Overview

If you are used to any 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 ones 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.

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 seem 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.

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 their 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.

v: master

Operators

Operators are just like initializers, except that some are already implemented in the `tools` module. Once you've chosen 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.cxTwoPoint)
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, allowing generic algorithms that do not depend on operator names. Note also that fitness values must be iterable, that is why we return a tuple in the evaluate function. More on this in the Operators and Algorithms tutorial and Examples.

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.