creator` and `Toolbox`. The former allows creation of classes via inheritance and composition and the latter contains the tools that are required by the evolutionary algorithm. The core functionalities of DEAP are levered by several peripheral modules. The `tools` module provides a bunch of operator that are commonly used in evolutionary computation such as initialization, mutation, crossover and selection functions. It also contains all sort of tools to gather information about the evolution such as statistics on almost every thing, genealogy of the produced individuals, hall-of-fame of the best individuals seen, and checkpoints allowing to restart an evolution from its last state. The `algorithms` module contains five of the most frequently used algorithms in EC: generational, steady-state, $(\mu\ ,\ \lambda)$, $(\mu + \lambda)$, and CMA-ES. These are readily usable for the most common population layouts. Specific genetic programming tools are provided in the `gp` module. A complete and efficient (and stand alone) parallelization module, base on MPI, providing some very useful parallel method is provided in the `dtm` module. Finally, common benchmark functions are readily implemented in the `benchmarks`.

## 3.1 Core Architecture

The core architecture of DEAP is composed of two simple structures, the `creator` and the `toolbox`. The former provides structuring capabilities, while the latter adds genericity potential to every algorithm. Both structures are described in detail in the following sections.

### 3.1.1 Creator

The `creator` module is the heart and soul of DEAP, it allows to create classes that will fulfill the needs of your evolutionary algorithms. This module follows the meta-factory paradigm by allowing to create new classes via both composition and inheritance. Attributes both datas and functions are added to existing types in order to create new types empowered with user specific evolutionary computation capabilities. In effect, new classes can be built from any imaginable type, from `list` to `set`, `dict`, `PrimitiveTree` and more, providing the possibility to implement genetic algorithms, genetic programming, evolution strategies, particle swarm optimizers, and many more.

`deap.creator.`**`create`**(*name*, *base*[, *attribute*[, ... ]])

Creates a new class named *name* inheriting from *base* in the `creator` module. The new class can have attributes defined by the subsequent keyword arguments passed to the function create. If the argument is a class (without the parenthesis), the __init__ function is called in the initialization of an instance of the new object and the returned instance is added as an attribute of the class' instance. Otherwise, if the argument is not a class, (for example an `int`), it is added as a "static" attribute of the class.

**Parameters**

- **name** – The name of the class to create.

- **base** – A base class from which to inherit.
- **attribute** – One or more attributes to add on instanciation of this class, optional.

The following is used to create a class `Foo` inheriting from the standard `list` and having an attribute `bar` being an empty dictionary and a static attribute `spam` initialized to 1.

```
create("Foo", list, bar=dict, spam=1)
```

This above line is exactly the same as defining in the `creator` module something like the following.

```
class Foo(list):
    spam = 1

    def __init__(self):
        self.bar = dict()
```

The *Creating Types* tutorial gives more examples of the creator usage.

`deap.creator.`**`class_replacers`** **= {<type 'array.array'>: <class 'deap.creator._array'>}**

Some classes in Python's standard library as well as third party library may be in part incompatible with the logic used in DEAP. In order to palliate to this problem, the method `create()` uses the dictionary *class_replacers* to identify if the base type provided is problematic, and if so the new class inherits from the replacement class instead of the original base class.

*class_replacers* keys are classes to be replaced and the values are the replacing classes.

### 3.1.2 Toolbox

The `Toolbox` is a container for the tools that are selected by the user. The toolbox is manually populated with the desired tools that best apply with the chosen representation and algorithm from the user's point of view. This way it is possible to build algorithms that are totally decoupled from the operator set, as one only need to update the toolbox in order to make the algorithm run with a different operator set as the algorithms are built to use aliases instead of direct function names.

**class** `deap.base.`**`Toolbox`**

A toolbox for evolution that contains the evolutionary operators. At first the toolbox contains two simple methods. The first method `clone()` duplicates any element it is passed as argument, this method defaults to the `copy.deepcopy()` function. The second method `map()` applies the function given as first argument to every items of the iterables given as next arguments, this method defaults to the `map()` function. You may populate the toolbox with any other function by using the `register()` method.

Concrete usages of the toolbox are shown for initialization in the *Creating Types* tutorial and for tools container in the *Next Step Toward Evolution* tutorial.

**`register`**(*alias*, *method*[, *argument*[, ...]])

Register a *method* in the toolbox under the name *alias*. You may provide default arguments that will be passed automatically when calling the registered method. Fixed arguments can then be overriden at function call time.

**Parameters**

- **alias** – The name the operator will take in the toolbox. If the alias already exist it will overwrite the the operator already present.
- **method** – The function to which refer the alias.
- **argument** – One or more argument (and keyword argument) to pass automatically to the registered function when called, optional.

The following code block is an example of how the toolbox is used.

```
>>> def func(a, b, c=3):
...     print a, b, c
...
>>> tools = Toolbox()
>>> tools.register("myFunc", func, 2, c=4)
>>> tools.myFunc(3)
2 3 4
```

The registered function will be given the attributes `__name__` set to the alias and `__doc__` set to the original function's documentation. The `__dict__` attribute will also be updated with the original function's instance dictionnary, if any.

**unregister**(*alias*)
Unregister *alias* from the toolbox.

**Parameters** **alias** – The name of the operator to remove from the toolbox.

**decorate**(*alias*, *decorator*[, *decorator*[, ...]])
Decorate *alias* with the specified *decorators*, *alias* has to be a registered function in the current toolbox.

**Parameters**

- **alias** – The name of the operator to decorate.
- **decorator** – One or more function decorator. If multiple decorators are provided they will be applied in order, with the last decorator decorating all the others.

Changed in version 0.8: Decoration is not signature preserving anymore.

### 3.1.3 Fitness

**class** deap.base.**Fitness**([*values*])
The fitness is a measure of quality of a solution. If *values* are provided as a tuple, the fitness is initalized using those values, otherwise it is empty (or invalid).

**Parameters** **values** – The initial values of the fitness as a tuple, optional.

Fitnesses may be compared using the `>`, `<`, `>=`, `<=`, `==`, `!=`. The comparison of those operators is made lexicographically. Maximization and minimization are taken care off by a multiplication between the `weights` and the fitness `values`. The comparison can be made between fitnesses of different size, if the fitnesses are equal until the extra elements, the longer fitness will be superior to the shorter.

Different types of fitnesses are created in the *Creating Types* tutorial.

---

**Note:** When comparing fitness values that are **minimized**, `a > b` will return `True` if *a* is **smaller** than *b*.

---

**valid**
Assess if a fitness is valid or not.

**values**
Fitness values. Use directly `individual.fitness.values = values` in order to set the fitness and `del individual.fitness.values` in order to clear (invalidate) the fitness. The (unweighted) fitness can be directly accessed via `individual.fitness.values`.

**weights = None**
The weights are used in the fitness comparison. They are shared among all fitnesses of the same type. When subclassing `Fitness`, the weights must be defined as a tuple where each element is associated to

an objective. A negative weight element corresponds to the minimization of the associated objective and positive weight to the maximization.

---

**Note:** If weights is not defined during subclassing, the following error will occur at instantiation of a subclass fitness object:

```
TypeError: Can't instantiate abstract <class Fitness[...]> with
abstract attribute weights.
```

---

**wvalues** = ()

Contains the weighted values of the fitness, the multiplication with the weights is made when the values are set via the property `values`. Multiplication is made on setting of the values for efficiency.

Generally it is unnecessary to manipulate wvalues as it is an internal attribute of the fitness used in the comparison operators.

## 3.2 Evolutionary Tools

The `tools` module contains the operators for evolutionary algorithms. They are used to modify, select and move the individuals in their environment. The set of operators it contains are readily usable in the `Toolbox`. In addition to the basic operators this module also contains utility tools to enhance the basic algorithms with `Statistics`, `HallOfFame`, `Checkpoint`, and `History`.

### 3.2.1 Operators

The operator set does the minimum job for transforming or selecting individuals. This means, for example, that providing two individuals to the crossover will transform those individuals in-place. The responsibility of making offspring(s) independent of their parent(s) and invalidating the fitness is left to the user and is generally fulfilled in the algorithms by calling `toolbox.clone()` on an individual to duplicate it and `del` on the `values` attribute of the individual's fitness to invalidate it.

Here is a list of the implemented operators in DEAP,

| Initialization | Crossover | Mutation | Selection | Migration |
|---|---|---|---|---|
| initRepeat() | cxOnePoint() | mutGaussian() | selTournament() | migRing() |
| initIterate() | cxTwoPoints() | mutShuffleIndexes() | selRoulette() | |
| initCycle() | cxUniform() | mutFlipBit() | selNSGA2() | |
| | cxPartialyMatched() | mutPolynomialBounded() | selSPEA2() | |
| | cxUniformPartialyMatched() | mutUniformInt() | selRandom() | |
| | cxOrdered() | mutESLogNormal() | selBest() | |
| | cxBlend() | | selWorst() | |
| | cxESBlend() | | selTournamentDCD() | |
| | cxESTwoPoints() | | selDoubleTournament() | |
| | cxSimulatedBinary() | | | |
| | cxSimulatedBinaryBounded() | | | |
| | cxMessyOnePoint() | | | |

and genetic programming specific operators.

| Initialization | Crossover | Mutation | Bloat control |
|---|---|---|---|
| genFull() | cxOnePoint() | mutUniform() | staticDepthLimit() |
| genGrow() | cxOnePointLeafBiased() | mutNodeReplacement() | staticSizeLimit() |
| genRamped() | | mutEphemeral() | selDoubleTournament() |
| | | mutInsert() | |

### Initialization

deap.tools.**initRepeat**(*container*, *func*, *n*)

Call the function *container* with a generator function corresponding to the calling *n* times the function *func*.

**Parameters**

- **container** – The type to put in the data from func.
- **func** – The function that will be called n times to fill the container.
- **n** – The number of times to repeat func.

**Returns** An instance of the container filled with data from func.

This helper function can can be used in conjunction with a Toolbox to register a generator of filled containers, as individuals or population.

```
>>> initRepeat(list, random.random, 2)
...
[0.4761..., 0.6302...]
```

See the *List of Floats* and *Population* tutorials for more examples.

deap.tools.**initIterate**(*container*, *generator*)

Call the function *container* with an iterable as its only argument. The iterable must be returned by the method or the object *generator*.

**Parameters**

- **container** – The type to put in the data from func.
- **generator** – A function returning an iterable (list, tuple, ...), the content of this iterable will fill the container.

**Returns** An instance of the container filled with data from the generator.

This helper function can can be used in conjunction with a Toolbox to register a generator of filled containers, as individuals or population.

```
>>> from random import sample
>>> from functools import partial
>>> gen_idx = partial(sample, range(10), 10)
>>> initIterate(list, gen_idx)
[4, 5, 3, 6, 0, 9, 2, 7, 1, 8]
```

See the *Permutation* and *Arithmetic Expression* tutorials for more examples.

deap.tools.**initCycle**(*container*, *seq_func*, *n=1*)

Call the function *container* with a generator function corresponding to the calling *n* times the functions present in *seq_func*.

**Parameters**

- **container** – The type to put in the data from func.
- **seq_func** – A list of function objects to be called in order to fill the container.

- **n** – Number of times to iterate through the list of functions.

**Returns** An instance of the container filled with data from the returned by the functions.

This helper function can can be used in conjunction with a Toolbox to register a generator of filled containers, as individuals or population.

```
>>> func_seq = [lambda:1 , lambda:'a', lambda:3]
>>> initCycle(list, func_seq, n=2)
[1, 'a', 3, 1, 'a', 3]
```

See the *A Funky One* tutorial for an example.

deap.gp.**genFull**(*pset*, *min_*, *max_*, *type_=None*)

Generate an expression where each leaf has a the same depth between *min* and *max*.

**Parameters**

- **pset** – A primitive set from wich to select primitives of the trees.
- **min** – Minimum height of the produced trees.
- **max** – Maximum Height of the produced trees.
- **type** – The type that should return the tree when called, when `None` (default) no return type is enforced.

**Returns** A full tree with all leaves at the same depth.

deap.gp.**genGrow**(*pset*, *min_*, *max_*, *type_=None*)

Generate an expression where each leaf might have a different depth between *min* and *max*.

**Parameters**

- **pset** – A primitive set from wich to select primitives of the trees.
- **min** – Minimum height of the produced trees.
- **max** – Maximum Height of the produced trees.
- **type** – The type that should return the tree when called, when `None` (default) no return type is enforced.

**Returns** A grown tree with leaves at possibly different depths.

deap.gp.**genRamped**(*pset*, *min_*, *max_*, *type_=None*)

Generate an expression with a PrimitiveSet *pset*. Half the time, the expression is generated with `genGrow()`, the other half, the expression is generated with `genFull()`.

**Parameters**

- **pset** – A primitive set from wich to select primitives of the trees.
- **min** – Minimum height of the produced trees.
- **max** – Maximum Height of the produced trees.
- **type** – The type that should return the tree when called, when `None` (default) no return type is enforced.

**Returns** Either, a full or a grown tree.

### Crossover

deap.tools.**cxOnePoint**(*ind1*, *ind2*)
Execute a one point crossover on the input individuals. The two individuals are modified in place. The resulting individuals will respectively have the length of the other.

**Parameters**

- **ind1** – The first individual participating in the crossover.
- **ind2** – The second individual participating in the crossover.

**Returns** A tuple of two individuals.

This function use the `randint()` function from the python base `random` module.

deap.tools.**cxTwoPoints**(*ind1*, *ind2*)
Execute a two points crossover on the input individuals. The two individuals are modified in place and both keep their original length.

**Parameters**

- **ind1** – The first individual participating in the crossover.
- **ind2** – The second individual participating in the crossover.

**Returns** A tuple of two individuals.

This function use the `randint()` function from the python base `random` module.

deap.tools.**cxUniform**(*ind1*, *ind2*, *indpb*)
Execute a uniform crossover that modify in place the two individuals. The attributes are swapped according to the *indpb* probability.

**Parameters**

- **ind1** – The first individual participating in the crossover.
- **ind2** – The second individual participating in the crossover.
- **indpb** – Independent probabily for each attribute to be exchanged.

**Returns** A tuple of two individuals.

This function use the `random()` function from the python base `random` module.

deap.tools.**cxPartialyMatched**(*ind1*, *ind2*)
Execute a partially matched crossover (PMX) on the input individuals. The two individuals are modified in place. This crossover expect iterable individuals of indices, the result for any other type of individuals is unpredictable.

**Parameters**

- **ind1** – The first individual participating in the crossover.
- **ind2** – The second individual participating in the crossover.

**Returns** A tuple of two individuals.

Moreover, this crossover consists of generating two children by matching pairs of values in a certain range of the two parents and swapping the values of those indexes. For more details see [Goldberg1985].

This function use the `randint()` function from the python base `random` module.

deap.tools.**cxUniformPartialyMatched**(*ind1*, *ind2*, *indpb*)
Execute a uniform partially matched crossover (UPMX) on the input individuals. The two individuals are modified in place. This crossover expect iterable individuals of indices, the result for any other type of individuals is unpredictable.

**Parameters**

- **ind1** – The first individual participating in the crossover.
- **ind2** – The second individual participating in the crossover.

**Returns** A tuple of two individuals.

Moreover, this crossover consists of generating two children by matching pairs of values chosen at random with a probability of *indpb* in the two parents and swapping the values of those indexes. For more details see [Cicirello2000].

This function use the `random()` and `randint()` functions from the python base `random` module.

`deap.tools.`**`cxOrdered`**(*ind1*, *ind2*)

Execute an ordered crossover (OX) on the input individuals. The two individuals are modified in place. This crossover expect iterable individuals of indices, the result for any other type of individuals is unpredictable.

**Parameters**

- **ind1** – The first individual participating in the crossover.
- **ind2** – The second individual participating in the crossover.

**Returns** A tuple of two individuals.

Moreover, this crossover consists of generating holes in the input individuals. A hole is created when an attribute of an individual is between the two crossover points of the other individual. Then it rotates the element so that all holes are between the crossover points and fills them with the removed elements in order. For more details see [Goldberg1989].

This function use the `sample()` function from the python base `random` module.

`deap.tools.`**`cxBlend`**(*ind1*, *ind2*, *alpha*)

Executes a blend crossover that modify in-place the input individuals. The blend crossover expect individuals formed of a list of floating point numbers.

**Parameters**

- **ind1** – The first individual participating in the crossover.
- **ind2** – The second individual participating in the crossover.
- **alpha** – Extent of the interval in which the new values can be drawn for each attribute on both side of the parents' attributes.

**Returns** A tuple of two individuals.

This function use the `random()` function from the python base `random` module.

`deap.tools.`**`cxESBlend`**(*ind1*, *ind2*, *alpha*)

Execute a blend crossover on both, the individual and the strategy. The individuals must have a `strategy` attribute. Adjustement of the minimal strategy shall be done after the call to this function, consider using a decorator.

**Parameters**

- **ind1** – The first evolution strategy participating in the crossover.
- **ind2** – The second evolution strategy participating in the crossover.
- **alpha** – Extent of the interval in which the new values can be drawn for each attribute on both side of the parents' attributes.

**Returns** A tuple of two evolution strategies.

deap.tools.**cxESTwoPoints**(*ind1*, *ind2*)
Execute a classical two points crossover on both the individual and its strategy. The individuals must have a `strategy` attribute.The crossover points for the individual and the strategy are the same.

**Parameters**

- **ind1** – The first evolution strategy participating in the crossover.
- **ind2** – The second evolution strategy participating in the crossover.

**Returns** A tuple of two evolution strategies.

deap.tools.**cxSimulatedBinary**(*ind1*, *ind2*, *eta*)
Executes a simulated binary crossover that modify in-place the input individuals. The simulated binary crossover expect individuals formed of a list of floating point numbers.

**Parameters**

- **ind1** – The first individual participating in the crossover.
- **ind2** – The second individual participating in the crossover.
- **eta** – Crowding degree of the crossover. A high eta will produce children resembling to their parents, while a small eta will produce solutions much more different.

**Returns** A tuple of two individuals.

This function use the `random()` function from the python base `random` module.

deap.tools.**cxSimulatedBinaryBounded**(*ind1*, *ind2*, *eta*, *low*, *up*)
Executes a simulated binary crossover that modify in-place the input individuals. The simulated binary crossover expect individuals formed of a list of floating point numbers.

**Parameters**

- **ind1** – The first individual participating in the crossover.
- **ind2** – The second individual participating in the crossover.
- **eta** – Crowding degree of the crossover. A high eta will produce children resembling to their parents, while a small eta will produce solutions much more different.
- **low** – A value or a sequence of values that is the lower bound of the search space.
- **up** – A value or a sequence of values that is the upper bound of the search space.

**Returns** A tuple of two individuals.

This function use the `random()` function from the python base `random` module.

---

**Note:** This implementation is similar to the one implemented in the original NSGA-II C code presented by Deb.

---

deap.tools.**cxMessyOnePoint**(*ind1*, *ind2*)
Execute a one point crossover that will in most cases change the individuals size. The two individuals are modified in place.

**Parameters**

- **ind1** – The first individual participating in the crossover.
- **ind2** – The second individual participating in the crossover.

**Returns** A tuple of two individuals.

This function use the `randint()` function from the python base `random` module.

deap.gp.**cxOnePoint**(*ind1*, *ind2*)
Randomly select in each individual and exchange each subtree with the point as root between each individual.

**Parameters**

- **ind1** – First tree participating in the crossover.
- **ind2** – Second tree participating in the crossover.

**Returns** A tuple of two trees.

deap.gp.**cxOnePointLeafBiased**(*ind1*, *ind2*, *termpb*)
Randomly select crossover point in each individual and exchange each subtree with the point as root between each individual.

**Parameters**

- **ind1** – First typed tree participating in the crossover.
- **ind2** – Second typed tree participating in the crossover.
- **termpb** – The probability of chosing a terminal node (leaf).

**Returns** A tuple of two typed trees.

When the nodes are strongly typed, the operator makes sure the second node type corresponds to the first node type.

The parameter *termpb* sets the probability to choose between a terminal or non-terminal crossover point. For instance, as defined by Koza, non- terminal primitives are selected for 90% of the crossover points, and terminals for 10%, so *termpb* should be set to 0.1.

### Mutation

deap.tools.**mutGaussian**(*individual*, *mu*, *sigma*, *indpb*)
This function applies a gaussian mutation of mean *mu* and standard deviation *sigma* on the input individual. This mutation expects an iterable individual composed of real valued attributes. The *indpb* argument is the probability of each attribute to be mutated.

**Parameters**

- **individual** – Individual to be mutated.
- **mu** – Mean around the individual of the mutation.
- **sigma** – Standard deviation of the mutation.
- **indpb** – Probability for each attribute to be mutated.

**Returns** A tuple of one individual.

This function uses the `random()` and `gauss()` functions from the python base `random` module.

deap.tools.**mutShuffleIndexes**(*individual*, *indpb*)
Shuffle the attributes of the input individual and return the mutant. The *individual* is expected to be iterable. The *indpb* argument is the probability of each attribute to be moved. Usually this mutation is applied on vector of indices.

**Parameters**

- **individual** – Individual to be mutated.
- **indpb** – Probability for each attribute to be exchanged to another position.

**Returns** A tuple of one individual.

This function uses the `random()` and `randint()` functions from the python base `random` module.

`deap.tools.`**`mutFlipBit`**(*individual*, *indpb*)

Flip the value of the attributes of the input individual and return the mutant. The *individual* is expected to be iterable and the values of the attributes shall stay valid after the `not` operator is called on them. The *indpb* argument is the probability of each attribute to be flipped. This mutation is usually applied on boolean individuals.

**Parameters**

- **individual** – Individual to be mutated.
- **indpb** – Probability for each attribute to be flipped.

**Returns** A tuple of one individual.

This function uses the `random()` function from the python base `random` module.

`deap.tools.`**`mutUniformInt`**(*individual*, *low*, *up*, *indpb*)

Mutate an individual by replacing attributes, with probability *indpb*, by a integer uniformly drawn between *low* and *up* inclusively.

**Parameters**

- **low** – The lower bound of the range from wich to draw the new integer.
- **up** – The upper bound of the range from wich to draw the new integer.
- **indpb** – Probability for each attribute to be mutated.

**Returns** A tuple of one individual.

`deap.tools.`**`mutPolynomialBounded`**(*individual*, *eta*, *low*, *up*, *indpb*)

Polynomial mutation as implemented in original NSGA-II algorithm in C by Deb.

**Parameters**

- **individual** – Individual to be mutated.
- **eta** – Crowding degree of the mutation. A high eta will produce a mutant resembling its parent, while a small eta will produce a solution much more different.
- **low** – A value or a sequence of values that is the lower bound of the search space.
- **up** – A value or a sequence of values that is the upper bound of the search space.

**Returns** A tuple of one individual.

`deap.tools.`**`mutESLogNormal`**(*individual*, *c*, *indpb*)

Mutate an evolution strategy according to its `strategy` attribute as described in [Beyer2002]. First the strategy is mutated according to an extended log normal rule, $\boldsymbol{\sigma}_t = \exp(\tau_0 \mathcal{N}_0(0,1))\,[\sigma_{t-1,1}\exp(\tau\mathcal{N}_1(0,1)), \ldots, \sigma_{t-1,n}\exp(\tau\mathcal{N}_n(0,1))]$, with $\tau_0 = \frac{c}{\sqrt{2n}}$ and $\tau = \frac{c}{\sqrt{2\sqrt{n}}}$, the the individual is mutated by a normal distribution of mean 0 and standard deviation of $\boldsymbol{\sigma}_t$ (its current strategy) then . A recommended choice is `c=1` when using a $(10, 100)$ evolution strategy [Beyer2002] [Schwefel1995].

**Parameters**

- **individual** – Individual to be mutated.
- **c** – The learning parameter.
- **indpb** – Probability for each attribute to be flipped.

**Returns** A tuple of one individual.

deap.gp.**mutUniform**(*individual*, *expr*)
Randomly select a point in the tree *individual*, then replace the subtree at that point as a root by the expression generated using method `expr()`.

**Parameters**

- **individual** – The tree to be mutated.
- **expr** – A function object that can generate an expression when called.

**Returns** A tuple of one tree.

deap.gp.**mutNodeReplacement**(*individual*)
Replaces a randomly chosen primitive from *individual* by a randomly chosen primitive with the same number of arguments from the `pset` attribute of the individual.

**Parameters** **individual** – The normal or typed tree to be mutated.

**Returns** A tuple of one tree.

deap.gp.**mutEphemeral**(*individual*, *mode*)
This operator works on the constants of the tree *individual*. In *mode* `"one"`, it will change the value of one of the individual ephemeral constants by calling its generator function. In *mode* `"all"`, it will change the value of **all** the ephemeral constants.

**Parameters**

- **individual** – The normal or typed tree to be mutated.
- **mode** – A string to indicate to change `"one"` or `"all"` ephemeral constants.

**Returns** A tuple of one tree.

deap.gp.**mutInsert**(*individual*)
Inserts a new branch at a random position in *individual*. The subtree at the chosen position is used as child node of the created subtree, in that way, it is really an insertion rather than a replacement. Note that the original subtree will become one of the children of the new primitive inserted, but not perforce the first (its position is randomly selected if the new primitive has more than one child).

**Parameters** **individual** – The normal or typed tree to be mutated.

**Returns** A tuple of one tree.

### Selection

deap.tools.**selTournament**(*individuals*, *k*, *tournsize*)
Select *k* individuals from the input *individuals* using *k* tournaments of *tournsize* individuals. The list returned contains references to the input *individuals*.

**Parameters**

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.
- **tournsize** – The number of individuals participating in each tournament.

**Returns** A list of selected individuals.

This function uses the `choice()` function from the python base `random` module.

deap.tools.**selRoulette**(*individuals*, *k*)
Select *k* individuals from the input *individuals* using *k* spins of a roulette. The selection is made by looking only at the first objective of each individual. The list returned contains references to the input *individuals*.

**Parameters**

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.

**Returns** A list of selected individuals.

This function uses the `random()` function from the python base `random` module.

**Warning:** The roulette selection by definition cannot be used for minimization or when the fitness can be smaller or equal to 0.

`deap.tools.`**`selNSGA2`**(*individuals*, *k*)

Apply NSGA-II selection operator on the *individuals*. Usually, the size of *individuals* will be larger than *k* because any individual present in *individuals* will appear in the returned list at most once. Having the size of *individuals* equals to *k* will have no effect other than sorting the population according to a non-domination scheme. The list returned contains references to the input *individuals*. For more details on the NSGA-II operator see [Deb2002].

**Parameters**

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.

**Returns** A list of selected individuals.

`deap.tools.`**`selSPEA2`**(*individuals*, *k*)

Apply SPEA-II selection operator on the *individuals*. Usually, the size of *individuals* will be larger than *n* because any individual present in *individuals* will appear in the returned list at most once. Having the size of *individuals* equals to *n* will have no effect other than sorting the population according to a strength Pareto scheme. The list returned contains references to the input *individuals*. For more details on the SPEA-II operator see [Zitzler2001].

**Parameters**

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.

**Returns** A list of selected individuals.

`deap.tools.`**`selRandom`**(*individuals*, *k*)

Select *k* individuals at random from the input *individuals* with replacement. The list returned contains references to the input *individuals*.

**Parameters**

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.

**Returns** A list of selected individuals.

This function uses the `choice()` function from the python base `random` module.

`deap.tools.`**`selBest`**(*individuals*, *k*)

Select the *k* best individuals among the input *individuals*. The list returned contains references to the input *individuals*.

**Parameters**

- **individuals** – A list of individuals to select from.

- **k** – The number of individuals to select.

**Returns** A list containing the k best individuals.

deap.tools.**selWorst**(*individuals*, *k*)

Select the *k* worst individuals among the input *individuals*. The list returned contains references to the input *individuals*.

**Parameters**

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.

**Returns** A list containing the k worst individuals.

deap.tools.**selDoubleTournament**(*individuals*, *k*, *fitness_size*, *parsimony_size*, *fitness_first*)

Tournament selection which use the size of the individuals in order to discriminate good solutions. This kind of tournament is obviously useless with fixed-length representation, but has been shown to significantly reduce excessive growth of individuals, especially in GP, where it can be used as a bloat control technique (see [Luke2002fighting]). This selection operator implements the double tournament technique presented in this paper.

The core principle is to use a normal tournament selection, but using a special sample function to select aspirants, which is another tournament based on the size of the individuals. To ensure that the selection pressure is not too high, the size of the size tournament (the number of candidates evaluated) can be a real number between 1 and 2. In this case, the smaller individual among two will be selected with a probability *size_tourn_size*/2. For instance, if *size_tourn_size* is set to 1.4, then the smaller individual will have a 0.7 probability to be selected.

---

**Note:** In GP, it has been shown that this operator produces better results when it is combined with some kind of a depth limit.

---

**Parameters**

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.
- **fitness_size** – The number of individuals participating in each fitness tournament
- **parsimony_size** – The number of individuals participating in each size tournament. This value has to be a real number in the range [1,2], see above for details.
- **fitness_first** – Set this to True if the first tournament done should be the fitness one (i.e. the fitness tournament producing aspirants for the size tournament). Setting it to False will behaves as the opposite (size tournament feeding fitness tournaments with candidates). It has been shown that this parameter does not have a significant effect in most cases (see [Luke2002fighting]).

**Returns** A list of selected individuals.

deap.tools.**selTournamentDCD**(*individuals*, *k*)

Tournament selection based on dominance (D) between two individuals, if the two individuals do not interdominate the selection is made based on crowding distance (CD). The *individuals* sequence length has to be a multiple of 4. Starting from the beginning of the selected individuals, two consecutive individuals will be different (assuming all individuals in the input list are unique). Each individual from the input list won't be selected more than twice.

This selection requires the individuals to have a `crowding_dist` attribute, which can be set by the `assignCrowdingDist()` function.

**Parameters**

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.

**Returns** A list of selected individuals.

deap.tools.**sortFastND**(*individuals*, *k*, *first_front_only=False*)
Sort the first *k individuals* according the the fast non-dominated sorting algorithm.

**Parameters**

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.
- **first_front_only** – If `True` sort only the first front and exit.

**Returns** A list of Pareto fronts (lists), with the first list being the true Pareto front.

deap.tools.**assignCrowdingDist**(*individuals*)
Assign a crowding distance to each individual of the list. The crowding distance is set to the `crowding_dist` attribute of each individual.

### Bloat control

deap.gp.**staticDepthLimit**(*max_depth*)
Implement a static limit on the depth of a GP tree, as defined by Koza in [Koza1989]. It may be used to decorate both crossover and mutation operators. When an invalid (too high) child is generated, it is simply replaced by one of its parents.

This operator can be used to avoid memory errors occuring when the tree gets higher than 90-95 levels (as Python puts a limit on the call stack depth), because it ensures that no tree higher than *max_depth* will ever be accepted in the population (except if it was generated at initialization time).

**Parameters** **max_depth** – The maximum depth allowed for an individual.

**Returns** A decorator that can be applied to a GP operator using `decorate()`

---

**Note:** If you want to reproduce the exact behavior intended by Koza, set the *max_depth* param to 17.

---

deap.gp.**staticSizeLimit**(*max_size*)
Implement a static limit on the size of a GP tree. It may be used to decorate both crossover and mutation operators. When an invalid (too big) child is generated, it is simply replaced by one of its parents.

**Parameters** **max_size** – The maximum size (number of nodes) allowed for an individual

**Returns** A decorator that can be applied to a GP operator using `decorate()`

### Migration

deap.tools.**migRing**(*populations*, *k*, *selection*[, *replacement*, *migarray*])
Perform a ring migration between the *populations*. The migration first select *k* emigrants from each population using the specified *selection* operator and then replace *k* individuals from the associated population in the *migarray* by the emigrants. If no *replacement* operator is specified, the immigrants will replace the emigrants of the

population, otherwise, the immigrants will replace the individuals selected by the *replacement* operator. The migration array, if provided, shall contain each population's index once and only once. If no migration array is provided, it defaults to a serial ring migration (1 – 2 – ... – n – 1). Selection and replacement function are called using the signature `selection(populations[i], k)` and `replacement(populations[i], k)`. It is important to note that the replacement strategy must select *k* **different** individuals. For example, using a traditional tournament for replacement strategy will thus give undesirable effects, two individuals will most likely try to enter the same slot.

**Parameters**

- **populations** – A list of (sub-)populations on which to operate migration.
- **k** – The number of individuals to migrate.
- **selection** – The function to use for selection.
- **replacement** – The function to use to select which individuals will be replaced. If `None` (default) the individuals that leave the population are directly replaced.
- **migarray** – A list of indices indicating where the individuals from a particular position in the list goes. This defaults to a ring migration.

### 3.2.2 Statistics

**class** `deap.tools.`**`Statistics`**(*[key][, n]*)

A statistics object that holds the required data for as long as it exists. When created the statistics object receives a *key* argument that is used to get the required data, if not provided the *key* argument defaults to the identity function. A statistics object can be represented as multiple 3 dimensional matrix, for each registered function there is a matrix.

**Parameters**

- **key** – A function to access the data on which to compute the statistics, optional.
- **n** – The number of independent statistics to maintain in this statistics object.

Along the first axis (wich length is given by the *n* argument) are independent statistics on different collections given this index in the `update()` method. The second is the accumulator of statistics, each time the update function is called the new statistics are added using the registered functions at the end of this axis. The third axis is used when the entered data is an iterable (for example a multiobjective fitness).

Data can be retrieved by accessing the statistic function name followed by the indicices of the element we wish to access.

```
>>> s = Statistics(n=2)
>>> s.register("Mean", mean)
>>> s.update([1, 2, 3, 4], index=0)
>>> s.update([5, 6, 7, 8], index=1)
>>> s.Mean[0][-1]
[2.5]
>>> s.Mean[1][-1]
[6.5]
>>> s.update([10, 20, 30, 40], index=0)
>>> s.update([50, 60, 70, 80], index=1)
>>> s.Mean[0][1]
[25.0]
>>> s.Mean[1][1]
[65.0]
```

**register**(*name*, *function*)
Register a function *function* that will be apply on the sequence each time `update()` is called.

**Parameters**

- **name** – The name of the statistics function as it would appear in the dictionnary of the statistics object.
- **function** – A function that will compute the desired statistics on the data as preprocessed by the key.

The function result will be accessible by using the string given by the argument *name* as a function of the statistics object.

```
>>> s = Statistics()
>>> s.register("Mean", mean)
>>> s.update([1,2,3,4,5,6,7])
>>> s.Mean
[[[4.0]]]
```

**update**(*seq*, *index=0*, *add=False*)
Apply to the input sequence *seq* each registered function and store each result in a list specific to the function and the data index *index*.

**Parameters**

- **seq** – A sequence on which the key will be applied to get the data.
- **index** – The index of the independent statistics object in which to add the computed statistics, optional (defaults to 0).
- **add** – A boolean to force adding depth to the statistics object when the index is out of range. If the given index is not yet registered, it adds it to the statistics only if it is one greater than the larger index, optional (defaults to False).

```
>>> s = Statistics()
>>> s.register("Mean", mean)
>>> s.register("Max", max)
>>> s.update([4,5,6,7,8])
>>> s.Max
[[[8]]]
>>> s.Mean
[[[6.0]]]
>>> s.update([1,2,3])
>>> s.Max
[[[8], [3]]]
```

deap.tools.**mean**(*seq*)
Returns the arithmetic mean of the sequence *seq* = $\{x_1, \ldots, x_n\}$ as $A = \frac{1}{n}\sum_{i=1}^{n} x_i$.

deap.tools.**median**(*seq*)
Returns the median of *seq* - the numeric value separating the higher half of a sample from the lower half. If there is an even number of elements in *seq*, it returns the mean of the two middle values.

deap.tools.**var**(*seq*)
Returns the variance $\sigma^2$ of *seq* = $\{x_1, \ldots, x_n\}$ as $\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$, where $\mu$ is the arithmetic mean of *seq*.

deap.tools.**std**(*seq*)
Returns the square root of the variance $\sigma^2$ of *seq*.

### 3.2.3 Hall-Of-Fame

**class** deap.tools.**HallOfFame**(*maxsize*)

The hall of fame contains the best individual that ever lived in the population during the evolution. It is sorted at all time so that the first element of the hall of fame is the individual that has the best first fitness value ever seen, according to the weights provided to the fitness at creation time.

**Parameters** **maxsize** – The maximum number of individual to keep in the hall of fame.

The class HallOfFame provides an interface similar to a list (without being one completely). It is possible to retrieve its length, to iterate on it forward and backward and to get an item or a slice from it.

**update**(*population*)

Update the hall of fame with the *population* by replacing the worst individuals in it by the best individuals present in *population* (if they are better). The size of the hall of fame is kept constant.

**Parameters** **population** – A list of individual with a fitness attribute to update the hall of fame with.

**insert**(*item*)

Insert a new individual in the hall of fame using the bisect_right() function. The inserted individual is inserted on the right side of an equal individual. Inserting a new individual in the hall of fame also preserve the hall of fame's order. This method **does not** check for the size of the hall of fame, in a way that inserting a new individual in a full hall of fame will not remove the worst individual to maintain a constant size.

**Parameters** **item** – The individual with a fitness attribute to insert in the hall of fame.

**remove**(*index*)

Remove the specified *index* from the hall of fame.

**Parameters** **index** – An integer giving which item to remove.

**clear**()

Clear the hall of fame.

**class** deap.tools.**ParetoFront**([*similar*])

The Pareto front hall of fame contains all the non-dominated individuals that ever lived in the population. That means that the Pareto front hall of fame can contain an infinity of different individuals.

**Parameters** **similar** – A function that tels the Pareto front whether or not two individuals are similar, optional.

The size of the front may become very large if it is used for example on a continuous function with a continuous domain. In order to limit the number of individuals, it is possible to specify a similarity function that will return True if the genotype of two individuals are similar. In that case only one of the two individuals will be added to the hall of fame. By default the similarity function is operator.__eq__().

Since, the Pareto front hall of fame inherits from the HallOfFame, it is sorted lexicographically at every moment.

**update**(*population*)

Update the Pareto front hall of fame with the *population* by adding the individuals from the population that are not dominated by the hall of fame. If any individual in the hall of fame is dominated it is removed.

**Parameters** **population** – A list of individual with a fitness attribute to update the hall of fame with.

### 3.2.4 Evolution Logger

**class** deap.tools.**EvolutionLogger**(*[col_names][, precision]*)

The evolution logger logs data about the evolution to either the stdout or a file. To change the destination of the logger simply change its attribute `output` to an filestream. The *columns* argument provides the data column to log when using statistics, the names should be identical to what is registered in the statistics object (it default to *["gen", "evals"]* wich will log the generation number and the number of evaluated individuals). When logging with function `logGeneration()`, the provided columns must be given either as arguments or in the statistics object. The *precision* indicates the precision to use when logging statistics.

**Parameters**

- **columns** – A list of strings of the name of the data as it will appear in the statistics object, optional.
- **precision** – Number of decimal digits to log, optional.

**logHeader**()

Logs the column titles specified during initialization.

**logGeneration**(*[stats[, index]][, data[, ...]]*)

Logs the registered generation identifiers given a columns on initialization. Each element of the columns must be provided either in the *stats* object or as keyword argument. When loggin through the stats object, the last entry of the data under the column names at *index* is logged (*index* defaults to 0).

**Parameters**

- **stats** – A statistics object containing the data to log, optional.
- **index** – The index in the statistics, optional.
- **data** – Kerword arguments of the data that is not in the statistics object, optional.

Here is an example on how to use the logger with a statistics object

```
>>> s = Statistics(n=2)
>>> s.register("mean", mean)
>>> s.register("max", max)
>>> s.update([1, 2, 3, 4], index=0)
>>> s.update([5, 6, 7, 8], index=1)
>>> l = EvolutionLogger(columns=["gen", "evals", "mean", "max"])
>>> l.logHeader()
gen      evals    mean     max
>>> l.logGeneration(gen="0_1", evals=4, stats=s, index=0)
0_1      4        [2.5000] [4.0000]
>>> l.logGeneration(gen="0_2", evals=4, stats=s, index=1)
0_2      4        [6.5000] [8.0000]
```

### 3.2.5 Checkpoint

**class** deap.tools.**Checkpoint**

A checkpoint is a file containing the state of any object that has been hooked. While initializing a checkpoint, add the objects that you want to be dumped by appending keyword arguments to the initializer or using the `add()`.

In order to use efficiently this module, you must understand properly the assignment principles in Python. This module uses the *pointers* you passed to dump the object, for example the following won't work as desired

```
>>> my_object = [1, 2, 3]
>>> cp = Checkpoint()
>>> cp.add("my_object", my_object)
>>> my_object = [3, 5, 6]
>>> cp.dump(open("example.ecp", "w"))
>>> cp.load(open("example.ecp", "r"))
>>> cp["my_object"]
[1, 2, 3]
```

In order to dump the new value of `my_object` it is needed to change its internal values directly and not touch the *label*, as in the following

```
>>> my_object = [1, 2, 3]
>>> cp = Checkpoint()
>>> cp.add("my_object", my_object)
>>> my_object[:] = [3, 5, 6]
>>> cp.dump(open("example.ecp", "w"))
>>> cp.load(open("example.ecp", "r"))
>>> cp["my_object"]
[3, 5, 6]
```

**dump** (*filestream*)
Dump the current registered object values in the provided *filestream*.

**Parameters** **filestream** – A stream in which write the data.

**load** (*filestream*)
Load a checkpoint from the provided *filestream* retrieving the dumped object values, it is not safe to load a checkpoint file in a checkpoint object that contains references as all conflicting names will be updated with the new values.

**Parameters** **filestream** – A stream from which to read a checkpoint.

**add** (*name*, *object*[, *key*])
Add an object to the list of objects to be dumped. The object is added under the name specified by the argument *name*, the object added is *object*, and the *key* argument allow to specify a subpart of the object that should be dumped (*key* defaults to an identity key that dumps the entire object).

**Parameters**

- **name** – The name under which the object will be dumped.
- **object** – The object to register for dumping.
- **key** – A function access the subcomponent of the object to dump, optional.

The following illustrates how to use the key.

```
>>> from operator import itemgetter
>>> my_object = [1, 2, 3]
>>> cp = Checkpoint()
>>> cp.add("item0", my_object, key=itemgetter(0))
>>> cp.dump(open("example.ecp", "w"))
>>> cp.load(open("example.ecp", "r"))
>>> cp["item0"]
1
```

**remove** (*name*[, ...])
Remove objects with the specified name from the list of objects to be dumped.

**Parameters** **name** – The name of one or more object to remove from dumping.

### 3.2.6 History

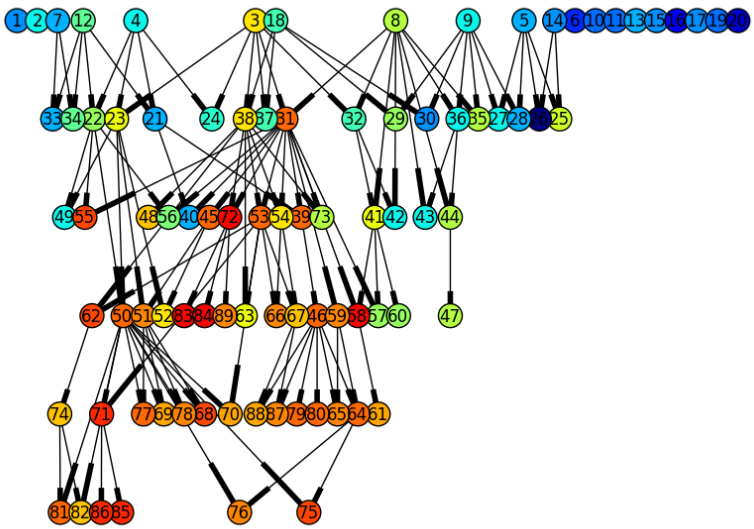
**class** deap.tools.**History**

The History class helps to build a genealogy of all the individuals produced in the evolution. It contains two attributes, the genealogy_tree that is a dictionary of lists indexed by individual, the list contain the indices of the parents. The second attribute genealogy_history contains every individual indexed by their individual number as in the genealogy tree.

The produced genealogy tree is compatible with NetworkX, here is how to plot the genealogy tree

```
history = History()

# Decorate the variation operators
toolbox.decorate("mate", history.decorator)
toolbox.decorate("mutate", history.decorator)

# Create the population and populate the history
population = toolbox.population(n=POPSIZE)
history.update(population)

# Do the evolution, the decorators will take care of updating the
# history
# [...]

import matplotlib.pyplot as plt
import networkx

graph = networkx.DiGraph(history.genealogy_tree)
graph = graph.reverse()     # Make the grah top-down
colors = [toolbox.evaluate(history.genealogy_history[i])[0] for i in graph]
networkx.draw(graph, node_color=colors)
plt.show()
```

Using NetworkX in combination with pygraphviz (dot layout) this amazing genealogy tree can be obtained from the OneMax example with a population size of 20 and 5 generations, where the color of the nodes indicate there fitness, blue is low and red is high.



**Note:** The genealogy tree might get very big if your population and/or the number of generation is large.

**update**(*individuals*)

Update the history with the new *individuals*. The index present in their `history_index` attribute will be used to locate their parents, it is then modified to a unique one to keep track of those new individuals. This method should be called on the individuals after each variation.

**Parameters** **individuals** – The list of modified individuals that shall be inserted in the history.

If the *individuals* do not have a `history_index` attribute, the attribute is added and this individual is considered as having no parent. This method should be called with the initial population to initialize the history.

Modifying the internal `genealogy_index` of the history or the `history_index` of an individual may lead to unpredictable results and corruption of the history.

**decorator**

Property that returns an appropriate decorator to enhance the operators of the toolbox. The returned decorator assumes that the individuals are returned by the operator. First the decorator calls the underlying operation and then calls the `update()` function with what has been returned by the operator. Finally, it returns the individuals with their history parameters modified according to the update function.

**getGenealogy**(*individual*[, *max_depth*])

Provide the genealogy tree of an *individual*. The individual must have an attribute `history_index` as defined by `update()` in order to retrieve its associated genealogy tree. The returned graph contains the parents up to *max_depth* variations before this individual. If not provided the maximum depth is up to the begining of the evolution.

**Parameters**

- **individual** – The individual at the root of the genealogy tree.
- **max_depth** – The approximate maximum distance between the root (individual) and the leaves (parents), optional.

**Returns** A dictionary where each key is an individual index and the values are a tuple corresponding to the index of the parents.

## 3.3 Algorithms

The `algorithms` module is intended to contain some specific algorithms in order to execute very common evolutionary algorithms. The method used here are more for convenience than reference as the implementation of every evolutionary algorithm may vary infinitely. Most of the algorithms in this module use operators registered in the toolbox. Generaly, the keyword used are `mate()` for crossover, `mutate()` for mutation, `select()` for selection and `evaluate()` for evaluation.

You are encouraged to write your own algorithms in order to make them do what you really want them to do.

### 3.3.1 Complete Algorithms

These are complete boxed algorithms that are somewhat limited to the very basic evolutionary computation concepts. All algorithms accept, in addition to their arguments, an initialized `Statistics` object to maintain stats of the evolution, an initialized `HallOfFame` to hold the best individual(s) to appear in the population, and a boolean *verbose* to specify wether to log what is happening during the evolution or not.

`deap.algorithms.`**eaSimple**(*population*, *toolbox*, *cxpb*, *mutpb*, *ngen*[, *stats*, *halloffame*, *verbose*])

This algorithm reproduce the simplest evolutionary algorithm as presented in chapter 7 of [Back2000].

**Parameters**

- **population** – A list of individuals.
- **toolbox** – A `Toolbox` that contains the evolution operators.
- **cxpb** – The probability of mating two individuals.
- **mutpb** – The probability of mutating an individual.
- **ngen** – The number of generation.
- **stats** – A `Statistics` object that is updated inplace, optional.
- **halloffame** – A `HallOfFame` object that will contain the best individuals, optional.
- **verbose** – Whether or not to log the statistics.

**Returns** The final population.

It uses $\lambda = \kappa = \mu$ and goes as follow. It first initializes the population ($P(0)$) by evaluating every individual presenting an invalid fitness. Then, it enters the evolution loop that begins by the selection of the $P(g + 1)$ population. Then the crossover operator is applied on a proportion of $P(g + 1)$ according to the *cxpb* probability, the resulting and the untouched individuals are placed in $P'(g + 1)$. Thereafter, a proportion of $P'(g + 1)$, determined by *mutpb*, is mutated and placed in $P''(g + 1)$, the untouched individuals are transferred $P''(g + 1)$. Finally, those new individuals are evaluated and the evolution loop continues until *ngen* generations are completed. Briefly, the operators are applied in the following order

```
evaluate(population)
for i in range(ngen):
    offspring = select(population)
    offspring = mate(offspring)
    offspring = mutate(offspring)
    evaluate(offspring)
    population = offspring
```

This function expects `toolbox.mate()`, `toolbox.mutate()`, `toolbox.select()` and `toolbox.evaluate()` aliases to be registered in the toolbox.

deap.algorithms.**eaMuPlusLambda**(*population*, *toolbox*, *mu*, *lambda_*, *cxpb*, *mutpb*, *ngen*[, *stats*, *halloffame*, *verbose*])

This is the $(\mu + \lambda)$ evolutionary algorithm.

**Parameters**

- **population** – A list of individuals.
- **toolbox** – A `Toolbox` that contains the evolution operators.
- **mu** – The number of individuals to select for the next generation.
- **lambda_** – The number of children to produce at each generation.
- **cxpb** – The probability that an offspring is produced by crossover.
- **mutpb** – The probability that an offspring is produced by mutation.
- **ngen** – The number of generation.
- **stats** – A `Statistics` object that is updated inplace, optional.
- **halloffame** – A `HallOfFame` object that will contain the best individuals, optional.
- **verbose** – Whether or not to log the statistics.

**Returns** The final population.

First, the individuals having an invalid fitness are evaluated. Then, the evolutionary loop begins by producing *lambda_* offspring from the population, the offspring are generated by a crossover, a mutation or a reproduction proportionally to the probabilities *cxpb*, *mutpb* and 1 - (cxpb + mutpb). The offspring are then evaluated and the next generation population is selected from both the offspring **and** the population. Briefly, the operators are applied as following

```
evaluate(population)
for i in range(ngen):
    offspring = varOr(population, toolbox, lambda_, cxpb, mutpb)
    evaluate(offspring)
    population = select(population + offspring, mu)
```

This function expects `toolbox.mate()`, `toolbox.mutate()`, `toolbox.select()` and `toolbox.evaluate()` aliases to be registered in the toolbox. This algorithm uses the `varOr()` variation.

deap.algorithms.**eaMuCommaLambda**(*population*, *toolbox*, *mu*, *lambda_*, *cxpb*, *mutpb*, *ngen*[, *stats*, *halloffame*, *verbose*])

This is the $(\mu\ ,\ \lambda)$ evolutionary algorithm.

**Parameters**

- **population** – A list of individuals.
- **toolbox** – A `Toolbox` that contains the evolution operators.
- **mu** – The number of individuals to select for the next generation.
- **lambda_** – The number of children to produce at each generation.
- **cxpb** – The probability that an offspring is produced by crossover.
- **mutpb** – The probability that an offspring is produced by mutation.
- **ngen** – The number of generation.
- **stats** – A `Statistics` object that is updated inplace, optional.
- **halloffame** – A `HallOfFame` object that will contain the best individuals, optional.
- **verbose** – Whether or not to log the statistics.

**Returns** The final population.

First, the individuals having an invalid fitness are evaluated. Then, the evolutionary loop begins by producing *lambda_* offspring from the population, the offspring are generated by a crossover, a mutation or a reproduction proportionally to the probabilities *cxpb*, *mutpb* and 1 - (cxpb + mutpb). The offspring are then evaluated and the next generation population is selected **only** from the offspring. Briefly, the operators are applied as following

```
evaluate(population)
for i in range(ngen):
    offspring = varOr(population, toolbox, lambda_, cxpb, mutpb)
    evaluate(offspring)
    population = select(offspring, mu)
```

This function expects `toolbox.mate()`, `toolbox.mutate()`, `toolbox.select()` and `toolbox.evaluate()` aliases to be registered in the toolbox. This algorithm uses the `varOr()` variation.

deap.algorithms.**eaGenerateUpdate**(*toolbox*, *ngen*[, *stats*, *halloffame*, *verbose*])

This is algorithm implements the ask-tell model proposed in [Colette2010], where ask is called *generate* and tell is called *update*.

**Parameters**

- **toolbox** – A `Toolbox` that contains the evolution operators.
- **ngen** – The number of generation.
- **stats** – A `Statistics` object that is updated inplace, optional.
- **halloffame** – A `HallOfFame` object that will contain the best individuals, optional.
- **verbose** – Whether or not to log the statistics.

**Returns** The final population.

The toolbox should contain a reference to the generate and the update method of the chosen strategy.

### 3.3.2 Variations

Variations are smaller parts of the algorithms that can be used separately to build more complex algorithms.

`deap.algorithms.`**`varAnd`**(*population*, *toolbox*, *cxpb*, *mutpb*)

Part of an evolutionary algorithm applying only the variation part (crossover **and** mutation). The modified individuals have their fitness invalidated. The individuals are cloned so returned population is independent of the input population.

**Parameters**

- **population** – A list of individuals to variate.
- **toolbox** – A `Toolbox` that contains the evolution operators.
- **cxpb** – The probability of mating two individuals.
- **mutpb** – The probability of mutating an individual.

**Returns** A list of varied individuals that are independent of their parents.

The variator goes as follow. First, the parental population $P_\mathrm{p}$ is duplicated using the `toolbox.clone()` method and the result is put into the offspring population $P_\mathrm{o}$. A first loop over $P_\mathrm{o}$ is executed to mate consecutive individuals. According to the crossover probability *cxpb*, the individuals $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$ are mated using the `toolbox.mate()` method. The resulting children $\mathbf{y}_i$ and $\mathbf{y}_{i+1}$ replace their respective parents in $P_\mathrm{o}$. A second loop over the resulting $P_\mathrm{o}$ is executed to mutate every individual with a probability *mutpb*. When an individual is mutated it replaces its not mutated version in $P_\mathrm{o}$. The resulting $P_\mathrm{o}$ is returned.

This variation is named *And* because of its propention to apply both crossover and mutation on the individuals. Note that both operators are not applied systematicaly, the resulting individuals can be generated from crossover only, mutation only, crossover and mutation, and reproduction according to the given probabilities. Both probabilities should be in $[0, 1]$.

`deap.algorithms.`**`varOr`**(*population*, *toolbox*, *lambda_*, *cxpb*, *mutpb*)

Part of an evolutionary algorithm applying only the variation part (crossover, mutation **or** reproduction). The modified individuals have their fitness invalidated. The individuals are cloned so returned population is independent of the input population.

**Parameters**

- **population** – A list of individuals to variate.
- **toolbox** – A `Toolbox` that contains the evolution operators.
- **lambda_** – The number of children to produce
- **cxpb** – The probability of mating two individuals.
- **mutpb** – The probability of mutating an individual.

**Returns** A list of varied individuals that are independent of their parents.

The variator goes as follow. On each of the *lambda_* iteration, it selects one of the three operations; crossover, mutation or reproduction. In the case of a crossover, two individuals are selected at random from the parental population $P_\mathrm{p}$, those individuals are cloned using the `toolbox.clone()` method and then mated using the `toolbox.mate()` method. Only the first child is appended to the offspring population $P_\mathrm{o}$, the second child is discarded. In the case of a mutation, one individual is selected at random from $P_\mathrm{p}$, it is cloned and then mutated using using the `toolbox.mutate()` method. The resulting mutant is appended to $P_\mathrm{o}$. In the case of a reproduction, one individual is selected at random from $P_\mathrm{p}$, cloned and appended to $P_\mathrm{o}$.

This variation is named *Or* because an offspring will never result from both operations crossover and mutation. The sum of both probabilities shall be in $[0, 1]$, the reproduction probability is 1 - *cxpb* - *mutpb*.

## 3.4 Covariance Matrix Adaptation Evolution Strategy

## 3.5 Genetic Programming

The `gp` module provides the methods and classes to perform Genetic Programming with DEAP. It essentially contains the classes to build a Genetic Program Tree, and the functions to evaluate it.

This module support both strongly and loosely typed GP.

**class** `deap.gp.`**`PrimitiveTree`**(*content*)
Tree spefically formated for optimization of genetic programming operations. The tree is represented with a list where the nodes are appended in a depth-first order. The nodes appended to the tree are required to define to have an attribute *arity* which defines the arity of the primitive. An arity of 0 is expected from terminals nodes.

**`height`**
Return the height of the tree, or the depth of the deepest node.

**`root`**
Root of the tree, the element 0 of the list.

**`searchSubtree`**(*begin*)
Return a slice object that corresponds to the range of values that defines the subtree which has the element with index *begin* as its root.

**class** `deap.gp.`**`PrimitiveSet`**(*name*, *arity*, *prefix='ARG'*)
Class same as `PrimitiveSetTyped`, except there is no definition of type.

**`addEphemeralConstant`**(*ephemeral*)
Add an ephemeral constant to the set.

**`addPrimitive`**(*primitive*, *arity*, *symbol=None*)
Add primitive *primitive* with arity *arity* to the set.

**`addTerminal`**(*terminal*, *name=None*)
Add a terminal to the set.

**class** `deap.gp.`**`Primitive`**(*primitive*, *args*, *ret*)
Class that encapsulates a primitive and when called with arguments it returns the Python code to call the primitive with the arguments.

```
>>> import operator
>>> pr = Primitive(operator.mul, (int, int), int)
>>> pr.format(1, 2)
'mul(1, 2)'
```

**class** `deap.gp.`**`Operator`**(*operator*, *symbol*, *args*, *ret*)
Class that encapsulates an operator and when called with arguments it returns the Python code to call the operator

with the arguments. It acts as the Primitive class, but instead of returning a function and its arguments, it returns an operator and its operands.

```
>>> import operator
>>> op = Operator(operator.mul, '*', (int, int), int)
>>> op.format(1, 2)
'(1 * 2)'
>>> op2 = Operator(operator.neg, '-', (int,), int)
>>> op2.format(1)
'-(1)'
```

**class** deap.gp.**Terminal**(*terminal*, *symbolic*, *ret*)
Class that encapsulates terminal primitive in expression. Terminals can be values or 0-arity functions.

**class** deap.gp.**Ephemeral**(*func*, *ret*)
Class that encapsulates a terminal which value is set at run-time. The value of the *Ephemeral* can be regenerated by creating a new Ephemeral object with the same parameters (func and ret).

deap.gp.**stringify**(*expr*)
Evaluate the expression *expr* into a string.

deap.gp.**evaluate**(*expr*, *pset*)
Evaluate the expression *expr* into Python code object.

deap.gp.**lambdify**(*expr*, *pset*)
Return a lambda function of the expression *expr*.

---

**Note:** This function is a stripped version of the lambdify function of sympy0.6.6.

---

deap.gp.**lambdifyADF**(*expr*)
Return a lambda function created from a list of trees. The first element of the list is the main tree, and the following elements are automatically defined functions (ADF) that can be called by the first tree.

**class** deap.gp.**PrimitiveSetTyped**(*name*, *in_types*, *ret_type*, *prefix='ARG'*)
Class that contains the primitives that can be used to solve a Strongly Typed GP problem. The set also defined the researched function return type, and input arguments type and number.

**addADF**(*adfset*)
Add an Automatically Defined Function (ADF) to the set.

*adfset* is a PrimitiveSetTyped containing the primitives with which the ADF can be built.

**addEphemeralConstant**(*ephemeral*, *ret_type*)
Add an ephemeral constant to the set. An ephemeral constant is a no argument function that returns a random value. The value of the constant is constant for a Tree, but may differ from one Tree to another.

*ephemeral* function with no arguments that returns a random value. *ret_type* is the type of the object returned by the function.

**addPrimitive**(*primitive*, *in_types*, *ret_type*, *symbol=None*)
Add a primitive to the set.

*primitive* is a callable object or a function. *in_types* is a list of argument's types the primitive takes. *ret_type* is the type returned by the primitive.

**addTerminal**(*terminal*, *ret_type*, *name=None*)
Add a terminal to the set.

*terminal* is an object, or a function with no arguments. *ret_type* is the type of the terminal. *name* defines the name of the terminal in the expression. This should be used : to define named constant (i.e.: pi);

to speed the evaluation time when the object is long to build; when the object does not have a __repr__ functions that returns the code to build the object; when the object class is not a Python built-in.

**renameArguments**(**kargs)
Rename function arguments with new names from *kargs*.

**terminalRatio**
Return the ratio of the number of terminals on the number of all kind of primitives.

## 3.6 Distributed Task Manager Overview

Deprecated since version 0.9: DTM will be removed in version 1.0. Please consider using SCOOP instead for distributing your evolutionnary algorithms or other software. DTM is a distributed task manager which works over many communication layers. Currently, all modern MPI implementations are supported through mpi4py, and an experimental TCP backend is available.

**Warning:** As on version 0.2, DTM is still in *alpha stage*, meaning that some specific bugs may appear; the communication and operation layers are quite stable, though. Feel free to report bugs and performance issues if you isolate a problematic situation.

### 3.6.1 DTM Main Features

DTM has some very interesting features :

- Offers a similar interface to the Python's multiprocessing module
- Automatically balances the load between workers (and therefore supports heterogeneous networks and different task duration)
- Supports an arbitrary number of workers without changing a byte in the program
- Abstracts the user from the communication management (the same program can be run over MPI, TCP or multiprocessing just by changing the communication manager)
- Provides easy-to-use parallelization paradigms
- Offers a trace mode, which can be used to tune the performance of the running program (still experimental)

### 3.6.2 Introductory example

First, lets take a look to a very simple distribution example. The sequential program we want to parallelize reads as follow :

```
def op(x):
    return x + 1./x     # Or any operation

if __name__ == "__main__":
    nbrs = range(1, 1000)
    results = map(op, nbrs)
```

This program simply applies an arbitrary operation to each item of a list. Although it is a very trivial program (and therefore would not benefit from a parallelization), lets assume we want to distribute the workload over a cluster. We just import the task manager, and use the DTM parallel version of `map()` :

```
from deap import dtm

def op(x):
    return x + 1./x      # Or any operation

def main():
    nbrs = range(1, 1000)
    results = dtm.map(op, nbrs)

dtm.start(main)
```

And we are done! This program can now run over MPI, with an arbitrary number of processors, without changing anything. We just run it like a normal MPI program (here with OpenMPI)

```
$ mpirun -n * python myProgram.py
```

The operation done in the op() function can be virtually any operation, including other DTM calls (which in turn may also spawn sub-tasks, and so on).

---

**Note:** The encapsulation of the main execution code into a function is required by DTM, in order to be able to control which worker will start the execution.

---

### 3.6.3 Functions documentation

**class** `deap.dtm.manager.`**`Control`**

Control is the main DTM class. The dtm object you receive when you use `from deap import dtm` is a proxy over an instance of this class.

Most of its methods are used by your program, in the execution tasks; however, two of thems (start() and setOptions()) MUST be called in the MainThread (i.e. the thread started by the Python interpreter).

As this class is instancied directly in the module, initializer takes no arguments.

**`apply`**(*function*, **args*, ***kwargs*)

Equivalent of the `apply()` built-in function. It blocks till the result is ready. Given this blocks, `apply_async()` is better suited for performing work in parallel. Additionally, the passed in function is only executed in one of the workers of the pool.

**`apply_async`**(*function*, **args*, ***kwargs*)

A non-blocking variant of the `apply()` method which returns a `AsyncResult` object.

**`filter`**(*function*, *iterable*)

Same behavior as the built-in `filter()`. The filtering is done localy, but the computation is distributed.

**`getWorkerId`**()

Return a unique ID for the current worker. Depending of the communication manager type, it can be virtually any Python immutable type.

---

**Note:** With MPI, the value returned is the MPI slot number.

---

**`imap`**(*function*, **iterables*)

An equivalent of `itertools.imap()`.

This function creates an iterator which return results of the map operation, in order. One of the main benefits of this function against the traditionnal map function is that the first result may be yielded even if the computation of the others is not finished.

**imap_unordered**(*function*, *iterable*, *chunksize=1*)
An equivalent of `itertools.imap()`, but returning the results in an arbitrary order. It launchs *chunksize* tasks, and wait for the completion of one the them. It then yields the result of the first completed task : therefore, the return order can be predicted only if you use no more than one worker or if chunksize = 1.

In return, as this function manages to have always *chunksize* tasks launched, it may speed up the evaluation when the execution times are very different. In this case, one should put a relatively large value of chunksize.

**map**(*function*, **iterables*, ***kwargs*)
A parallel equivalent of the `map()` built-in function. It blocks till the result is ready. This method chops the iterables into a number of chunks determined by DTM in order to get the most efficient use of the workers. It takes any number of iterables (though it will shrink all of them to the length of the smallest one), and any others kwargs that will be transmitted as is to the *function* target.

**map_async**(*function*, **iterables*, ***kwargs*)
A non-blocking variant of the `map()` method which returns a `AsyncResult` object. It takes any number of iterables (though it will shrink all of them to the length of the smallest one), and any others kwargs that will be transmitted as is to the *function* target.

---

**Note:** A callback argument is not implemented, since it would make DTM unable to dispatch tasks and handle messages while it executes.

---

**repeat**(*function*, *n*, **args*, ***kwargs*)
Repeat the function *function n* times, with given args and keyworded args. Return a list containing the results.

**setOptions**(**args*, ***kwargs*)
Set a DTM global option.

> **Warning:** This function must be called BEFORE `start()`. It is also the user responsability to ensure that the same option is set on every worker.

**Currently, the supported options are :**

- **communicationManager** : can be *deap.dtm.mpi4py* (default) or *deap.dtm.commManagerTCP*.
- **loadBalancer** : currently only the default *PDB* is available.
- **printSummary** : if set, DTM will print a task execution summary at the end (mean execution time of each tasks, how many tasks did each worker do, ...)
- **setTraceMode** : if set, will enable a special DTM tracemode. In this mode, DTM logs all its activity in XML files (one by worker). Mainly for DTM debug purpose, but can also be used for profiling.
- **traceDir** : only available when setTraceMode is True. Path to the directory where DTM should save its log.
- **taskGranularity** : set the granularity of the parallelism. It is specified in seconds, and represents the minimum amount of time to make a "task". If a task duration is smaller than this minimum, then DTM will try to combine two or more of those tasks to reach the wanted level of granularity. This can be very useful and may greatly reduce distribution overhead if some of your tasks are very small, or if you are working on a low-performance network. As on DTM 0.3, this only applies to synchronous calls (map, repeat, filter).

This function can be called more than once. Any unknown parameter will have no effect.

**start**(*initialTarget*, **args*, ***kwargs*)
Start the execution with the target *initialTarget*. Calling this function create and launch the first task on the root worker (defined by the communication manager, for instance, with MPI, the root worker is the worker with rank 0.).

> **Warning:** This function must be called only ONCE, and after the target has been parsed by the Python interpreter.

**testAll**(*reqList*=[ ])
Check whether all pending asynchronous tasks in list *reqList* are done. It does not lock if it is not the case, but returns False.

---

**Note:** If *reqList* is not specified or an empty list, DTM will test the completion of all the current asynchronous tasks.

---

**testAny**(*reqList*=[ ])
Test the completion of any pending asynchronous task in list *reqList*, then return the `AsyncResult` object of the last finished asynchronous task (say, the most recent one). If there is no pending asynchronous tasks, this function will return None.

---

**Note:** If *reqList* is not specified or an empty list, DTM will test the completion of any of the current asynchronous tasks.

---

> **Warning:** This function always returns the same task (the last) if called more than once with the same parameters (i.e. it does not delete the state of a completed task when called, so a second call will produce the same output). It is the user responsability to provide a *reqList* containing only the tasks which he does not know whether they are completed or not. Similarly, multiple calls without specifing the *reqList* param will always return the last finished asynchronous task.

**waitAll**(*reqList*=[ ])
Wait for all pending asynchronous results in list *reqList*. When this function returns, DTM guarantees that all ready() call those asynchronous tasks will return true.

---

**Note:** If *reqList* is not specified or an empty list, DTM will wait over all the current asynchronous tasks.

---

**waitAny**(*reqList*=[ ])
Wait for any pending asynchronous tasks in list *reqList*, then return the `AsyncResult` object of the last finished asynchronous task (say, the most recent one). If there is no pending asynchronous tasks, this function will return None.

---

**Note:** If *reqList* is not specified or an empty list, DTM will wait over any of the current asynchronous tasks.

---

> **Warning:** This function only guarantees that at least one of the asynchronous task will be done when it returns, but actually many others may have been done. In this case, this function returns only the last one, even if called more than once.

**class** `deap.dtm.manager.`**AsyncResult**(*control*, *waitingInfo*, *taskKey*, *resultIsIterable*)
The class of the result returned by `map_async()` and `apply_async()`.

**get**()
Return the result when it arrives. If an exception has been raised in the child task (and thus catched by DTM), then it will be raised when this method will be called.

---

**Note:** This is a blocking call : caller will wait in this function until the result is ready. To check for the avaibility of the result, use `ready()`.

---

**ready**()
Return whether the asynchronous task has completed.

**successful**()
Return whether the task completed without error. Will raise AssertionError if the result is not ready.

**wait**()
Wait until the result is available. When this function returns, DTM guarantees that a call to `ready()` will return true.

### 3.6.4 DTM launching

DTM can support many communication environment. The only real restriction on the choice of a communication backend is that is must provide an access from each worker to every other (that is, the worker 0 must be able to directly communicate with all other workers, and so for the worker 1, 2, etc.). Currently, two main backends are available : one using MPI through the mpi4py layer, and another using TCP (with SSH for the launch process).

If you already have a functionnal MPI installation, you should choose the mpi4py backend, as it can provide better performances in some cases and easier configuration. On the other side, MPI implementations may cause some strange errors when you use low-level system operations (such as fork). In any case, both backends should be ready for production use, and as the backend switch is as easy as changing only one line in your script, this choice should not be an issue.

#### Launch DTM with the mpi4py backend

When this backend is used, DTM delegates the start-up process to MPI. In this way, any MPI option can be used. For instance :

```
mpirun -n 32 -hostfile myHosts -npernode 4 -bind-to-core python yourScript.py --yourScriptParams ...
```

Will launch your script on 32 workers, distributed over the hosts listed in 'myHosts', without exceeding 4 workers by host, and will bind DTM workers to a CPU core. The bind-to-core option can provide a good performance improvement in some environments, and does not affect the behavior of DTM at all.

#### Launch DTM with the pure-TCP backend

The TCP backend includes a launcher which works with SSH in order to start remote DTM workers. Therefore, your execution environment must provide a SSH access to every host, and a shared file system such as NFS (or, at least, ensure that the script you want to execute and the DTM libraries are located in a common path for every worker). You also have to provide a host file which follows the same synthax as the MPI host files, for instance :

```
firstComputer.myNetwork.com slots=4
otherComputer.myNetwork.com slots=6
221.32.118.3 slots=4
```

**Warning:** The hostnames / addresses you write in this file must be accessible and translable on every machine used. For instance, putting 'localhost' in this file among other remote hosts **will fail** because each host will try to bind ports for 'localhost' (which will fail, as this is not a network-accessible address).

Then you can launch DTM with this file :

```
python myScript.py --dtmTCPhosts=myHostFile
```

---

**Note:** Do not forget to explicitly set the communication manager to **deap.dtm.commManagerTCP** :

```
dtm.setOptions(communicationManager="deap.dtm.commManagerTCP")
```

---

This backend can also be useful if you want to launch local jobs. If your hostfile contains only 'localhost' or '127.0.0.1', DTM will start all the workers locally, without using SSH.

### 3.6.5 Troubleshooting and Pitfalls

Here are the most common errors or problems reported with DTM. Some of them are caused by a bad use of the task manager, others are limitations from the libraries and programs used by DTM.

#### Isolation per worker

In DTM, the atomic independent working units are called workers. They are separate processes, and do not share any information other than those from the communications (explicitly called). Therefore, two variables cannot interfere if they are used in different workers, and your program should not rely on this. Thus, one has to be extremely careful about which data is global, and which is local to a task. For instance, consider the following program :

```
from deap import dtm

foo = [0]

def bar(n):
    foo[0] += n
    return foo[0]

def main():
    listNbr = range(30)
    results = dtm.map(bar, listNbr)
    print("Results : " + str(results))

dtm.start(main)
```

Although it is syntactically correct, it may not produce the result you are waiting for. On a serial evaluation (using the built-in `map()` function), it simply produces a list containing the sums of numbers from 0 to 30 (it is a quite odd approach, but it works) :

```
Results : [0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, 153, 171, 190, 210, 231
```

But with DTM, as foo is not shared between workers, the program generate a completely unpredictable output, for instance :

```
Results : [0, 1, 2, 5, 5, 10, 11, 17, 25, 20, 30, 36, 48, 43, 57, 72, 88, 65, 83, 107, 127, 104, 126,
```

The reverse problem should also be taken into account. If an object keeps its state, it is generally not a good idea to make it global (accessible from all tasks). For instance, if you create a log like this :

```
from deap import dtm

logfile = open('myLogFile', 'w+')

def task1(args):
    # [...]
    logfile.write("Log task 1")
    # [...]
def task2(args):
    # [...]
    logfile.write("Log task 2")
    # [...]

def main():
    listNbr = range(100)
    statusTask1 = dtm.map_async(task1, listNBr)
    statusTask2 = dtm.map_async(task2, listNBr)
    # [...]

dtm.start()
```

You may experience some unusual outputs, as task1 and task2 writings will probably overlap (because they use the same resource if, by chance, they are executed on the same worker, which will probably happens). In doubt, use local variables.

### Exceptions

When an Python exception occurs during a task execution, DTM catchs it (and try to run another task on this worker). This exception is then raised in the *parent task*. If there is no such task (the task where the exception occurs is the root task), then it is thrown and DTM stops its execution.

The moment when the exception will be raised in the parent tasks depends on the child task type : if it is a synchronous call (like `apply()` or `map()`), it is raised when the parent awake (i.e. as if it has been raised by the DTM function itself). If it is an asynchronous call (like `apply_async()` or `map_async()`), the exception is raised when the parent task performs a `get()` on the `AsyncResult` object. Also, the `successful()` will return *False* if an exception occured, without raising it.

---

**Note:** When DTM catches an exception, it outputs a warning on the standard error output stating the exception type and arguments. This warning does not mean that the exception has been raised in the parent task (actually, in some situations, it may take a lot of time if every workers are busy); it is logged only for information purpose.

---

### MPI and threads

Recent MPI implementations supports four levels of threading : single, funneled, serialized and multiple. However, many environments (like Infiniband backend) do not support other level than single. In that case, if you use the mpi4py backend, make sure that mpi4py does not initialize MPI environment in another mode than single (as DTM has been designed so that only the communication thread makes MPI calls, this mode works well even if there is more than one active thread in a DTM worker). This setting can be changed in the file "site-packages/mpi4py/rc.py", with the variable *thread_level*.

### Cooperative multitasking

DTM works on a cooperative philosophy. There is no preemption system to interrupt an executing thread (and eventually starts one with a higher priority). When a task begins its execution, the worker will execute it until the task returns or makes a DTM synchronous call, like map or apply. If the task enters an infinite loop or reaches a dead-lock state, then the worker will also be in dead-lock – it will be able to transfer its other tasks to other workers though. DTM is not a fair scheduler, and thus cannot guarantee any execution delay or avoid any case of starvation; it just tries to reach the best execution time knowing some information about the tasks.

### Pickling

When dealing with non trivial programs, you may experience error messages like this :

```
PicklingError: Can't pickle <class '__main__.****'>: attribute lookup __main__.**** failed
```

This is because DTM makes use of the Python `pickle` module in order to serialize data and function calls (so they can be transferred from one worker to another). Although the pickle module is very powerful (it handles recursive and shared objects, multiple references, etc.), it has some limitations. Most of the time, a pickling error can be easily solved by adding __setstate__() and __getstate__() methods to the problematic class (see the Python documentation for more details about the pickling protocol).

This may also be used to accelerate pickling : by defining your own pickling methods, you can speedup the pickling operation (the same way you can speedup the deepcopy operation by defining your own *__deepcopy__()* method. If your program use thousands of objects from the same class, it may be worthwhile.

Take also note of the following Python interpreter limitations :

- As on version 2.6, partial functions cannot be pickled. Python 2.7 works fine.
- Lambda functions cannot be pickled in every Python version (up to 3.2). User should use normal functions, or tools from functools, or ensure that its parallelization never need to explicitly transfer a lambda function (not its result, but the lambda object itself) from a worker to another.
- Functions are usually never pickled : they are just referenced, and should be importable in the unpickling environment, even if they are standard functions (defined with the keyword **def**). For instance, consider this (faulty) code :

```
from deap import dtm
def main():
    def bar(n):
        return n**2

    listNbr = range(30)
    results = dtm.map(bar, listNbr)

dtm.start(main)
```

On the execution, this will produce an error like :

```
TypeError: can't pickle function objects
```

Because the pickler will not be able to find a global reference to the function *bar()*. The same restriction applies on classes and modules.

### Asynchronous tasks and active waiting

DTM supports both synchronous and asynchronous tasks (that do not stop the parent task). For the asynchronous tasks, DTM returns an object with an API similar to the Python `multiprocessing.pool.AsyncResult`. This

object offers some convenient functions to wait on a result, or test if the task is done. However, some issues may appear in DTM with a program like that :

```
from deap import dtm
def myTask(param):
    # [...] Long treatment
    return param+2

def main():
    listTasks = range(100)
    asyncReturn = dtm.map_async(myTask, listTasks)

    while not asyncReturn.ready():
        continue

    # Other instructions...

dtm.start(main)
```

This active waiting (by looping while the result is not available), although syntactically valid, might produce unexpected "half-deadlocks". Keep in mind that DTM is not a preemptive system : if you load a worker only for waiting on another, the load balancer may not work properly. For instance, it may consider that as the parent task is still working, the asynchronous child tasks can still wait, or that one of the child tasks should remain on the same worker than its parent to balance the load between workers. As the load balancer is not completely deterministic, the child tasks should eventually complete, but in an unpredictable time.

It is way better to do something like this :

```
def main():
    listTasks = range(100)
    asyncReturn = dtm.map_async(myTask, listTasks)

    asyncReturn.wait()  # or dtm.waitForAll()
    # [...]
```

By calling one of these functions, you effectively notify DTM that you are now waiting for those asynchronous tasks, and willing to let the worker do another job. The call will then return to your parent function when all the results will be available.

## 3.7 Benchmarks

Regroup typical EC benchmarks functions to import easily and benchmark examples.

| Single Objective Continuous | Multi Objective Continuous | Binary |
|---|---|---|
| cigar() | fonseca() | chuang_f1() |
| plane() | kursawe() | chuang_f2() |
| sphere() | schaffer_mo() | chuang_f3() |
| rand() | dtlz1() | royal_road1() |
| ackley() | dtlz2() | royal_road2() |
| bohachevsky() | dtlz3() | |
| griewank() | dtlz4() | |
| h1() | zdt1() | |
| himmelblau() | zdt2() | |
| rastrigin() | zdt3() | |
| rastrigin_scaled() | zdt4() | |
| rastrigin_skew() | zdt6() | |
| rosenbrock() | | |
| schaffer() | | |
| schwefel() | | |
| shekel() | | |

### 3.7.1 Continuous Optimization

deap.benchmarks.**cigar**(*individual*)
Cigar test objective function.

| | |
|---|---|
| Type | minimization |
| Range | none |
| Global optima | $x_i = 0, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$ |
| Function | $f(\mathbf{x}) = x_0^2 + 10^6 \sum_{i=1}^{N} x_i^2$ |

deap.benchmarks.**plane**(*individual*)
Plane test objective function.

| | |
|---|---|
| Type | minimization |
| Range | none |
| Global optima | $x_i = 0, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$ |
| Function | $f(\mathbf{x}) = x_0$ |

deap.benchmarks.**sphere**(*individual*)
Sphere test objective function.

| | |
|---|---|
| Type | minimization |
| Range | none |
| Global optima | $x_i = 0, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$ |
| Function | $f(\mathbf{x}) = \sum_{i=1}^{N} x_i^2$ |

deap.benchmarks.**rand**(*individual*)
Random test objective function.

| | |
|---|---|
| Type | minimization or maximization |
| Range | none |
| Global optima | none |
| Function | $f(\mathbf{x}) = \texttt{random}(0, 1)$ |

deap.benchmarks.**ackley**(*individual*)
Ackley test objective function.

| | |
|---|---|
| Type | minimization |
| Range | $x_i \in [-15, 30]$ |
| Global optima | $x_i = 0, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$ |
| Function | $f(\mathbf{x}) = 20 - 20\exp\left(-0.2\sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}\right) + e - \exp\left(\frac{1}{N}\sum_{i=1}^{N}\cos(2\pi x_i)\right)$ |

deap.benchmarks.**bohachevsky**(*individual*)

Bohachevsky test objective function.

| | |
|---|---|
| Type | minimization |
| Range | $x_i \in [-100, 100]$ |
| Global optima | $x_i = 0, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$ |
| Function | $f(\mathbf{x}) = \sum_{i=1}^{N-1}(x_i^2 + 2x_{i+1}^2 - 0.3\cos(3\pi x_i) - 0.4\cos(4\pi x_{i+1}) + 0.7)$ |

deap.benchmarks.**griewank**(*individual*)

Griewank test objective function.

| | |
|---|---|
| Type | minimization |
| Range | $x_i \in [-600, 600]$ |
| Global optima | $x_i = 0, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$ |
| Function | $f(\mathbf{x}) = \frac{1}{4000}\sum_{i=1}^{N} x_i^2 - \prod_{i=1}^{N}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$ |

deap.benchmarks.**h1**(*individual*)

Simple two-dimensional function containing several local maxima. From: The Merits of a Parallel Genetic Algorithm in Solving Hard Optimization Problems, A. J. Knoek van Soest and L. J. R. Richard Casius, J. Biomech. Eng. 125, 141 (2003)

| | |
|---|---|
| Type | maximization |
| Range | $x_i \in [-100, 100]$ |
| Global optima | $\mathbf{x} = (8.6998, 6.7665), f(\mathbf{x}) = 2$ |
| Function | $f(\mathbf{x}) = \frac{\sin(x_1 - \frac{x_2}{8})^2 + \sin(x_2 + \frac{x_1}{8})^2}{\sqrt{(x_1 - 8.6998)^2 + (x_2 - 6.7665)^2} + 1}$ |

deap.benchmarks.**himmelblau**(*individual*)

The Himmelblau's function is multimodal with 4 defined minimums in $[-6, 6]^2$.

| | |
|---|---|
| Type | minimization |
| Range | $x_i \in [-6, 6]$ |
| Global optima | $\mathbf{x}_1 = (3.0, 2.0), f(\mathbf{x}_1) = 0$<br>$\mathbf{x}_2 = (-2.805118, 3.131312), f(\mathbf{x}_2) = 0$<br>$\mathbf{x}_3 = (-3.779310, -3.283186), f(\mathbf{x}_3) = 0$<br>$\mathbf{x}_4 = (3.584428, -1.848126), f(\mathbf{x}_4) = 0$ |
| Function | $f(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$ |

deap.benchmarks.**rastrigin**(*individual*)

Rastrigin test objective function.

| | |
|---|---|
| Type | minimization |
| Range | $x_i \in [-5.12, 5.12]$ |
| Global optima | $x_i = 0, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$ |
| Function | $f(\mathbf{x}) = 10N\sum_{i=1}^{N} x_i^2 - 10\cos(2\pi x_i)$ |

deap.benchmarks.**rastrigin_scaled**(*individual*)

Scaled Rastrigin test objective function.

$f_{\text{RastScaled}}(\mathbf{x}) = 10N + \sum_{i=1}^{N}\left(10^{\left(\frac{i-1}{N-1}\right)} x_i\right)^2 x_i)^2 - 10\cos\left(2\pi 10^{\left(\frac{i-1}{N-1}\right)} x_i\right)$

deap.benchmarks.**rastrigin_skew**(*individual*)
Skewed Rastrigin test objective function.

$f_{\text{RastSkew}}(\mathbf{x}) = 10N \sum_{i=1}^{N} \left(y_i^2 - 10\cos(2\pi x_i)\right)$

with $y_i = \begin{cases} 10 \cdot x_i & \text{if } x_i > 0, \\ x_i & \text{otherwise} \end{cases}$

deap.benchmarks.**rosenbrock**(*individual*)
Rosenbrock test objective function.

| | |
|---|---|
| Type | minimization |
| Range | none |
| Global optima | $x_i = 1, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$ |
| Function | $f(\mathbf{x}) = \sum_{i=1}^{N-1} (1 - x_i)^2 + 100(x_{i+1} - x_i^2)^2$ |

deap.benchmarks.**schaffer**(*individual*)
Schaffer test objective function.

| | |
|---|---|
| Type | minimization |
| Range | $x_i \in [-100, 100]$ |
| Global optima | $x_i = 0, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$ |
| Function | $f(\mathbf{x}) = \sum_{i=1}^{N-1} (x_i^2 + x_{i+1}^2)^{0.25} \cdot \left[\sin^2(50 \cdot (x_i^2 + x_{i+1}^2)^{0.10}) + 1.0\right]$ |

deap.benchmarks.**schwefel**(*individual*)
Schwefel test objective function.

| | |
|---|---|
| Type | minimization |
| Range | $x_i \in [-500, 500]$ |
| Global optima | $x_i = 420.96874636, \forall i \in \{1 \dots N\}, f(\mathbf{x}) = 0$ |
| Function | $f(\mathbf{x}) = 418.9828872724339 \cdot N - \sum_{i=1}^{N} x_i \sin\left(\sqrt{\lvert x_i \rvert}\right)$ |

deap.benchmarks.**shekel**(*individual*, *a*, *c*)
The Shekel multimodal function can have any number of maxima. The number of maxima is given by the length of any of the arguments *a* or *c*, *a* is a matrix of size $M \times N$, where *M* is the number of maxima and *N* the number of dimensions and *c* is a $M \times 1$ vector. The matrix $\mathcal{A}$ can be seen as the position of the maxima and the vector $\mathbf{c}$, the width of the maxima.

$f_{\text{Shekel}}(\mathbf{x}) = \sum_{i=1}^{M} \frac{1}{c_i + \sum_{j=1}^{N} (x_j - a_{ij})^2}$

The following figure uses

$\mathcal{A} = \begin{bmatrix} 0.5 & 0.5 \\ 0.25 & 0.25 \\ 0.25 & 0.75 \\ 0.75 & 0.25 \\ 0.75 & 0.75 \end{bmatrix}$ and $\mathbf{c} = \begin{bmatrix} 0.002 \\ 0.005 \\ 0.005 \\ 0.005 \\ 0.005 \end{bmatrix}$, thus defining 5 maximums in $\mathbb{R}^2$.

### Multi-objective

deap.benchmarks.**fonseca**(*individual*)
Fonseca and Fleming's multiobjective function. From: C. M. Fonseca and P. J. Fleming, "Multiobjective optimization and multiple constraint handling with evolutionary algorithms – Part II: Application example", IEEE Transactions on Systems, Man and Cybernetics, 1998.

$f_{\text{Fonseca1}}(\mathbf{x}) = 1 - e^{-\sum_{i=1}^{3} (x_i - \frac{1}{\sqrt{3}})^2}$

$f_{\text{Fonseca2}}(\mathbf{x}) = 1 - e^{-\sum_{i=1}^{3} (x_i + \frac{1}{\sqrt{3}})^2}$

deap.benchmarks.**kursawe**(*individual*)
Kursawe multiobjective function.

$f_{\text{Kursawe1}}(\mathbf{x}) = \sum_{i=1}^{N-1} -10e^{-0.2\sqrt{x_i^2+x_{i+1}^2}}$

$f_{\text{Kursawe2}}(\mathbf{x}) = \sum_{i=1}^{N} |x_i|^{0.8} + 5\sin(x_i^3)$

deap.benchmarks.**schaffer_mo**(*individual*)
Schaffer's multiobjective function on a one attribute *individual*. From: J. D. Schaffer, "Multiple objective optimization with vector evaluated genetic algorithms", in Proceedings of the First International Conference on Genetic Algorithms, 1987.

$f_{\text{Schaffer1}}(\mathbf{x}) = x_1^2$

$f_{\text{Schaffer2}}(\mathbf{x}) = (x_1 - 2)^2$

deap.benchmarks.**dtlz1**(*individual*, *obj*)
DTLZ1 mutliobjective function. It returns a tuple of *obj* values. The individual must have at least *obj* elements. From: K. Deb, L. Thiele, M. Laumanns and E. Zitzler. Scalable Multi-Objective Optimization Test Problems. CEC 2002, p. 825 - 830, IEEE Press, 2002.

$g(\mathbf{x}_m) = 100\left(|\mathbf{x}_m| + \sum_{x_i \in \mathbf{x}_m}\left((x_i - 0.5)^2 - \cos(20\pi(x_i - 0.5))\right)\right)$

$f_{\text{DTLZ11}}(\mathbf{x}) = \frac{1}{2}(1 + g(\mathbf{x}_m))\prod_{i=1}^{m-1} x_i$

$f_{\text{DTLZ12}}(\mathbf{x}) = \frac{1}{2}(1 + g(\mathbf{x}_m))(1 - x_{m-1})\prod_{i=1}^{m-2} x_i$

...

$f_{\text{DTLZ1}m-1}(\mathbf{x}) = \frac{1}{2}(1 + g(\mathbf{x}_m))(1 - x_2)x_1$

$f_{\text{DTLZ1}m}(\mathbf{x}) = \frac{1}{2}(1 - x_1)(1 + g(\mathbf{x}_m))$

Where $m$ is the number of objectives and $\mathbf{x}_m$ is a vector of the remaining attributes $[x_m \ \ldots \ x_n]$ of the individual in $n > m$ dimensions.

deap.benchmarks.**dtlz2**(*individual*, *obj*)
DTLZ2 mutliobjective function. It returns a tuple of *obj* values. The individual must have at least *obj* elements. From: K. Deb, L. Thiele, M. Laumanns and E. Zitzler. Scalable Multi-Objective Optimization Test Problems. CEC 2002, p. 825 - 830, IEEE Press, 2002.

$g(\mathbf{x}_m) = \sum_{x_i \in \mathbf{x}_m}(x_i - 0.5)^2$

$f_{\text{DTLZ21}}(\mathbf{x}) = (1 + g(\mathbf{x}_m))\prod_{i=1}^{m-1}\cos(0.5x_i\pi)$

$f_{\text{DTLZ22}}(\mathbf{x}) = (1 + g(\mathbf{x}_m))\sin(0.5x_{m-1}\pi)\prod_{i=1}^{m-2}\cos(0.5x_i\pi)$

...

$f_{\text{DTLZ2}m}(\mathbf{x}) = (1 + g(\mathbf{x}_m))\sin(0.5x_1\pi)$

Where $m$ is the number of objectives and $\mathbf{x}_m$ is a vector of the remaining attributes $[x_m \ \ldots \ x_n]$ of the individual in $n > m$ dimensions.

deap.benchmarks.**dtlz3**(*individual*, *obj*)
DTLZ3 mutliobjective function. It returns a tuple of *obj* values. The individual must have at least *obj* elements. From: K. Deb, L. Thiele, M. Laumanns and E. Zitzler. Scalable Multi-Objective Optimization Test Problems. CEC 2002, p. 825 - 830, IEEE Press, 2002.

$g(\mathbf{x}_m) = 100\left(|\mathbf{x}_m| + \sum_{x_i \in \mathbf{x}_m}\left((x_i - 0.5)^2 - \cos(20\pi(x_i - 0.5))\right)\right)$

$f_{\text{DTLZ31}}(\mathbf{x}) = (1 + g(\mathbf{x}_m))\prod_{i=1}^{m-1}\cos(0.5x_i\pi)$

$f_{\text{DTLZ32}}(\mathbf{x}) = (1 + g(\mathbf{x}_m))\sin(0.5x_{m-1}\pi)\prod_{i=1}^{m-2}\cos(0.5x_i\pi)$

...

$f_{\text{DTLZ3}m}(\mathbf{x}) = (1 + g(\mathbf{x}_m)) \sin(0.5x_1\pi)$

Where $m$ is the number of objectives and $\mathbf{x}_m$ is a vector of the remaining attributes $[x_m \ \ldots \ x_n]$ of the individual in $n > m$ dimensions.

deap.benchmarks.**dtlz4**(*individual*, *obj*, *alpha*)

DTLZ4 mutliobjective function. It returns a tuple of *obj* values. The individual must have at least *obj* elements. The *alpha* parameter allows for a meta-variable mapping in `dtlz2()` $x_i \rightarrow x_i^\alpha$, the authors suggest $\alpha = 100$. From: K. Deb, L. Thiele, M. Laumanns and E. Zitzler. Scalable Multi-Objective Optimization Test Problems. CEC 2002, p. 825 - 830, IEEE Press, 2002.

$g(\mathbf{x}_m) = \sum_{x_i \in \mathbf{x}_m} (x_i - 0.5)^2$

$f_{\text{DTLZ41}}(\mathbf{x}) = (1 + g(\mathbf{x}_m)) \prod_{i=1}^{m-1} \cos(0.5x_i^\alpha\pi)$

$f_{\text{DTLZ42}}(\mathbf{x}) = (1 + g(\mathbf{x}_m)) \sin(0.5x_{m-1}^\alpha\pi) \prod_{i=1}^{m-2} \cos(0.5x_i^\alpha\pi)$

...

$f_{\text{DTLZ4}m}(\mathbf{x}) = (1 + g(\mathbf{x}_m)) \sin(0.5x_1^\alpha\pi)$

Where $m$ is the number of objectives and $\mathbf{x}_m$ is a vector of the remaining attributes $[x_m \ \ldots \ x_n]$ of the individual in $n > m$ dimensions.

deap.benchmarks.**zdt1**(*individual*)

ZDT1 multiobjective function.

$g(\mathbf{x}) = 1 + \frac{9}{n-1} \sum_{i=2}^{n} x_i$

$f_{\text{ZDT11}}(\mathbf{x}) = x_1$

$f_{\text{ZDT12}}(\mathbf{x}) = g(\mathbf{x}) \left[1 - \sqrt{\frac{x_1}{g(\mathbf{x})}}\right]$

deap.benchmarks.**zdt2**(*individual*)

ZDT2 multiobjective function.

$g(\mathbf{x}) = 1 + \frac{9}{n-1} \sum_{i=2}^{n} x_i$

$f_{\text{ZDT21}}(\mathbf{x}) = x_1$

$f_{\text{ZDT22}}(\mathbf{x}) = g(\mathbf{x}) \left[1 - \left(\frac{x_1}{g(\mathbf{x})}\right)^2\right]$

deap.benchmarks.**zdt3**(*individual*)

ZDT3 multiobjective function.

$g(\mathbf{x}) = 1 + \frac{9}{n-1} \sum_{i=2}^{n} x_i$

$f_{\text{ZDT31}}(\mathbf{x}) = x_1$

$f_{\text{ZDT32}}(\mathbf{x}) = g(\mathbf{x}) \left[1 - \sqrt{\frac{x_1}{g(\mathbf{x})}} - \frac{x_1}{g(\mathbf{x})} \sin(10\pi x_1)\right]$

deap.benchmarks.**zdt4**(*individual*)

ZDT4 multiobjective function.

$g(\mathbf{x}) = 1 + 10(n - 1) + \sum_{i=2}^{n} \left[x_i^2 - 10\cos(4\pi x_i)\right]$

$f_{\text{ZDT41}}(\mathbf{x}) = x_1$

$f_{\text{ZDT42}}(\mathbf{x}) = g(\mathbf{x}) \left[1 - \sqrt{x_1/g(\mathbf{x})}\right]$

deap.benchmarks.**zdt6**(*individual*)
ZDT6 multiobjective function.

$g(\mathbf{x}) = 1 + 9\left[\left(\sum_{i=2}^{n} x_i\right)/(n-1)\right]^{0.25}$

$f_{\text{ZDT61}}(\mathbf{x}) = 1 - \exp(-4x_1)\sin^6(6\pi x_1)$

$f_{\text{ZDT62}}(\mathbf{x}) = g(\mathbf{x})\left[1 - (f_{\text{ZDT61}}(\mathbf{x})/g(\mathbf{x}))^2\right]$

### 3.7.2 Binary Optimization

deap.benchmarks.binary.**chuang_f1**(*individual*)
Binary deceptive function from : Multivariate Multi-Model Approach for Globally Multimodal Problems by Chung-Yao Chuang and Wen-Lian Hsu.

The function takes individual of 40+1 dimensions and has two global optima in [1,1,...,1] and [0,0,...,0].

deap.benchmarks.binary.**chuang_f2**(*individual*)
Binary deceptive function from : Multivariate Multi-Model Approach for Globally Multimodal Problems by Chung-Yao Chuang and Wen-Lian Hsu.

The function takes individual of 40+1 dimensions and has four global optima in [1,1,...,0,0], [0,0,...,1,1], [1,1,...,1] and [0,0,...,0].

deap.benchmarks.binary.**chuang_f3**(*individual*)
Binary deceptive function from : Multivariate Multi-Model Approach for Globally Multimodal Problems by Chung-Yao Chuang and Wen-Lian Hsu.

The function takes individual of 40+1 dimensions and has two global optima in [1,1,...,1] and [0,0,...,0].

deap.benchmarks.binary.**royal_road1**(*individual*, *order*)
Royal Road Function R1 as presented by Melanie Mitchell in : "An introduction to Genetic Algorithms".

deap.benchmarks.binary.**royal_road2**(*individual*, *order*)
Royal Road Function R2 as presented by Melanie Mitchell in : "An introduction to Genetic Algorithms".

deap.benchmarks.binary.**bin2float**(*min_*, *max_*, *nbits*)
Convert a binary array into an array of float where each float is composed of *nbits* and is between *min_* and *max_* and return the result of the decorated function.

---

**Note:** This decorator requires the first argument of the evaluation function to be named *individual*.

---

### 3.7.3 Moving Peaks Benchmark

Re-implementation of the Moving Peaks Benchmark by Jurgen Branke. With the addition of the fluctuating number of peaks presented in *du Plessis and Engelbrecht, 2013, Self-Adaptive Environment with Fluctuating Number of Optima.*

**class** deap.benchmarks.movingpeaks.**MovingPeaks**(*self, dim[, pfunc][, npeaks][, bfunc][, random][, ...]*)
The Moving Peaks Benchmark is a fitness function changing over time. It consists of a number of peaks, changing in height, width and location. The peaks function is given by *pfunc*, wich is either a function object or a list of function objects (the default is `function1()`). The number of peaks is determined by *npeaks* (which defaults to 5). This parameter can be either a integer or a sequence. If it is set to an integer the number of peaks won't change, while if set to a sequence of 3 elements, the number of peaks will fluctuate between the first and third element of that sequence, the second element is the inital number of peaks. When fluctuating the number of peaks, the parameter *number_severity* must be included, it represents the number of peak fraction that is

allowed to change. The dimensionality of the search domain is *dim*. A basis function *bfunc* can also be given to act as static landscape (the default is no basis function). The argument *random* serves to grant an independent random number generator to the moving peaks so that the evolution is not influenced by number drawn by this object (the default uses random functions from the Python module `random`). Various other keyword parameters listed in the table below are required to setup the benchmark, default parameters are based on scenario 1 of this benchmark.

| Parameter | SCENARIO_1 (Default) | SCENARIO_2 | SCENARIO_3 | Details |
|---|---|---|---|---|
| `pfunc` | `function1()` | `cone()` | `cone()` | The peak function or a list of peak function. |
| `npeaks` | 5 | 10 | 50 | Number of peaks. If an integer, the number of peaks won't change, if a sequence it will fluctuate [min, current, max]. |
| `bfunc` | `None` | `None` | `lambda x: 10` | Basis static function. |
| `min_coord` | 0.0 | 0.0 | 0.0 | Minimum coordinate for the centre of the peaks. |
| `max_coord` | 100.0 | 100.0 | 100.0 | Maximum coordinate for the centre of the peaks. |
| `min_height` | 30.0 | 30.0 | 30.0 | Minimum height of the peaks. |
| `max_height` | 70.0 | 70.0 | 70.0 | Maximum height of the peaks. |
| `uniform_height` | 50.0 | 50.0 | 0 | Starting height for all peaks, if `uniform_height <= 0` the initial height is set randomly for each peak. |
| `min_width` | 0.0001 | 1.0 | 1.0 | Minimum width of the peaks. |
| `max_width` | 0.2 | 12.0 | 12.0 | Maximum width of the peaks |
| `uniform_width` | 0.1 | 0 | 0 | Starting width for all peaks, if `uniform_width <= 0` the initial width is set randomly for each peak. |
| `lambda_` | 0.0 | 0.5 | 0.5 | Correlation between changes. |
| `move_severity` | 1.0 | 1.5 | 1.0 | The distance a single peak moves when peaks change. |
| `height_severity` | 7.0 | 7.0 | 1.0 | The standard deviation of the change made to the height of a peak when peaks change. |
| `width_severity` | 0.01 | 1.0 | 0.5 | The standard deviation of the change made to the width of a peak when peaks change. |
| `period` | 5000 | 5000 | 1000 | Period between two changes. |

Dictionnaries `SCENARIO_1`, `SCENARIO_2` and `SCENARIO_3` of this module define the defaults for these parameters. The scenario 3 requires a constant basis function which can be given as a lambda function `lambda x: constant`.

The following shows an example of scenario 1 with non uniform heights and widths.

**changePeaks**()
Order the peaks to change position, height, width and number.

**globalMaximum**()
Returns the global maximum value and position.

**maximums**()
Returns all visible maximums value and position sorted with the global maximum first.

deap.benchmarks.movingpeaks.**cone**(*individual*, *position*, *height*, *width*)
The cone peak function to be used with scenario 2 and 3.

$$f(\mathbf{x}) = h - w\sqrt{\sum_{i=1}^{N}(x_i - p_i)^2}$$

deap.benchmarks.movingpeaks.**function1**(*individual*, *position*, *height*, *width*)
The function1 peak function to be used with scenario 1.

$$f(\mathbf{x}) = \frac{h}{1 + w\sqrt{\sum_{i=1}^{N}(x_i - p_i)^2}}$$

### 3.7.4 Benchmarks tools

Module containing tools that are useful when benchmarking algorithms

deap.benchmarks.tools.**convergence**(*first_front*, *optimal_front*)
Given a Pareto front *first_front* and the optimal Pareto front, this function returns a metric of convergence of the front as explained in the original NSGA-II article by K. Deb. The smaller the value is, the closer the front is to the optimal one.

deap.benchmarks.tools.**diversity**(*first_front*, *first*, *last*)
Given a Pareto front *first_front* and the two extreme points of the optimal Pareto front, this function returns a metric of the diversity of the front as explained in the original NSGA-II article by K. Deb. The smaller the value is, the better the front is.

deap.benchmarks.tools.**noise**(*noise*)
Decorator for evaluation functions, it evaluates the objective function and adds noise by calling the function(s) provided in the *noise* argument. The noise functions are called without any argument, consider using the `Toolbox` or Python's `functools.partial()` to provide any required argument. If a single function is provided it is applied to all objectives of the evaluation function. If a list of noise functions is provided, it must be of length equal to the number of objectives. The noise argument also accept `None`, which will leave the objective without noise.

This decorator adds a `noise()` method to the decorated function.

noise.**noise**(*noise*)
Set the current noise to *noise*. After decorating the evaluation function, this function will be available directly from the function object.

```
prand = functools.partial(random.gauss, mu=0.0, sigma=1.0)

@noise(prand)
def evaluate(individual):
    return sum(individual),

# This will remove noise from the evaluation function
evaluate.noise(None)
```

deap.benchmarks.tools.**rotate**(*matrix*)
Decorator for evaluation functions, it rotates the objective function by *matrix* which should be a valid orthogonal NxN rotation matrix, with N the length of an individual. When called the decorated function should take as first argument the individual to be evaluated. The inverse rotation matrix is actually applied to the individual and the resulting list is given to the evaluation function. Thus, the evaluation function shall not be expecting an individual as it will receive a plain list (numpy.array). The multiplication is done using numpy.

This decorator adds a `rotate()` method to the decorated function.

---

**Note:** A random orthogonal matrix Q can be created via QR decomposition.

```
A = numpy.random.random((n,n))
Q, _ = numpy.linalg.qr(A)
```

---

rotate.**rotate**(*matrix*)
Set the current rotation to *matrix*. After decorating the evaluation function, this function will be available directly from the function object.

```
# Create a random orthogonal matrix
A = numpy.random.random((n,n))
Q, _ = numpy.linalg.qr(A)
```

```
@rotate(Q)
def evaluate(individual):
    return sum(individual),

# This will reset rotation to identity
evaluate.rotate(numpy.identity(n))
```

deap.benchmarks.tools.**scale**(*factor*)

Decorator for evaluation functions, it scales the objective function by *factor* which should be the same length as the individual size. When called the decorated function should take as first argument the individual to be evaluated. The inverse factor vector is actually applied to the individual and the resulting list is given to the evaluation function. Thus, the evaluation function shall not be expecting an individual as it will receive a plain list.

This decorator adds a `scale()` method to the decorated function.

scale.**scale**(*factor*)

Set the current scale to *factor*. After decorating the evaluation function, this function will be available directly from the function object.

```
@scale([0.25, 2.0, ..., 0.1])
def evaluate(individual):
    return sum(individual),

# This will cancel the scaling
evaluate.scale([1.0, 1.0, ..., 1.0])
```

deap.benchmarks.tools.**translate**(*vector*)

Decorator for evaluation functions, it translates the objective function by *vector* which should be the same length as the individual size. When called the decorated function should take as first argument the individual to be evaluated. The inverse translation vector is actually applied to the individual and the resulting list is given to the evaluation function. Thus, the evaluation function shall not be expecting an individual as it will receive a plain list.

This decorator adds a `translate()` method to the decorated function.

translate.**translate**(*vector*)

Set the current translation to *vector*. After decorating the evaluation function, this function will be available directly from the function object.

```
@translate([0.25, 0.5, ..., 0.1])
def evaluate(individual):
    return sum(individual),

# This will cancel the translation
evaluate.translate([0.0, 0.0, ..., 0.0])
```

CHAPTER

FOUR

# WHAT'S NEW?

Here is a (incomplete) log of the changes made to DEAP over time.

## 4.1 Release 0.9

- Major overhaul of the genetic programming with significant speed increase.
- Added state of the art operators to control bloat in GP.
- Several new examples from diverse fields.
- Examples are now compatible Python 2 and 3 out of the box.
- Organization of the examples.

## 4.2 Release 0.8

- Added forward compatibility to Python 3.2
- Replaced `varSimple()` and `varLambda()` variation operators for the more specific `varAnd()` and `varOr()` operators.
- Added a logging facility (`EvolutionLogger`) that produce easier to read console logging and a utility to transform that output into a Python dictionary.
- Introduced the exact NSGA-II algorithm as described in *Deb et al., 2002, A Fast Elitist Multiobjective Genetic Algorithm: NSGA-II*.
- NSGA-II selection algorithm revisited :
  - Added a C++ version;
  - Speed up of the Python version (up to 5x when the objectives are discrete).
- Added some new benchmarks (multiobjective, binary and moving peaks).
- Added translation, rotation, scaling and noise decorators to enhance benchmarks.

## 4.3 Release 0.7

- Modified structure so that DTM is a module of DEAP.

- Restructured modules in a more permanent and coherent way.
  - The toolbox is now in the module base.
  - The operators have been moved to the tools module.
  - Checkpoint, Statistics, History and Hall-of-Fame are now also in the tools module.
  - Moved the GP specific operators to the gp module.
- Renamed some operator for coherence.
- Reintroduced a convenient, coherent and simple Statistics module.
- Changed the Milestone module name for the more common Checkpoint name.
- Eliminated the confusing *content_init* and *size_init* keywords in the toolbox.
- Refactored the whole documentation in a more structured manner.
- Added a benchmark module containing some of the most classic benchmark functions.
- Added a lot of examples again :
  - Differential evolution (*x2*);
  - Evolution strategy : One fifth rule;
  - *k*-nearest neighbours feature selection;
  - One Max Multipopulation;
  - Particle Swarm Optimization;
  - Hillis' coevolution of sorting networks;
  - CMA-ES $1 + \lambda$.

## 4.4 Release 0.6

- Operator modify in-place the individuals (simplify a lot the algorithms).
- Toolbox now contains two basic methods, map and clone that are useful in the algorithms.
- The two methods can be replaced (as usual) to modify the behaviour of the algorithms.
- Added new module History (compatible with NetworkX).
- Genetic programming is now possible with Automatically Defined Functions (ADFs).
- Algorithms now refers to literature algorithms.
- Added new examples :
  - Coevolution;
  - Variable length genotype;
  - Multiobjective;
  - Inheriting from a Set;
  - Using ADFs;
  - Multiprocessing.
- Basic operators can now be enhanced with decorators to do all sort of funny stuff.

## 4.5 Release 0.5

- Added a new module Milestone.
- Enhanced Fitness efficiency when comparing fitnesses.
- Replaced old base types with python built-in types.
- Added an example of deriving from sets.
- Added SPEA-II algorithm.
- Fitnesses are no more extended when assigning value, the values are simply assigned.

CHAPTER

FIVE

# REPORTING A BUG

You can report a bug on the issue list on google code.

http://code.google.com/p/deap/issues/list

For simple questions, contact us on the deap users group.

http://groups.google.com/group/deap-users

CHAPTER

SIX

# CONTRIBUTING

## 6.1 Reporting a bug

You can report a bug on the issue list on google code.

http://code.google.com/p/deap/issues/list

## 6.2 Retrieving the latest code

You can check the latest sources with the command:

```
hg clone https://deap.googlecode.com/hg
```

If you want to use the development version, you have to update the repository to the developing branch with the command:

```
hg update dev
```

## 6.3 Contributing code

Contact us on the deap users list at http://groups.google.com/group/deap-users.

## 6.4 Coding guidelines

Most of those conventions are base on Python PEP8.

> *A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.*

### 6.4.1 Code layout

Same as PEP8.

### 6.4.2 Imports

First imports in a file are the standard library module, then come the imports of eap module, and finally the custom module for a problem. Each block of imports should be separated by a new line.

```
import system

from deap import base

import mymodule
```

### 6.4.3 Whitespace in Expressions and Statements

Same as PEP8.

### 6.4.4 Comments

Same as PEP8

### 6.4.5 Documentation Strings

Same as PEP8

### 6.4.6 Naming Conventions

- **Module** : use the lowercase convention.
- **Class** : class names use the CapWords? convention.
- **Function /** Procedure : use the mixedCase convention. First word should be an action verb.
- **Variable** : use the lower_case_with_underscores convention.

# BIBLIOGRAPHY


[Potter2001] Potter, M. and De Jong, K., 2001, Cooperative Coevolution: An Architecture for Evolving Co-adapted Subcomponents.

[Beyer2002] Beyer and Schwefel, 2002, Evolution strategies - A Comprehensive Introduction

[Hansen2001] Hansen and Ostermeier, 2001. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation*

[Hansen2001] Hansen and Ostermeier, 2001. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation*

[Hansen2009] Hansen, 2009. Benchmarking a BI-Population CMA-ES on the BBOB-2009 Function Testbed. *GECCO'09*

[Poli2007] Ricardo Poli, James Kennedy and Tim Blackwell, "Particle swarm optimization an overview". Swarm Intelligence. 2007; 1: 33–57

[Knoek2003] Arthur J. Knoek van Soest and L. J. R. Richard Casius, "The merits of a parallel genetic algorithm in solving hard optimization problems". Journal of Biomechanical Engineering. 2003; 125: 141–146

[Goldberg1985] Goldberg and Lingel, "Alleles, loci, and the traveling salesman problem", 1985.

[Cicirello2000] Cicirello and Smith, "Modeling GA performance for control parameter optimization", 2000.

[Goldberg1989] Goldberg. Genetic algorithms in search, optimization and machine learning. Addison Wesley, 1989

[Beyer2002] Beyer and Schwefel, 2002, Evolution strategies - A Comprehensive Introduction

[Schwefel1995] Schwefel, 1995, Evolution and Optimum Seeking. Wiley, New York, NY

[Deb2002] Deb, Pratab, Agarwal, and Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II", 2002.

[Zitzler2001] Zitzler, Laumanns and Thiele, "SPEA 2: Improving the strength Pareto evolutionary algorithm", 2001.

[Luke2002fighting] Luke and Panait, 2002, Fighting bloat with nonparametric parsimony pressure

[Koza1989] J.R. Koza, Genetic Programming - On the Programming of Computers by Means of Natural Selection (MIT Press, Cambridge, MA, 1992)

[Back2000] Back, Fogel and Michalewicz, "Evolutionary Computation 1 : Basic Algorithms and Operators", 2000.

[Colette2010] Collette, Y., N. Hansen, G. Pujol, D. Salazar Aponte and R. Le Riche (2010). On Object-Oriented Programming of Optimizers - Examples in Scilab. In P. Breitkopf and R. F. Coelho, eds.: Multidisciplinary Design Optimization in Computational Mechanics, Wiley, pp. 527-565;

# PYTHON MODULE INDEX

## d