Table of Contents

Previous topic

Benchmarking Against the Bests (BBOB)

Next topic

Examples

This Page

Show Source

Quick search

Go

Inheriting from Numpy

DEAP's `creator` allows to inherit from `numpy.ndarray` so that individuals can have the properties of the powerful Numpy library. As with any other base class, inheriting from a `numpy.ndarray` is no more complicated than putting it as a base class.

```
import numpy
from deap import base, creator
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", numpy.ndarray, fitness=creator.FitnessMax)
```

What You Should be Concerned With!

Inheriting from `numpy.ndarray` is an appealing feature, but some care must be taken regarding validity of the data and performance of the system.

Copy and Slicing

Slicing a `numpy.ndarray` should be done with care. The returned element is a `numpy.ndarray.view()` of the original object. This leads to bug prone code when swapping data from one array to another. For example, the two points crossover use the following for swapping data between two lists.

```
>>> a = [1,2,3,4]
>>> b = [5,6,7,8]
>>> a[1:3], b[1:3] = b[1:3], a[1:3]
>>> print(a)
[1, 6, 7, 4]
>>> print(b)
[5, 2, 3, 8]
```

With `numpy.array`, the same operation leads to a single resulting individual being changed.

```
>>> import numpy
>>> a = numpy.array([1,2,3,4])
>>> b = numpy.array([5,6,7,8])
>>> a[1:3], b[1:3] = b[1:3], a[1:3]
>>> print(a)
[1 6 7 4]
>>> print(b)
[5 6 7 8]
```

The problem is that, first, the elements in `a` are replaced by the elements of the view returned by `b` and the element of `b` are replaced by the element in the view of `a` which are now the one intially in `b` leading to the wrong final result. One way of to circumvent this problem is to explicitely copy the view returned by the `__getitem__`.

```
>>> import numpy
>>> a = numpy.array([1,2,3,4])
>>> b = numpy.array([5,6,7,8])
>>> a[1:3], b[1:3] = b[1:3].copy(), a[1:3].copy()
>>> print(a)
[1 6 7 4]
>>> print(b)
[5 2 3 8]
```

Thus, care must be taken when inheriting from `numpy.ndarray`; **none** of the operators in the `tools` module implement such copying. See the One Max with Numpy example for the complete two points crossover.

Comparing Individuals

When one wants to use a `HallOfFame` or `ParetoFront` hall-of-fame. The *similar* function should be changed to a compare all function. Using the regular `operator.eq()` function will result in a vector of comparisons

```
>>> a = numpy.array([1, 2, 3])
>>> b = numpy.array([1, 2, 3])
>>> operator.eq(a, b)
array([ True,  True,  True], dtype=bool)
```

This cannot be used as a condition

```
>>> if operator.eq(a, b):
...     print "Gosh!"
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: The truth value of an array with more than one element is ambiguous. Use a.any()
```

One must replace the *similar* operator by a numpy function like `numpy.array_equal()` or `numpy.allclose()`.

```
hof = tools.HallOfFame(1, similar=numpy.array_equal)
```

Now the condition can be computed and the hall-of-fame will be happy.

```
>>> if numpy.array_equal(a, b):
...     print "Yeah!"
"Yeah!"
```

Performance

If your intent is performance, DEAP Speed reveals that using an `array.array` should be prefered to `numpy.ndarray`. This is mostly because the creation (also required by the deep copy) of new arrays is longer for the `numpy.array` than for `array.array`.

What You Don't Need to Know

The creator replaces systematically several functions of the basic `numpy.ndarray` so that

- array instances can be created from an iterable;
- it deep copies the attributes added in the `__dict__` of the object;
- pickling includes the dictionary of attributes.

See the implementation of `_numpy_array` in the `creator` module for more details.

v: master

Built on Jun 05, 2019. Found a bug?
Created using Sphinx 1.8.5.