Creating Types

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

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 `create()` function takes at least two arguments, a name for the newly created class and a base class. Any subsequent argument becomes an attribute of the class. As specified in the `fitness` documentation, the `weights` attribute 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))
```

This code produces a fitness that minimizes the first objective and maximize the second one. The weights can also be used to vary 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.

Individual

Simply by thinking about the different flavors 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`.

Warning: Before inherinting from `numpy.ndarray` you should **absolutely** read the Inheriting from *ndarray* tutorial and have a look at the One Max Problem: Using Numpy example!

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` type and have a `fitness` attribute.

```
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)
```

The newly introduced `register()` method takes at least two arguments: an alias and a function assigned to this alias. Any subsequent argument is passed to the function when called (à la `functools.partial()`). Thus, the preceding code creates two aliases in the toolbox; `attr_float` and `individual`. The first one redirects to the `random.random()` function. The second one is a shortcut to the `initRepeat()` function, fixing its `container` argument to the `creator.Individual` class, its `func` argument to the `toolbox.attr_float()` function, and its number of repetitions argument to `IND_SIZE`.

Now, calling `toolbox.individual()` will call `initRepeat()` with the fixed arguments and return a complete `creator.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)
```

Type inheriting from arrays 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.

```
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)
```

The first registered function `indices` refers to the `random.sample()` function with its arguments fixed to sample `IND_SIZE` numbers from the given range. The second registered function `individual` is a shortcut to the `initIterate()` function, with its `container` argument set to the `creator.Individual` class and its `generator` argument to the `toolbox.indices()` alias.

Calling `toolbox.individual()` will call `initIterate()` with the fixed arguments and return a complete `creator.Individual` composed of a permutation 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 `primitive_set` must be defined containing all possible mathematical operators that our individual can use. Here, the set is called `pset` 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 with the addition of a static attribute `pset` to remember the global primitive set. This time, the content of the individuals will be generated by the `genHalfAndHalf()` function that generates trees in a list format based on a ramped procedure. Once again, the individual is initialized 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.genHalfAndHalf, 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 lists, 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 ourselves. The `uniform()` function receives two classes and instantiates them generating itself the random numbers in the ranges 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 as 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 has 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.

Population

Populations are much like individuals. Instead of being initialized with attributes, they are 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 the toolbox and the `initRepeat()` function directly.

```
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 the grid where each cell contains a single individual. However, its implementation only differs from the list of the bag population, in 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[i][j]`. For the moment, there is no algorithm specialized for structured populations, we are awaiting your submissions.

Swarm

A swarm is used in particle swarm optimization. It is different in the sense that it contains a communication network. The simplest network is the completely connected one, where each particle knows the best position that have ever been visited by any 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 to an island in the island model. Demes, being only sub-populations, are in fact not different from populations, aside from 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):
    with open(filename, "r") as pop_file:
        contents = json.load(pop_file)
    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 `individual()` 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")
```