Table of Contents

Operators and Algorithms

Before starting with complex algorithms, we will see some basics 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.

A First Individual

First import the required modules and register the different functions required to create individuals that are lists 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.

Evaluation

The evaluation is the most personal part of an evolutionary algorithm, it is the only part of the library that you must write yourself. A typical evaluation function takes one individual as argument and returns its fitness as a `tuple`. As shown in the in the core 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 assigns 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.

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 types of individuals. 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 another 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're not related to the individual anymore. As stated above, the mutation does mutate and only mutate an individual it is neither 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
```

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 types 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 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.

Selection

Selection is made among a population by the selection operators that are available in the `deap.tools` 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]
```

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 algorithm. 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.cxTwoPoint)
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 pseudo code. Now it is up to you to write and experiment on your own.

Tool Decoration

Tool decoration is a very powerful feature that helps to control very precise things 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 this 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 checked 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.

Variations

Variations allow 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. This variation can be used to simplify the writing of an algorithm as follows.

```
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 pseudo code.

Algorithms

There are 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 the 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, is 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.