Peach - Computational Intelligence for Python

API Documentation

December 21, 2009

Contents

1 Package peach

Peach is a pure-Python package with aims to implement techniques of machine learning and computational intelligence. It contains packages for

- Neural Networks, including, but not limited to, multi-layer perceptrons and self-organizing maps;
- Fuzzy logic and fuzzy inference systems, including Mamdani-type and Sugeno-type controllers;
- Optimization packages, including multidimensional optimization;
- Stochastic Optimizations, including genetic algorithms, simulated annealing, particle swarm optimization;
- A lot more.

Author: José Alexandre Nalon

Version: 0.1.0

1.1 Modules

  - **lrules**: Learning rules for neural networks and base classes for custom learning.
    (Section 18, p. 135)
  - **nn**: Basic topologies of neural networks.
    (Section 19, p. 153)
- **optm**: This package implements deterministic optimization methods.
  (Section 20, p. 165)
  - **linear**: This package implements basic one variable only optimizers.
    (Section 21, p. 166)
  - **multivar**: This package implements basic multivariable optimizers, including gradient and Newton searches.
    (Section 22, p. 175)
  - **optm**: Basic definitons and base class for optimizers
    (Section 23, p. 182)
  - **quasinewton**: This package implements basic quasi-Newton optimizers.
    (Section 24, p. 186)
  - **sa**: This package implements two versions of simulated annealing optimization.
    (Section 25, p. 194)
  - **stochastic**: General methods of stochastic optimization.
    (Section 26, p. 201)

1.2 Variables

Name	Description
__doc__	**Value:** ...

2 Package peach.fuzzy

This package implements fuzzy logic. Consult:

fuzzy Basic definitions, classes and operations in fuzzy logic;

mf Membership functions;

defuzzy Defuzzification methods;

control Fuzzy controllers (FIS - Fuzzy Inference Systems), for Mamdani- and Sugeno-type controllers and others;

2.1 Modules

- **control**: This package implements fuzzy controllers, of fuzzy inference systems. *(Section 3, p. 5)*
- **defuzzy**: This package implements defuzzification methods for use with fuzzy controllers. *(Section 4, p. 19)*
- **fuzzy**: This package implements basic definitions for fuzzy logic *(Section 5, p. 21)*
- **mf**: Membership functions *(Section 6, p. 47)*
- **norms**: This package implements operations of fuzzy logic. *(Section 7, p. 65)*

2.2 Variables

Name	Description
__doc__	**Value:** ...

3 Module peach.fuzzy.control

This package implements fuzzy controllers, of fuzzy inference systems.

There are two types of controllers implemented in this package. The Mamdani controller is the traditional approach, where input (or controlled) variables are fuzzified, a set of decision rules determine the outcome in a fuzzified way, and a defuzzification method is applied to obtain the numerical result.

The Sugeno controller operates in a similar way, but there is no defuzzification step. Instead, the value of the output (or manipulated) variable is determined by parametric models, and the final result is determined by a weighted average based on the decision rules. This type of controller is also known as parametric controller.

3.1 Variables

Name	Description
__doc__	**Value:** `...`
MAMDANI_INFERENCE	**Value:** `MamdaniImplication, MamdaniAglutination`
PROB_INFERENCE	**Value:** `ProbabilisticImplication, ProbabilisticAglutination`
PROB_NORMS	**Value:** `ProbabilisticAnd, ProbabilisticOr, ProbabilisticNot`
ZADEH_NORMS	**Value:** `ZadehAnd, ZadehOr, ZadehNot`
cos	**Value:** `<ufunc 'cos'>`
exp	**Value:** `<ufunc 'exp'>`
pi	**Value:** `3.14159265359`

3.2 Class Controller

object
└ **peach.fuzzy.control.Controller**

Known Subclasses: peach.fuzzy.control.Mamdani

Basic Mamdani controller

This class implements a standard Mamdani controller. A controller based on fuzzy logic has a somewhat complex behaviour, so it is not explained here. There are numerous references that can be consulted.

It is essential to understand the format that decision rules must follow to obtain correct behaviour of the controller. A rule is a tuple given by:

```
((mx0, mx1, ..., mxn), my)
```

where `mx0` is a membership function of the first input variable, `mx1` is a membership function of the second input variable and so on; and `my` is a membership function or a fuzzy set of the output variable.

Notice that `mx`'s are *functions* not fuzzy sets! They will be applied to the values of the input variables given in the function call, so, if they are anything different from a membership function, an exception will be raised. Please, consult the examples to see how they must be used.

3.2.1 Methods

__init__(*self*, *yrange*, *rules*=[], *defuzzy*=<function Centroid at 0x885ecdc>, *norm*=<function ZadehAnd at 0x885eaac>, *conorm*=<function ZadehOr at 0x885eae4>, *negation*=<function ZadehNot at 0x885eb1c>, *imply*=<function MamdaniImplication at 0x885eb54>, *aglutinate*=<function MamdaniAglutination at 0x885eb8c>)

Creates and initialize the controller.

Parameters

`yrange:` The range of the output variable. This must be given as a set of points belonging to the interval where the output variable is defined, not only the start and end points. It is strongly suggested that the interval is divided in some (eg.: 100) points equally spaced;

`rules:` The set of decision rules, as defined above. If none is given, an empty set of rules is assumed;

`defuzzy:` The defuzzification method to be used. If none is given, the Centroid method is used;

`norm:` The norm (`and` operation) to be used. Defaults to Zadeh and.

`conorm:` The conorm (`or` operation) to be used. Defaults to Zadeh or.

`negation:` The negation (`not` operation) to be used. Defaults to Zadeh not.

`imply:` The implication method to be used. Defaults to Mamdani implication.

`aglutinate:` The aglutination method to be used. Defaults to Mamdani aglutination.

Overrides: object.__init__

__gety(*self*)

__getrules(*self*)

set_norm(*self*, *f*)

Sets the norm (`and`) to be used.
This method must be used to change the behavior of the `and` operation of the controller.

Parameters

`f:` The function can be any function that takes two numerical values and return one numerical value, that corresponds to the `and` result.

set_conorm(*self*, *f*)

Sets the conorm (`or`) to be used.
This method must be used to change the behavior of the `or` operation of the controller.

Parameters

`f:` The function can be any function that takes two numerical values and return one numerical value, that corresponds to the `or` result.

set_negation(*self*, *f*)

Sets the negation (`not`) to be used.
This method must be used to change the behavior of the `not` operation of the controller.

Parameters

`f:` The function can be any function that takes one numerical value and return one numerical value, that corresponds to the `not` result.

set_implication(*self*, *f*)

Sets the implication to be used.
This method must be used to change the behavior of the implication operation of the controller.

Parameters

`f:` The function can be any function that takes two numerical values and return one numerical value, that corresponds to the implication result.

set_aglutination(*self*, *f*)

Sets the aglutination to be used.
This method must be used to change the behavior of the aglutination operation of the controller.

Parameters

`f:` The function can be any function that takes two numerical values and return one numerical value, that corresponds to the aglutination result.

add_rule(*self*, *rule*)

Adds a decision rule to the knowledge base.
It is essential to understand the format that decision rules must follow to obtain correct behaviour of the controller. A rule is a tuple must have the following format:

```
((mx0, mx1, ..., mxn), my)
```

where `mx0` is a membership function of the first input variable, `mx1` is a membership function of the second input variable and so on; and `my` is a membership function or a fuzzy set of the output variable.
Notice that `mx`'s are *functions* not fuzzy sets! They will be applied to the values of the input variables given in the function call, so, if they are anything different from a membership function, an exception will be raised when the controller is used. Please, consult the examples to see how they must be used.

add_table(*self*, *lx1*, *lx2*, *table*)

Adds a table of decision rules in a two variable controller.
Typically, fuzzy controllers are used to control two variables. In that case, the set of decision rules are given in the form of a table, since that is a more compact format and very easy to visualize. This is a convenience function that allows to add decision rules in the form of a table. Notice that the resulting knowledge base will be the same if this function is used or the `add_rule` method is used with every single rule. The second method is in general easier to read in a script, so consider well.

Parameters

`lx1:` The set of membership functions to the variable `x1`, or the lines of the table
`lx2:` The set of membership functions to the variable `x2`, or the columns of the table
`table:` The consequent of the rule where the condition is the line `and` the column. These can be the membership functions or fuzzy sets.

eval(*self*, *r*, *xs*)

Evaluates one decision rule in this controller
Takes a rule from the controller and evaluates it given the values of the input variables.

Parameters

`r`: The rule in the standard format, or an integer number. If `r` is an integer, then the `r` th rule in the knowledge base will be evaluated.

`xs`: A tuple, a list or an array containing the values of the input variables. The dimension must be coherent with the given rule.

Return Value

This method evaluates each membership function in the rule for each given value, and `and` 's the results to obtain the condition. If the condition is zero, a tuple `(0.0, None)` `is returned. Otherwise, the condition is ‘‘imply` ed in the membership function of the output variable. A tuple containing `(condition, imply)` (the membership value associated to the condition and the result of the implication) is returned.

eval_all(*self*, **xs*)

Evaluates all the rules and aglutinates the results.
Given the values of the input variables, evaluate and apply every rule in the knowledge base (with the `eval` method) and aglutinates the results.

Parameters

`xs`: A tuple, a list or an array with the values of the input variables.

Return Value

A fuzzy set containing the result of the evaluation of every rule in the knowledge base, with the results aglutinated.

__call__(*self*, **xs*)

Apply the controller to the set of input variables
Given the values of the input variables, evaluates every decision rule, aglutinates the results and defuzzify it. Returns the response of the controller.

Parameters

`xs`: A tuple, a list or an array with the values of the input variables.

Return Value

The response of the controller.

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)
Return Value
`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

3.2.2 Properties

Name	Description
y	Property that returns the output variable interval. Not writable **Value:** `<property object at 0x887be3c>`
rules	Property that returns the list of decision rules. Not writable **Value:** `<property object at 0x887be64>`
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

3.3 Class Mamdani

object
peach.fuzzy.control.Controller
peach.fuzzy.control.Mamdani

`Mandani` is an alias to `Controller`

3.3.1 Methods

__call__(*self*, **xs*)

Apply the controller to the set of input variables
Given the values of the input variables, evaluates every decision rule, aglutinates the results and defuzzify it. Returns the response of the controller.

Parameters

`xs`: A tuple, a list or an array with the values of the input variables.

Return Value

The response of the controller.

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__init__(*self*, *yrange*, *rules*=`[]`, *defuzzy*=`<function Centroid at 0x885ecdc>`, *norm*=`<function ZadehAnd at 0x885eaac>`, *conorm*=`<function ZadehOr at 0x885eae4>`, *negation*=`<function ZadehNot at 0x885eb1c>`, *imply*=`<function MamdaniImplication at 0x885eb54>`, *aglutinate*=`<function MamdaniAglutination at 0x885eb8c>`)

Creates and initialize the controller.

Parameters

`yrange`:	The range of the output variable. This must be given as a set of points belonging to the interval where the output variable is defined, not only the start and end points. It is strongly suggested that the interval is divided in some (eg.: 100) points equally spaced;
`rules`:	The set of decision rules, as defined above. If none is given, an empty set of rules is assumed;
`defuzzy`:	The defuzzification method to be used. If none is given, the Centroid method is used;
`norm`:	The norm (`and` operation) to be used. Defaults to Zadeh and.
`conorm`:	The conorm (`or` operation) to be used. Defaults to Zadeh or.
`negation`:	The negation (`not` operation) to be used. Defaults to Zadeh not.
`imply`:	The implication method to be used. Defaults to Mamdani implication.
`aglutinate`:	The aglutination method to be used. Defaults to Mamdani aglutination.

Overrides: object.__init__

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

add_rule(*self*, *rule*)

Adds a decision rule to the knowledge base.
It is essential to understand the format that decision rules must follow to obtain correct behaviour of the controller. A rule is a tuple must have the following format:

```
((mx0, mx1, ..., mxn), my)
```

where `mx0` is a membership function of the first input variable, `mx1` is a membership function of the second input variable and so on; and `my` is a membership function or a fuzzy set of the output variable.
Notice that `mx`'s are *functions* not fuzzy sets! They will be applied to the values of the input variables given in the function call, so, if they are anything different from a membership function, an exception will be raised when the controller is used. Please, consult the examples to see how they must be used.

add_table(*self*, *lx1*, *lx2*, *table*)

Adds a table of decision rules in a two variable controller.
Typically, fuzzy controllers are used to control two variables. In that case, the set of decision rules are given in the form of a table, since that is a more compact format and very easy to visualize. This is a convenience function that allows to add decision rules in the form of a table. Notice that the resulting knowledge base will be the same if this function is used or the `add_rule` method is used with every single rule. The second method is in general easier to read in a script, so consider well.

Parameters

`lx1:` The set of membership functions to the variable `x1`, or the lines of the table
`lx2:` The set of membership functions to the variable `x2`, or the columns of the table
`table:` The consequent of the rule where the condition is the line `and` the column. These can be the membership functions or fuzzy sets.

eval(*self*, *r*, *xs*)

Evaluates one decision rule in this controller
Takes a rule from the controller and evaluates it given the values of the input variables.

Parameters

`r`: The rule in the standard format, or an integer number. If `r` is an integer, then the `r` th rule in the knowledge base will be evaluated.

`xs`: A tuple, a list or an array containing the values of the input variables. The dimension must be coherent with the given rule.

Return Value

This method evaluates each membership function in the rule for each given value, and `and` 's the results to obtain the condition. If the condition is zero, a tuple `(0.0, None)` `is returned. Otherwise, the condition is ‘‘imply` ed in the membership function of the output variable. A tuple containing `(condition, imply)` (the membership value associated to the condition and the result of the implication) is returned.

eval_all(*self*, **xs*)

Evaluates all the rules and aglutinates the results.
Given the values of the input variables, evaluate and apply every rule in the knowledge base (with the `eval` method) and aglutinates the results.

Parameters

`xs`: A tuple, a list or an array with the values of the input variables.

Return Value

A fuzzy set containing the result of the evaluation of every rule in the knowledge base, with the results aglutinated.

set_aglutination(*self*, *f*)

Sets the aglutination to be used.
This method must be used to change the behavior of the aglutination operation of the controller.

Parameters

`f`: The function can be any function that takes two numerical values and return one numerical value, that corresponds to the aglutination result.

set_conorm(*self*, *f*)

Sets the conorm (`or`) to be used.
This method must be used to change the behavior of the `or` operation of the controller.

Parameters

`f`: The function can be any function that takes two numerical values and return one numerical value, that corresponds to the `or` result.

set_implication(*self*, *f*)

Sets the implication to be used.
This method must be used to change the behavior of the implication operation of the controller.

Parameters

`f:` The function can be any function that takes two numerical values and return one numerical value, that corresponds to the implication result.

set_negation(*self*, *f*)

Sets the negation (`not`) to be used.
This method must be used to change the behavior of the `not` operation of the controller.

Parameters

`f:` The function can be any function that takes one numerical value and return one numerical value, that corresponds to the `not` result.

set_norm(*self*, *f*)

Sets the norm (`and`) to be used.
This method must be used to change the behavior of the `and` operation of the controller.

Parameters

`f:` The function can be any function that takes two numerical values and return one numerical value, that corresponds to the `and` result.

3.3.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`
rules	Property that returns the list of decision rules. Not writable **Value:** `<property object at 0x887be64>`
y	Property that returns the output variable interval. Not writable **Value:** `<property object at 0x887be3c>`

3.4 Class Parametric

object
└ **peach.fuzzy.control.Parametric**

Known Subclasses: peach.fuzzy.control.Sugeno

Basic Parametric controller

This class implements a standard parametric (or Takagi-Sugeno) controller. A controller based on fuzzy logic has a somewhat complex behaviour, so it is not explained here. There are numerous references that can be consulted.

It is essential to understand the format that decision rules must follow to obtain correct behaviour of the controller. A rule is a tuple given by:

```
((mx0, mx1, ..., mxn), (a0, a1, ..., an))
```

where `mx0` is a membership function of the first input variable, `mx1` is a membership function of the second input variable and so on; and `a0` is the linear parameter, `a1` is the parameter associated with the first input variable, `a2` is the parameter associated with the second input variable and so on. The response to the rule is calculated by:

```
y = a0 + a1*x1 + a2*x2 + ... + an*xn
```

Notice that `mx`'s are *functions* not fuzzy sets! They will be applied to the values of the input variables given in the function call, so, if they are anything different from a membership function, an exception will be raised. Please, consult the examples to see how they must be used.

3.4.1 Methods

__init__(*self*, *rules*=[], *norm*=<function ProbabilisticAnd at 0x885ebc4>, *conorm*=<function ProbabilisticOr at 0x885ebfc>, *negation*=<function ProbabilisticNot at 0x885ec34>)

Creates and initializes the controller.

Parameters

`rules:`	List containing the decision rules for the controller. If not given, an empty set of decision rules is used.
`norm:`	The norm (`and` operation) to be used. Defaults to Probabilistic and.
`conorm:`	The conorm (`or` operation) to be used. Defaults to Probabilistic or.
`negation:`	The negation (`not` operation) to be used. Defaults to Probabilistic not.

Overrides: object.__init__

__getrules(*self*)

add_rule(*self*, *rule*)

Adds a decision rule to the knowledge base.
It is essential to understand the format that decision rules must follow to obtain correct behaviour of the controller. A rule is a tuple given by:

```
((mx0, mx1, ..., mxn), (a0, a1, ..., an))
```

where `mx0` is a membership function of the first input variable, `mx1` is a membership function of the second input variable and so on; and `a0` is the linear parameter, `a1` is the parameter associated with the first input variable, `a2` is the parameter associated with the second input variable and so on. Notice that `mx`'s are *functions* not fuzzy sets! They will be applied to the values of the input variables given in the function call, so, if they are anything different from a membership function, an exception will be raised. Please, consult the examples to see how they must be used.

eval(*self*, *r*, *xs*)

Evaluates one decision rule in this controller
Takes a rule from the controller and evaluates it given the values of the input variables. The format of the rule is as given, and the response to the rule is calculated by:

```
y = a0 + a1*x1 + a2*x2 + ... + an*xn
```

Parameters

`r:` The rule in the standard format, or an integer number. If `r` is an integer, then the `r` th rule in the knowledge base will be evaluated.

`xs:` A tuple, a list or an array containing the values of the input variables. The dimension must be coherent with the given rule.

Return Value

This method evaluates each membership function in the rule for each given value, and `and` 's the results to obtain the condition. If the condition is zero, a tuple `(0.0, 0.0) is returned. Otherwise, the result as given above is calculate, and a tuple containing ‘‘(condition, result)` (the membership value associated to the condition and the result of the calculation) is returned.

__call__(*self*, **xs*)

Apply the controller to the set of input variables
Given the values of the input variables, evaluates every decision rule, and calculates the weighted average of the results. Returns the response of the controller.

Parameters

`xs:` A tuple, a list or an array with the values of the input variables.

Return Value

The response of the controller.

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

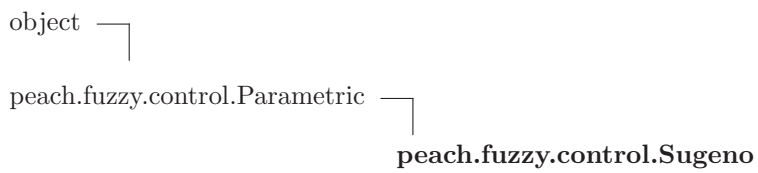
__str__(*x*)

str(x)

3.4.2 Properties

Name	Description
rules	Property that returns the list of decision rules. Not writable **Value:** `<property object at 0x887bedc>`
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

3.5 Class Sugeno



`Sugeno` is an alias to `Parametric`

3.5.1 Methods

__call__(*self*, **xs*)

Apply the controller to the set of input variables
Given the values of the input variables, evaluates every decision rule, and calculates the weighted average of the results. Returns the response of the controller.

Parameters

`xs`: A tuple, a list or an array with the values of the input variables.

Return Value

The response of the controller.

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__init__(*self*, *rules*=[], *norm*=<function ProbabilisticAnd at 0x885ebc4>, *conorm*=<function ProbabilisticOr at 0x885ebfc>, *negation*=<function ProbabilisticNot at 0x885ec34>)

Creates and initializes the controller.

Parameters

`rules:` List containing the decision rules for the controller. If not given, an empty set of decision rules is used.

`norm:` The norm (`and` operation) to be used. Defaults to Probabilistic and.

`conorm:` The conorm (`or` operation) to be used. Defaults to Probabilistic or.

`negation:` The negation (`not` operation) to be used. Defaults to Probabilistic not.

Overrides: object.__init__

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

add_rule(*self*, *rule*)

Adds a decision rule to the knowledge base.
It is essential to understand the format that decision rules must follow to obtain correct behaviour of the controller. A rule is a tuple given by:

```
((mx0, mx1, ..., mxn), (a0, a1, ..., an))
```

where `mx0` is a membership function of the first input variable, `mx1` is a membership function of the second input variable and so on; and `a0` is the linear parameter, `a1` is the parameter associated with the first input variable, `a2` is the parameter associated with the second input variable and so on.
Notice that `mx`'s are *functions* not fuzzy sets! They will be applied to the values of the input variables given in the function call, so, if they are anything different from a membership function, an exception will be raised. Please, consult the examples to see how they must be used.

eval(*self*, *r*, *xs*)

Evaluates one decision rule in this controller
Takes a rule from the controller and evaluates it given the values of the input variables. The format of the rule is as given, and the response to the rule is calculated by:

```
y = a0 + a1*x1 + a2*x2 + ... + an*xn
```

Parameters

`r`: The rule in the standard format, or an integer number. If `r` is an integer, then the `r` th rule in the knowledge base will be evaluated.

`xs`: A tuple, a list or an array containing the values of the input variables. The dimension must be coherent with the given rule.

Return Value

This method evaluates each membership function in the rule for each given value, and `and` 's the results to obtain the condition. If the condition is zero, a tuple `(0.0, 0.0) is returned. Otherwise, the result as given above is calculate, and a tuple containing ‘‘(condition, result)` (the membership value associated to the condition and the result of the calculation) is returned.

3.5.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`
rules	Property that returns the list of decision rules. Not writable **Value:** `<property object at 0x887bedc>`

4 Module peach.fuzzy.defuzzy

This package implements defuzzification methods for use with fuzzy controllers.

Defuzzification methods take a set of numerical values, their corresponding fuzzy membership values and calculate a defuzzified value for them. They're implemented as functions, not as classes. So, to implement your own, use the directions below.

These methods are implemented as functions with the signature `(mf, y)`, where `mf` is the fuzzy set, and `y` is an array of values. That is, `mf` is a fuzzy set containing the membership values of each one in the `y` array, in the respective order. Both arrays should have the same dimensions, or else the methods won't work.

See the example:

```
>>> import numpy
>>> from peach import *
>>> y = numpy.linspace(0., 5., 100)
>>> m_y = Triangle(1., 2., 3.)
>>> Centroid(m_y(y), y)
2.0001030715316435
```

The methods defined here are the most commonly used.

4.1 Functions

Centroid(*mf*, *y*)

Center of gravity method.
The center of gravity is calculate using the standard formula found in any calculus book. The integrals are calculated using the trapezoid method.

Parameters

`mf`: Fuzzy set containing the membership values of the elements in the vector given in sequence

`y`: Array of domain values of the defuzzified variable.

Return Value

The center of gravity of the fuzzy set.

Bissector(*mf*, *y*)

Bissection method
The bissection method finds a coordinate `y` in domain that divides the fuzzy set in two subsets with the same area. Integrals are calculated using the trapezoid method. This method only works if the values in `y` are equally spaced, otherwise, the method will fail.

Parameters

`mf`: Fuzzy set containing the membership values of the elements in the vector given in sequence

`y`: Array of domain values of the defuzzified variable.

Return Value

Defuzzified value by the bissection method.

SmallestOfMaxima(*mf*, *y*)

Smallest of maxima method.
This method finds all the points in the domain which have maximum membership value in the fuzzy set, and returns the smallest of them.

Parameters

`mf`: Fuzzy set containing the membership values of the elements in the vector given in sequence

`y`: Array of domain values of the defuzzified variable.

Return Value

Defuzzified value by the smallest of maxima method.

LargestOfMaxima(*mf*, *y*)

Largest of maxima method.
This method finds all the points in the domain which have maximum membership value in the fuzzy set, and returns the largest of them.

Parameters

`mf`: Fuzzy set containing the membership values of the elements in the vector given in sequence

`y`: Array of domain values of the defuzzified variable.

Return Value

Defuzzified value by the largest of maxima method.

MeanOfMaxima(*mf*, *y*)

Mean of maxima method.
This method finds the smallest and largest of maxima, and returns their average.

Parameters

`mf`: Fuzzy set containing the membership values of the elements in the vector given in sequence

`y`: Array of domain values of the defuzzified variable.

Return Value

Defuzzified value by the of maxima method.

4.2 Variables

Name	Description
__doc__	**Value:** ...

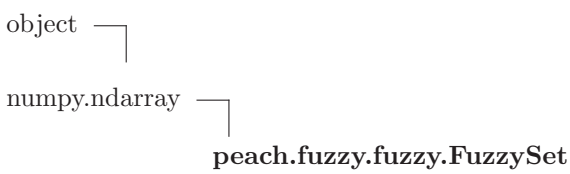
5 Module peach.fuzzy.fuzzy

This package implements basic definitions for fuzzy logic

5.1 Variables

Name	Description
__doc__	**Value:** ...

5.2 Class FuzzySet



Array containing fuzzy values for a set.

This class defines the behavior of a fuzzy set. It is an array of values in the range from 0 to 1, and the basic operations of the logic -- and (using the `&` operator); or (using the `|` operator); not (using `~` operator) -- can be defined according to a set of norms. The norms can be redefined using the appropriated methods.

To create a FuzzySet, instantiate this class with a sequence as argument, for example:

```
fuzzy_set = FuzzySet([ 0., 0.25, 0.5, 0.75, 1.0 ])
```

5.2.1 Methods

__new__(*cls*, *data*)

Allocates space for the array.
A fuzzy set is derived from the basic NumPy array, so the appropriate functions and methods are called to allocate the space. In theory, the values for a fuzzy set should be in the range `0.0 <= x <= 1.0`, but to increase efficiency, no verification is made.

Return Value
A new array object with the fuzzy set definitions.

Overrides: numpy.ndarray.__new__

__init__(*self*, *data*=`[]`)

Initializes the object.
Operations are defaulted to Zadeh norms `(max, min, 1-x)`

Overrides: object.__init__

__and__(*self*, *a*)

Fuzzy and (`&`) operation.

Overrides: numpy.ndarray.__and__

__or__(*self*, *a*)

Fuzzy or (|) operation.

Overrides: numpy.ndarray.__or__

__invert__(*self*)

Fuzzy not (~) operation.

Overrides: numpy.ndarray.__invert__

set_norm(*self*, *f*)

Selects a t-norm (and operation)
Use this method to change the behaviour of the and operation.

Parameters

`f:` A function of two parameters which must return the `and` of the values.

set_conorm(*self*, *f*)

Selects a t-conorm (or operation)
Use this method to change the behaviour of the or operation.

Parameters

`f:` A function of two parameters which must return the `or` of the values.

set_negation(*self*, *f*)

Selects a negation (not operation)
Use this method to change the behaviour of the not operation.

Parameters

`f:` A function of one parameter which must return the `not` of the value.

__abs__(*x*)

abs(x)

__add__(*x*, *y*)

x+y

__array__(...)

a.__array__(|dtype) -> reference if type unchanged, copy otherwise.
Returns either a new reference to self if dtype is not given or a new array of provided data type if dtype is different from the current dtype of the array.

__array_wrap__(*a*, *obj*)
Return Value

```
Object of same type as a from ndarray obj.
```

__contains__(*x*, *y*)

y in x

__copy__(*a*, *order*=...)

Return a copy of the array.

Parameters

order (**{'C', 'F', 'A'}, optional**)
: If order is 'C' (False) then the result is contiguous (default). If order is 'Fortran' (True) then the result has fortran order. If order is 'Any' (None) then the result has fortran order only if the array already is in fortran order.

__deepcopy__(*a*)

Used if copy.deepcopy is called on an array.

Return Value
: `Deep copy of array`

__delattr__(...)

x.__delattr__('name') <==> del x.name

__delitem__(*x*, *y*)

del x[y]

__delslice__(*x*, *i*, *j*)

del x[i:j]
Use of negative indices is not supported.

__div__(*x*, *y*)

x/y

__divmod__(*x*, *y*)

divmod(x, y)

__eq__(*x*, *y*)

x==y

__float__(*x*)

float(x)

__floordiv__(*x*, *y*)

x//y

__ge__(*x*, *y*)

x>=y

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__getitem__(*x*, *y*)

x[y]

__getslice__(*x*, *i*, *j*)

x[i:j]
Use of negative indices is not supported.

__gt__(*x*, *y*)

x>y

__hash__(*x*)

hash(x)

__hex__(*x*)

hex(x)

__iadd__(*x*, *y*)

x+y

__iand__(*x*, *y*)

x&y

__idiv__(*x*, *y*)

x/y

__ifloordiv__(*x*, *y*)

x//y

__ilshift__(*x*, *y*)

x<<y

__imod__(*x*, *y*)

x%y

__imul__(*x*, *y*)

x*y

__index__(...)

x[y:z] <==> x[y.__index__():z.__index__()]

__int__(*x*)

int(x)

__ior__(*x*, *y*)

x|y

__ipow__(*x*, *y*)

x**y

__irshift__(*x*, *y*)

x>>y

__isub__(*x*, *y*)

x-y

__iter__(*x*)

iter(x)

__itruediv__(*x*, *y*)

x/y

__ixor__(*x*, *y*)

x^y

__le__(*x*, *y*)

x<=y

__len__(*x*)

len(x)

__long__(*x*)

long(x)

__lshift__(*x*, *y*)

x<<y

__lt__(*x*, *y*)

x<y

__mod__(*x*, *y*)

x%y

__mul__(*x*, *y*)

x*y

__ne__(*x*, *y*)

x!=y

__neg__(*x*)

-x

__nonzero__(*x*)

x != 0

__oct__(*x*)

oct(x)

__pos__(*x*)

+x

__pow__(*x*, *y*, *z*=...)

pow(x, y[, z])

__radd__(*x*, *y*)

y+x

__rand__(*x*, *y*)

y&x

__rdiv__(*x*, *y*)

y/x

__rdivmod__(*x*, *y*)

divmod(y, x)

__reduce__(*a*)

For pickling.

Overrides: object.__reduce__

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

Overrides: object.__repr__

__rfloordiv__(*x*, *y*)

y//x

__rlshift__(*x*, *y*)

y<<x

__rmod__(*x*, *y*)

y%x

__rmul__(*x*, *y*)

y*x

__ror__(*x*, *y*)

y|x

__rpow__(*y*, *x*, *z*=*...*)

pow(x, y[, z])

__rrshift__(*x*, *y*)

y>>x

__rshift__(*x*, *y*)

x>>y

__rsub__(*x*, *y*)

y-x

__rtruediv__(*x*, *y*)

y/x

__rxor__(*x*, *y*)

y^x

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__setitem__(*x*, *i*, *y*)

x[i]=y

__setslice__(*x*, *i*, *j*, *y*)

x[i:j]=y
Use of negative indices is not supported.

__setstate__(*a*, *version*, *shape*, *dtype*, *isfortran*, *rawdata*)

```
For unpickling.

Parameters
----------
version : int
    optional pickle version. If omitted defaults to 0.
shape : tuple
dtype : data-type
isFortran : bool
rawdata : string or list
    a binary string with the data (or a list if 'a' is an object array)
```

__str__(*x*)

str(x)

Overrides: object.__str__

__sub__(*x*, *y*)

x-y

__truediv__(*x*, *y*)

x/y

__xor__(*x*, *y*)

x^y

all(*a*, *axis*=None, *out*=None)

Returns True if all elements evaluate to True.
Refer to `numpy.all` for full documentation.

See Also

numpy.all : equivalent function

any(*a*, *axis*=None, *out*=None)

Check if any of the elements of `a` are true.
Refer to `numpy.any` for full documentation.

See Also

numpy.any : equivalent function

argmax(*a*, *axis*=None, *out*=None)

Return indices of the maximum values along the given axis of `a`.

Parameters

axis (**int, optional**)
: Axis along which to operate. By default flattened input is used.

out (**ndarray, optional**)
: Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

Returns

index_array (**ndarray**)
: An array of indices or single index value, or a reference to `out` if it was specified.

Examples

```
>>> a = np.arange(6).reshape(2,3)
>>> a.argmax()
5
>>> a.argmax(0)
array([1, 1, 1])
>>> a.argmax(1)
array([2, 2])
```

argmin(*a*, *axis*=None, *out*=None)

Return indices of the minimum values along the given axis of `a`.
Refer to `numpy.ndarray.argmax` for detailed documentation.

argsort(*a*, *axis*=-1, *kind*='quicksort', *order*=None)

Returns the indices that would sort this array.
Refer to `numpy.argsort` for full documentation.

See Also

numpy.argsort : equivalent function

astype(*a*, *t*)

Copy of the array, cast to a specified type.

Parameters

t (**string or dtype**)
Typecode or data-type to which the array is cast.

Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([ 1. ,  2. ,  2.5])
>>> x.astype(int)
array([1, 2, 2])
```

byteswap(*a*, *inplace*)

Swap the bytes of the array elements
Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place.

Parameters

inplace: bool, optional If `True`, swap bytes in-place, default is `False`.

Returns

out: ndarray The byteswapped array. If `inplace` is `True`, this is a view to self.

Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> map(hex, A)
['0x1', '0x100', '0x2233']
>>> A.byteswap(True)
array([  256,     1, 13090], dtype=int16)
>>> map(hex, A)
['0x100', '0x1', '0x3322']
```

Arrays of strings are not swapped

```
>>> A = np.array(['ceg', 'fac'])
>>> A.byteswap()
array(['ceg', 'fac'],
      dtype='|S3')
```

choose(*a*, *choices*, *out*=`None`, *mode*=`'raise'`)

Use an index array to construct a new array from a set of choices.
Refer to `numpy.choose` for full documentation.

See Also

numpy.choose : equivalent function

clip(*a*, *a_min*, *a_max*, *out*=`None`)

Return an array whose values are limited to `[a_min, a_max]`.
Refer to `numpy.clip` for full documentation.

See Also

numpy.clip : equivalent function

compress(*a*, *condition*, *axis*=`None`, *out*=`None`)

Return selected slices of this array along given axis.
Refer to `numpy.compress` for full documentation.

See Also

numpy.compress : equivalent function

conj(*a*)

Return an array with all complex-valued elements conjugated.

conjugate(*a*)

Return an array with all complex-valued elements conjugated.

copy(*a*, *order*='C')

Return a copy of the array.

Parameters

order ({'C', 'F', 'A'}, **optional**)
By default, the result is stored in C-contiguous (row-major) order in memory. If `order` is `F`, the result has 'Fortran' (column-major) order. If order is 'A' ('Any'), then the result has the same order as the input.

Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
>>> y = x.copy()
>>> x.fill(0)
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
>>> y.flags['C_CONTIGUOUS']
True
```

cumprod(*a*, *axis*=None, *dtype*=None, *out*=None)

Return the cumulative product of the elements along the given axis.
Refer to `numpy.cumprod` for full documentation.

See Also

numpy.cumprod : equivalent function

cumsum(*a*, *axis*=None, *dtype*=None, *out*=None)

Return the cumulative sum of the elements along the given axis.
Refer to `numpy.cumsum` for full documentation.

See Also

numpy.cumsum : equivalent function

diagonal(*a*, *offset*=0, *axis1*=0, *axis2*=1)

Return specified diagonals.
Refer to `numpy.diagonal` for full documentation.

See Also

numpy.diagonal : equivalent function

dump(*a*, *file*)

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

Parameters

file (**str**)
: A string naming the dump file.

dumps(*a*)

Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

fill(*a*, *value*)

Fill the array with a scalar value.

Parameters

a (**ndarray**)
: Input array

value (**scalar**)
: All elements of `a` will be assigned this value.

Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([ 1.,  1.])
```

flatten(*a*, *order*='C')

Collapse an array into one dimension.

Parameters

order (**{'C', 'F'}, optional**)
Whether to flatten in C (row-major) or Fortran (column-major) order. The default is 'C'.

Returns

y (**ndarray**)
A copy of the input array, flattened to one dimension.

Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

getfield(*a*, *dtype*, *offset*)

Returns a field of the given array as a certain type. A field is a view of the array data with each itemsize determined by the given type and the offset into the current array.

item(*a*)

Copy the first element of array to a standard Python scalar and return it. The array must be of size one.

itemset(...)

max(*a*, *axis*=None, *out*=None)

Return the maximum along a given axis.
Refer to `numpy.amax` for full documentation.

See Also

numpy.amax : equivalent function

mean(*a*, *axis*=None, *dtype*=None, *out*=None)

Returns the average of the array elements along given axis.
Refer to `numpy.mean` for full documentation.

See Also

numpy.mean : equivalent function

min(*a*, *axis*=None, *out*=None)

Return the minimum along a given axis.
Refer to `numpy.amin` for full documentation.

See Also

numpy.amin : equivalent function

newbyteorder(*a*, *byteorder*)

Equivalent to a.view(a.dtype.newbytorder(byteorder))

nonzero(*a*)

Return the indices of the elements that are non-zero.
Refer to `numpy.nonzero` for full documentation.

See Also

numpy.nonzero : equivalent function

prod(*a*, *axis*=None, *dtype*=None, *out*=None)

Return the product of the array elements over the given axis
Refer to `numpy.prod` for full documentation.

See Also

numpy.prod : equivalent function

ptp(*a*, *axis*=None, *out*=None)

Peak to peak (maximum - minimum) value along a given axis.
Refer to `numpy.ptp` for full documentation.

See Also

numpy.ptp : equivalent function

put(*a*, *indices*, *values*, *mode*=`'raise'`)

Set a.flat[n] = values[n] for all n in indices.
Refer to `numpy.put` for full documentation.

See Also

numpy.put : equivalent function

ravel(*a*, *order*=`...`)

Return a flattened array.
Refer to `numpy.ravel` for full documentation.

See Also

numpy.ravel : equivalent function

repeat(*a*, *repeats*, *axis*=`None`)

Repeat elements of an array.
Refer to `numpy.repeat` for full documentation.

See Also

numpy.repeat : equivalent function

reshape(*a*, *shape*, *order*=`'C'`)

Returns an array containing the same data with a new shape.
Refer to `numpy.reshape` for full documentation.

See Also

numpy.reshape : equivalent function

resize(*a*, *new_shape*, *refcheck*=`True`, *order*=`False`)

Change shape and size of array in-place.

Parameters

a (**ndarray**)
: Input array.

new_shape (**{tuple, int}**)
: Shape of resized array.

refcheck (**bool, optional**)
: If False, memory referencing will not be checked. Default is True.

order (**bool, optional**)
: <needs an explanation>. Default if False.

Returns

None

Raises

ValueError If `a` does not own its own data, or references or views to it exist.

Examples

Shrinking an array: array is flattened in C-order, resized, and reshaped:

```
>>> a = np.array([[0,1],[2,3]])
>>> a.resize((2,1))
>>> a
array([[0],
       [1]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0,1],[2,3]])
>>> b.resize((2,3))
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing:

```
>>> c = a
>>> a.resize((1,1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that has been referenced ...
```

round(*a*, *decimals*=`0`, *out*=`None`)

Return an array rounded a to the given number of decimals.
Refer to `numpy.around` for full documentation.

See Also

numpy.around : equivalent function

searchsorted(*a*, *v*, *side*=`'left'`)

Find indices where elements of v should be inserted in a to maintain order.
For full documentation, see `numpy.searchsorted`

See Also

numpy.searchsorted : equivalent function

setfield(*m*, *value*, *dtype*, *offset*)

places val into field of the given array defined by the data type and offset.

Return Value

`None`

setflags(*a*, *write*=`None`, *align*=`None`, *uic*=`None`)

sort(*a*, *axis*=-1, *kind*='quicksort', *order*=None)

Sort an array, in-place.

Parameters

axis (**int, optional**)
: Axis along which to sort. Default is -1, which means sort along the last axis.

kind (**{'quicksort', 'mergesort', 'heapsort'}, optional**)
: Sorting algorithm. Default is 'quicksort'.

order (**list, optional**)
: When `a` is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

See Also

numpy.sort : Return a sorted copy of an array. argsort : Indirect sort. lexsort : Indirect stable sort on multiple keys. searchsorted : Find elements in sorted array.

Notes

See `sort` for notes on the different sorting algorithms.

Examples

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the `order` keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([('c', 1), ('a', 2)],
      dtype=[('x', '|S1'), ('y', '<i4')])
```

squeeze(*a*)

Remove single-dimensional entries from the shape of `a`.
Refer to `numpy.squeeze` for full documentation.

See Also

numpy.squeeze : equivalent function

std(*a*, *axis*=`None`, *dtype*=`None`, *out*=`None`, *ddof*=`0`)

Returns the standard deviation of the array elements along given axis.
Refer to `numpy.std` for full documentation.

See Also

numpy.std : equivalent function

sum(*a*, *axis*=`None`, *dtype*=`None`, *out*=`None`)

Return the sum of the array elements over the given axis.
Refer to `numpy.sum` for full documentation.

See Also

numpy.sum : equivalent function

swapaxes(*a*, *axis1*, *axis2*)

Return a view of the array with `axis1` and `axis2` interchanged.
Refer to `numpy.swapaxes` for full documentation.

See Also

numpy.swapaxes : equivalent function

take(*a*, *indices*, *axis*=`None`, *out*=`None`, *mode*=`'raise'`)

Return an array formed from the elements of a at the given indices.
Refer to `numpy.take` for full documentation.

See Also

numpy.take : equivalent function

tofile(*a*, *fid*, *sep*="", *format*="%s")

Write array to a file as text or binary.
Data is always written in 'C' order, independently of the order of `a`. The data produced by this method can be recovered by using the function fromfile().
This is a convenience function for quick storage of array data. Information on endianess and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianess. Some of these problems can be overcome by outputting the data as text files at the expense of speed and file size.

Parameters

fid (**file or string**)
: An open file object or a string containing a filename.

sep (**string**)
: Separator between array items for text output. If "" (empty), a binary file is written, equivalently to file.write(a.tostring()).

format (**string**)
: Format string for text file output. Each entry in the array is formatted to text by converting it to the closest Python type, and using "format" % item.

tolist(*a*)

Return the array as a possibly nested list.
Return a copy of the array data as a hierarchical Python list. Data items are converted to the nearest compatible Python type.

Parameters

none

Returns

y (**list**)
The possibly nested list of array elements.

Notes

The array may be recreated, `a = np.array(a.tolist())`.

Examples

```
>>> a = np.array([1, 2])
>>> a.tolist()
[1, 2]
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

tostring(*a*, *order*='C')

Construct a Python string containing the raw data bytes in the array.

Parameters

order ({'C', 'F', None})
Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

trace(*a*, *offset*=0, *axis1*=0, *axis2*=1, *dtype*=None, *out*=None)

Return the sum along diagonals of the array.
Refer to `numpy.trace` for full documentation.

See Also

numpy.trace : equivalent function

transpose(*a*, **axes*)

Returns a view of 'a' with axes transposed. If no axes are given, or None is passed, switches the order of the axes. For a 2-d array, this is the usual matrix transpose. If axes are given, they describe how the axes are permuted.

Examples

```
>>> a = np.array([[1,2],[3,4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1,0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1,0)
array([[1, 3],
       [2, 4]])
```

var(*a*, *axis*=None, *dtype*=None, *out*=None, *ddof*=0)

Returns the variance of the array elements, along given axis.
Refer to `numpy.var` for full documentation.

See Also

numpy.var : equivalent function

view(*a*, *dtype*=None, *type*=None)

New view of array with the same data.

Parameters

dtype (**data-type**)
Data-type descriptor of the returned view, e.g. float32 or int16.

type (**python type**)
Type of the returned view, e.g. ndarray or matrix.

Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> print y.dtype
int16
>>> print type(y)
<class 'numpy.core.defmatrix.matrix'>
```

Using a view to convert an array to a record array:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

5.2.2 Properties

Name	Description
T	**Value:** <attribute 'T' of 'numpy.ndarray' objects>
__array_finalize__	**Value:** <attribute '__array_finalize__' of 'numpy.ndarray' objects>
__array_interface__	**Value:** <attribute '__array_interface__' of 'numpy.ndarray' objects>
__array_priority__	**Value:** <attribute '__array_priority__' of 'numpy.ndarray' objects>
__array_struct__	**Value:** <attribute '__array_struct__' of 'numpy.ndarray' objects>
__class__	**Value:** <attribute '__class__' of 'object' objects>
base	**Value:** <attribute 'base' of 'numpy.ndarray' objects>
ctypes	**Value:** <attribute 'ctypes' of 'numpy.ndarray' objects>
data	**Value:** <attribute 'data' of 'numpy.ndarray' objects>
dtype	**Value:** <attribute 'dtype' of 'numpy.ndarray' objects>

continued on next page

Name	Description
flags	**Value:** <attribute 'flags' of 'numpy.ndarray' objects>
flat	**Value:** <attribute 'flat' of 'numpy.ndarray' objects>
imag	**Value:** <attribute 'imag' of 'numpy.ndarray' objects>
itemsize	**Value:** <attribute 'itemsize' of 'numpy.ndarray' objects>
nbytes	**Value:** <attribute 'nbytes' of 'numpy.ndarray' objects>
ndim	**Value:** <attribute 'ndim' of 'numpy.ndarray' objects>
real	**Value:** <attribute 'real' of 'numpy.ndarray' objects>
shape	**Value:** <attribute 'shape' of 'numpy.ndarray' objects>
size	**Value:** <attribute 'size' of 'numpy.ndarray' objects>
strides	**Value:** <attribute 'strides' of 'numpy.ndarray' objects>

6 Module peach.fuzzy.mf

Membership functions

Membership functions are actually subclasses of a main class called Membership, see below. Instantiate a class to generate a function, optional arguments can be specified to configure the function as needed. For example, to create a triangle function starting at 0, with peak in 3, and ending in 4, use:

```
mu = Triangle(0, 3, 4)
```

Please notice that the return value is a *function*. To use it, apply it as a normal function. For example, the function above, applied to the value 1.5 should return 0.5:

```
>>> print mu(1.5)
0.5
```

6.1 Functions

Saw(*interval*, *n*)

Splits an `interval` into `n` triangle functions.
Given an interval in any domain, this function will create `n` triangle functions of the same size equally spaced in the interval. It is very useful to create membership functions for controllers. The command below will create 3 triangle functions equally spaced in the interval (0, 4):

```
mf1, mf2, mf3 = Saw((0, 4), 3)
```

This is the same as the following commands:

```
mf1 = Triangle(0, 1, 2)
mf2 = Triangle(1, 2, 3)
mf3 = Triangle(2, 3, 4)
```

Parameters

`interval`: A tuple containing the start and the end of the interval, in the format `(start, end)`;

`n`: The number of functions in which the interval must be split.

Return Value

A list of triangle membership functions, in order.

FlatSaw(*interval*, *n*)

Splits an `interval` into a decreasing ramp, `n-2` triangle functions and an increasing ramp.
Given an interval in any domain, this function will create a decreasing ramp in the start of the interval, `n-2` triangle functions of the same size equally spaced in the interval, and a increasing ramp in the end of the interval. It is very useful to create membership functions for controllers. The command below will create a decreasing ramp, a triangle function and an increasing ramp equally spaced in the interval (0, 2):

```
mf1, mf2, mf3 = FlatSaw((0, 2), 3)
```

This is the same as the following commands:

```
mf1 = DecreasingRamp(0, 1)
mf2 = Triangle(0, 1, 2)
mf3 = Increasingramp(1, 2)
```

Parameters

`interval`: A tuple containing the start and the end of the interval, in the format `(start, end)`;

`n`: The number of functions in which the interval must be split.

Return Value

A list of corresponding functions, in order.

6.2 Variables

Name	Description
__doc__	**Value:** ...

6.3 Class Membership

object
└ **peach.fuzzy.mf.Membership**

Known Subclasses: peach.fuzzy.mf.Bell, peach.fuzzy.mf.DecreasingRamp, peach.fuzzy.mf.DecreasingSigmoid, peach.fuzzy.mf.Gaussian, peach.fuzzy.mf.IncreasingRamp, peach.fuzzy.mf.IncreasingSigmoid, peach.fuzzy.mf.RaisedCosine, peach.fuzzy.mf.Trapezoid, peach.fuzzy.mf.Triangle

Base class of all membership functions.

This class is used as base of the implemented membership functions, and can also be used to transform a regular function in a membership function that can be used with the fuzzy logic package.

To create a membership function from a regular function `f`, use:

```
mf = Membership(f)
```

A function this converted can be used with vectors and matrices and always return a FuzzySet object. Notice that the value range is not verified so that it fits in the range [0, 1]. It is responsibility of the programmer to warrant that.

To subclass Membership, just use it as a base class. It is suggested that the `__init__` method of the derived class allows configuration, and the `__call__` method is used to apply the function over its arguments.

6.3.1 Methods

__init__(*self*, *f*)

Builds a membership function from a regular function

Parameters

`f:` Function to be transformed into a membership function. It must be given, and it must be a `FunctionType` object, otherwise, a `ValueError` is raised.

Overrides: object.__init__

__call__(*self*, *x*)

Maps the function on a vector

Parameters

`x:` A value, vector or matrix over which the function is evaluated.

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

6.3.2 Properties

Name	Description
__class__	**Value:** <attribute '__class__' of 'object' objects>

6.4 Class IncreasingRamp

object

peach.fuzzy.mf.Membership

peach.fuzzy.mf.IncreasingRamp

Increasing ramp.

Given two points, `x0` and `x1`, with `x0 < x1`, creates a function which returns:

0, if `x <= x0`;

`(x - x0) / (x1 - x0)`, if `x0 < x <= x1`;

1, if `x > x1`.

6.4.1 Methods

__init__(*self*, *x0*, *x1*)

Initializes the function.

Parameters

`x0`: Start of the ramp;
`x1`: End of the ramp.

Overrides: peach.fuzzy.mf.Membership.__init__

__call__(*self*, *x*)

Maps the function on a vector

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)
Return Value
`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

6.4.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

6.5 Class DecreasingRamp

object
peach.fuzzy.mf.Membership
peach.fuzzy.mf.DecreasingRamp

Decreasing ramp.

Given two points, `x0` and `x1`, with `x0 < x1`, creates a function which returns:

1, if `x <= x0`;

`(x1 - x) / (x1 - x0)`, if `x0 < x <= x1`;

0, if `x > x1`.

6.5.1 Methods

__init__(*self*, *x0*, *x1*)

Initializes the function.

Parameters

`x0`: Start of the ramp;
`x1`: End of the ramp.

Overrides: peach.fuzzy.mf.Membership.__init__

__call__(*self*, *x*)

Maps the function on a vector

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

6.5.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

6.6 Class Triangle

object

peach.fuzzy.mf.Membership

peach.fuzzy.mf.Triangle

Triangle function.

Given three points, `x0`, `x1` and `x2`, with `x0 < x1 < x2`, creates a function which returns:

0, if `x <= x0` or `x > x2`;

`(x - x0) / (x1 - x0)`, if `x0 < x <= x1`;

`(x2 - x) / (x2 - x1)`, if `x1 < x <= x2`.

6.6.1 Methods

__init__(*self*, *x0*, *x1*, *x2*)

Initializes the function.

Parameters

`x0`: Start of the triangle;
`x1`: Peak of the triangle;
`x2`: End of triangle.

Overrides: peach.fuzzy.mf.Membership.__init__

__call__(*self*, *x*)

Maps the function on a vector

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)
Return Value
`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

6.6.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

6.7 Class Trapezoid

object

peach.fuzzy.mf.Membership

peach.fuzzy.mf.Trapezoid

Trapezoid function.

Given four points, `x0`, `x1`, `x2` and `x3`, with `x0 < x1 < x2 < x3`, creates a function which returns:

0, if `x <= x0` or `x > x3`;

`(x - x0)/(x1 - x0)`, if `x0 <= x < x1`;

1, if `x1 <= x < x2`;

`(x3 - x)/(x3 - x2)`, if `x2 <= x < x3`.

6.7.1 Methods

__init__(*self*, *x0*, *x1*, *x2*, *x3*)

Initializes the function.

Parameters

`x0`: Start of the trapezoid;
`x1`: First peak of the trapezoid;
`x2`: Last peak of the trapezoid;
`x3`: End of trapezoid.

Overrides: peach.fuzzy.mf.Membership.__init__

__call__(*self*, *x*)

Maps the function on a vector

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

6.7.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

6.8 Class Gaussian

object

peach.fuzzy.mf.Membership

peach.fuzzy.mf.Gaussian

Gaussian function.

Given the center and the width, creates a function which returns a gaussian fit to these parameters, that is:

```
exp(-a*(x - x0)**2)
```

6.8.1 Methods

__init__(*self*, *x0*=`0.0`, *a*=`1.0`)

Initializes the function.

Parameters

`x0`: Center of the gaussian. Default value `0.0`;
`a`: Width of the gaussian. Default value `1.0`.

Overrides: peach.fuzzy.mf.Membership.__init__

__call__(*self*, *x*)
Maps the function on a vector

Return Value
A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)
Return Value
`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

6.8.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

6.9 Class IncreasingSigmoid

object
peach.fuzzy.mf.Membership
peach.fuzzy.mf.IncreasingSigmoid

Increasing Sigmoid function.

Given the center and the slope, creates an increasing sigmoidal function. It goes to `0` as `x` approaches to -infinity, and goes to `1` as `x` approaches infinity, that is:

```
1 / (1 + exp(-a*(x - x0))
```

6.9.1 Methods

__init__(*self*, *x0*=`0.0`, *a*=`1.0`)

Initializes the function.

Parameters

`x0`: Center of the sigmoid. Default value `0.0`. The function evaluates to `0.5` if `x = x0`;

`a`: Slope of the sigmoid. Default value `1.0`.

Overrides: peach.fuzzy.mf.Membership.__init__

__call__(*self*, *x*)

Maps the function on a vector

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

6.9.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

6.10 Class DecreasingSigmoid

object

peach.fuzzy.mf.Membership

peach.fuzzy.mf.DecreasingSigmoid

Decreasing Sigmoid function.

Given the center and the slope, creates an decreasing sigmoidal function. It goes to `1` as `x` approaches to -infinity, and goes to `0` as `x` approaches infinity, that is:

```
1 / (1 + exp(a*(x - x0))
```

6.10.1 Methods

__init__(*self*, *x0*=`0.0`, *a*=`1.0`)

Initializes the function.

Parameters

`x0`: Center of the sigmoid. Default value `0.0`. The function evaluates to `0.5` if `x = x0`;

`a`: Slope of the sigmoid. Default value `1.0`.

Overrides: peach.fuzzy.mf.Membership.__init__

__call__(*self*, *x*)
Maps the function on a vector

Return Value
A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)
Return Value
`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

6.10.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

6.11 Class RaisedCosine

object
peach.fuzzy.mf.Membership
peach.fuzzy.mf.RaisedCosine

Raised Cosine function.

Given the center and the frequency, creates a function that is a period of a raised cosine, that is:

0, if `x <= xm - pi/w` or `x > xm + pi/w`;

`0.5 + 0.5 * cos(w*(x - xm))`, if `xm - pi/w <= x < xm + pi/w`;

6.11.1 Methods

__init__(*self*, *xm*=`0.0`, *w*=`1.0`)

Initializes the function.

Parameters

`xm`: Center of the cosine. Default value `0.0`. The function evaluates to `1` if `x = xm`;

`w`: Frequency of the cosine. Default value `1.0`.

Overrides: peach.fuzzy.mf.Membership.__init__

__call__(*self*, *x*)

Maps the function on a vector

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

6.11.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

6.12 Class Bell

object

peach.fuzzy.mf.Membership

peach.fuzzy.mf.Bell

Generalized Bell function.

A generalized bell is a symmetric function with its peak in its center and fast decreasing to `0` outside a given interval, that is:

```
1 / (1 + ((x - x0)/a)**(2*b))
```

6.12.1 Methods

__init__(*self*, *x0*=`0.0`, *a*=`1.0`, *b*=`1.0`)

Initializes the function.

Parameters

`x0`: Center of the bell. Default value `0.0`. The function evaluates to `1` if `x = xm`;

`a`: Size of the interval. Default value `1.0`. A generalized bell evaluates to `0.5` if `x = -a` or `x = a`;

`b`: Measure of *flatness* of the bell. The bigger the value of `b`, the flatter is the resulting function. Default value `1.0`.

Overrides: peach.fuzzy.mf.Membership.__init__

__call__(*self*, *x*)

Maps the function on a vector

Return Value

A `FuzzySet` object containing the evaluation of the function over each of the components of the input.

Overrides: peach.fuzzy.mf.Membership.__call__ extit(inherited documentation)

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

6.12.2 Properties

Name	Description
__class__	**Value:** <attribute '__class__' of 'object' objects>

7 Module peach.fuzzy.norms

This package implements operations of fuzzy logic.

Basic operations are `and (&)`, `or (|)` and `not (~)`. Those are implemented as functions of, respectively, two, two and one values. The `and` is the t-norm of the fuzzy logic, and it is a function that takes two values and returns the result of the `and` operation. The `or` is a function that takes two values and returns the result of the `or` operation. the `not` is a function that takes one value and returns the result of the `not` operation. To implement your own operations there is no need to subclass -- just create the functions and use them where appropriate.

Also, implication and aglutination functions are defined here. Implication is the result of the generalized modus ponens used in fuzzy inference systems. Aglutination is the generalization from two different conclusions used in fuzzy inference systems. Both are implemented as functions that take two values and return the result of the operation. As above, to implement your own operations, there is no need to subclass -- just create the functions and use them where appropriate.

The functions here are provided as convenience.

7.1 Functions

ZadehAnd(x, y)

And operation as defined by Lofti Zadeh.
And operation is the minimum of the two values.

Return Value
The result of the and operation.

ZadehOr(x, y)

Or operation as defined by Lofti Zadeh.
Or operation is the maximum of the two values.

Return Value
The result of the or operation.

ZadehNot(x)

Not operation as defined by Lofti Zadeh.
Not operation is the complement to 1 of the given value, that is, `1 - x`.

Return Value
The result of the not operation.

MamdaniImplication(x, y)

Implication operation as defined by Mamdani.
Implication is the minimum of the two values.

Return Value
The result of the implication.

MamdaniAglutination(*x*, *y*)

Aglutination as defined by Mamdani.
Aglutination is the maximum of the two values.

Return Value
The result of the aglutination.

ProbabilisticAnd(*x*, *y*)

And operation as a probabilistic operation.
And operation is the product of the two values.

Return Value
The result of the and operation.

ProbabilisticOr(*x*, *y*)

Or operation as a probabilistic operation.
Or operation is given as the probability of the intersection of two events, that is, x + y - xy.

Return Value
The result of the or operation.

ProbabilisticNot(*x*)

Not operation as a probabilistic operation.
Not operation is the complement to 1 of the given value, that is, `1 - x`.

Return Value
The result of the not operation.

ProbabilisticImplication(*x*, *y*)

Implication as a probabilistic operation.
Implication is the product of the two values.

Return Value
The result of the and implication.

ProbabilisticAglutination(*x*, *y*)

Implication as a probabilistic operation.
Implication is given as the probability of the intersection of two events, that is, x + y - xy.

Return Value
The result of the and implication.

7.2 Variables

Name	Description
__doc__	**Value:** ...
ZADEH_NORMS	Tuple containing, in order, Zadeh and, or and not operations **Value:** `ZadehAnd, ZadehOr, ZadehNot`

continued on next page

Name	Description
MAMDANI_INFERENCE	Tuple containing, in order, Mamdani implication and algutination **Value:** `MamdaniImplication, MamdaniAglutination`
PROB_NORMS	Tuple containing, in order, probabilistic and, or and not operations **Value:** `ProbabilisticAnd, ProbabilisticOr, ProbabilisticNot`
PROB_INFERENCE	Tuple containing, in order, probabilistic implication and algutination **Value:** `ProbabilisticImplication, ProbabilisticAglutination`

8 Package peach.ga

This package implements genetic algorithms. Consult:

chromosome Basic definitions to work with chromosomes. Defined as arrays of bits;

crossover Defines crossover operators and base classes;

fitness Defines fitness functions and base classes;

ga Implementation of the basic genetic algorithm;

mutation Defines mutation operators and base classes;

selection Defines selection operators and base classes;

8.1 Modules

- **chromosome**: Basic definitions and classes for manipulating chromosomes
(Section 9, p. 69)
- **crossover**: Basic definitions for crossover operations and base classes.
(Section 10, p. 77)
- **fitness**: Basic definitions and base classes for definition of fitness functions for use with genetic algorithms.
(Section 11, p. 84)
- **ga**: Basic Genetic Algorithm (GA)
(Section 12, p. 89)
- **mutation**: Basic definitions and classes for operating mutation on chromosomes.
(Section 13, p. 97)
- **selection**: Basic classes and definitions for selection operator.
(Section 14, p. 101)

8.2 Variables

Name	Description
__doc__	**Value:** ...

9 Module peach.ga.chromosome

Basic definitions and classes for manipulating chromosomes

This sub-package is a vital part of the genetic algorithms framework within the module. This uses the `bitarray` module to implement a chromosome as an array of bits. It is, thus, necessary that this module is installed in your Python system. Please, check within the Python website how to install the `bitarray` module.

The class defined in this module is derived from `bitarray` and can also be derived if needed. In general, users or programmers won't need to instance this class directly -- it is manipulated by the genetic algorithm itself. Check the class definition for more information.

9.1 Variables

Name	Description
__doc__	**Value:** ...

9.2 Class Chromosome

Implements a chromosome as a bit array.

Data is structured according to the `struct` module that exists in the Python standard library. Internally, data used in optimization with a genetic algorithm are represented as arrays of bits, so the `bitarray` module must be installed. Please consult the Python package index for more information on how to install `bitarray`. In general, the user don't need to worry about how the data is manipulated internally, but a specification of the format as in the `struct` module is needed.

If the internal format of the data is specified as an `struct` format, the genetic algorithm will take care of encoding and decoding data from and to the optimizer. However, it is possible to specify, instead of a format, the length of the chromosome. In that case, the fitness function must deal with the encoding and decoding of the information. It is strongly suggested that you use `struct` format strings, as they are much easier. This second option is provided as a convenience.

The `Chromosome` class is derived from the `bitarray` class. So, every property and method of this class should be accessible.

9.2.1 Methods

__new__(*cls*, *fmt*='', *endian*='little')

Allocates new memory space for the chromosome
This function overrides the `bitarray.__new__` function to deal with the length of the chromosome. It should never be directly used, as it is automatically called by the Python interpreter in the moment of object creation.

Return Value

A new `Chromosome` object.

Overrides: bitarray._bitarray.__new__

__init__(*self*, *fmt*='')

Initializes the chromosome.
This method is automatically called by the Python interpreter and initializes the data in the chromosome. No data should be provided to be encoded in the chromosome, as it is usually better start with random estimates. This method, in particular, does not clear the memory used in the time of creation of the `bitarray` from which a `Chromosome` derives -- so the random noise in the memory is used as initial value.

Parameters

`fmt`: This parameter can be passed in two different ways. If `fmt` is a string, then it is assumed to be a `struct`-format string. Its size is calculated and a `bitarray` of the corresponding size is created. Please, consult the `struct` documentation, since what is explained there is exactly what is used here. For example, if you are going to use the optimizer to deal with three-dimensional vectors of continuous variables, the format would be something like:

```
fmt = 'fff'
```

If `fmt`, however, is an integer, then a `bitarray` of the given length is created. Note that, in this case, no format is given to the chromosome, and it is responsability of the programmer and the fitness function to provide for it.
Default value is an empty string.

Overrides: object.__init__

__get_size(*self*)

decode(*self*)

This method decodes the information given in the chromosome.
Data in the chromosome is encoded as a `struct`-formated string in a `bitarray` object. This method decodes the information and returns the encoded values. If a format string is not given, then it is assumed that this chromosome is just an array of bits, which is returned.

Return Value

A tuple containing the decoded values, in the order specified by the format string.

Overrides: bitarray._bitarray.decode

encode(*self*, *values*)

This method encodes the information into the chromosome.
Data in the chromosome is encoded as a `struct`-formated string in a `bitarray` object. This method encodes the given information in the bitarray. If a format string is not given, this method raises a `TypeError` exception.

Parameters

`values`: A tuple containing the values to be encoded in an order consistent with the given `struct`-format.

Overrides: bitarray._bitarray.encode

__add__(...)

__and__(...)

__contains__(*x*)

Return True if bitarray contains x, False otherwise. If x is an integer (which includes booleans), it is determined whether or not the corresponding bit is contained in the bitarray. If x is an object which can be cast into a bitarray, such as e.g. the string '0110', a list, or a bitarray itself, a sequential search will be performed to determine return value.

__copy__()

Return a copy of the bitarray.

__deepcopy__()

Return a copy of the bitarray.

__delattr__(...)

x.__delattr__('name') <==> del x.name

__delitem__(...)

__eq__(*x*, *y*)

x==y

__ge__(*x*, *y*)

x>=y

__getattribute__(...)

x.__getattribute__('name') <==> x.name

Overrides: object.__getattribute__

__getitem__(...)

__gt__(*x*, *y*)

x>y

__hash__(*x*)

hash(x)

__iadd__(...)

__iand__(...)

__imul__(...)

__invert__(...)

__ior__(...)

__iter__(*x*)

iter(x)

__ixor__(...)

__le__(*x*, *y*)

x<=y

__len__()

Return the length, i.e. number of bits stored in the bitarray. This method will fail for a bitarray object with 2^31 or more elements on a 32bit machine. Use bitarray.length() instead.

__lt__(*x*, *y*)

x<y

__mul__(...)

__ne__(*x*, *y*)

x!=y

__or__(...)

__reduce__(...)

Return state information for pickling.

Overrides: object.__reduce__

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

Overrides: object.__repr__

__rmul__(...)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__setitem__(...)

__str__(*x*)

str(x)

__xor__(...)

all()

Returns True when all bits in the array are True.

any()

Returns True when any bit in the array is True.

append(*x*)

Append the value bool(x) to the end of the bitarray.

buffer_info()

Return a tuple (address, size, endianness, unused, allocated) giving the current memory address, the size (in bytes) used to hold the bitarray's contents, the bit endianness as a string, the number of unused bits (e.g. a bitarray of length 11 will have a buffer size of 2 bytes and 5 unused bits), and the size (in bytes) of the allocated memory.

bytereverse()

For all bytes representing the bitarray, reverse the bit order (in-place). Note: This method changes the actual machine values representing the bitarray; it does not change the endianness of the bitarray object.

copy()

Return a copy of the bitarray.

count(*x*=...)

Return number of occurrences of x in the bitarray. x defaults to True.

endian()

Return the bit endianness as a string (either 'little' or 'big').

extend(*object*)

Append bits to the end of the bitarray. The objects which can be passed to this method are the same iterable objects which can given to a bitarray object upon initialization.

fill()

Returns the number of bits added (0..7) at the end of the array. When the length of the bitarray is not a multiple of 8, increase the length slightly such that the new length is a multiple of 8, and set the few new bits to False.

fromfile(*f*, *n*=...)

Read n bytes from the file object f and append them to the bitarray interpreted as machine values. When n is omitted, as many bytes are read until EOF is reached.

fromstring(*string*)

Append from a string, interpreting the string as machine values.

index(*x*)

Return index of first occurrence of x in the bitarray. It is an error when x does not occur in the bitarray

insert(*i*, *x*)

Insert a new item x into the bitarray before position i.

invert(*x*)

Invert all bits in the array (in-place), i.e. convert each 1-bit into a 0-bit and vice versa.

length()

Return the length, i.e. number of bits stored in the bitarray. This method is preferred over __len__, [used when typing `len(a)`], since __len__ will fail for a bitarray object with 2^31 or more elements on a 32bit machine, whereas this method will return the correct value, on 32bit and 64bit machines.

pack(*string*)

Extend the bitarray from a string, where each characters corresponds to a single bit. The character 'x00' maps to bit 0 and all other characters map to bit 1. This method, as well as the unpack method, are meant for efficient transfer of data between bitarray objects to other python objects (for example NumPy's ndarray object) which have a different view of memory.

pop(*i*=...)

Return the i-th element and delete it from the bitarray. i defaults to -1.

remove(*x*)

Remove the first occurrence of x in the bitarray.

reverse()

Reverse the order of bits in the array (in-place).

search(*x*, *limit*=...)

Given a bitarray x (or an object which can be converted to a bitarray), returns the start positions of x matching self as a list. The optional argument limits the number of search results to the integer specified. By default, all search results are returned.

setall(*x*)

Set all bits in the bitarray to bool(x).

sort(*reverse*=`False`)

Sort the bits in the array (in-place).

to01()

Return a string containing '0's and '1's, representing the bits in the bitarray object. Note: To extend a bitarray from a string containing '0's and '1's, use the extend method.

tofile(*f*)

Write all bits (as machine values) to the file object f. When the length of the bitarray is not a multiple of 8, the remaining bits (1..7) are set to 0.

tolist()

Return an ordinary list with the items in the bitarray. Note: To extend a bitarray with elements from a list, use the extend method.

tostring()

Return the string representing (machine values) of the bitarray. When the length of the bitarray is not a multiple of 8, the few remaining bits (1..7) are set to 0.

unpack(*zero*=`'\x00'`, *one*=`'\xff'`)

Return a string containing one character for each bit in the bitarray, using the specified mapping. Note that unpack('0', '1') has the same effect as to01(). See also the pack method.

9.2.2 Properties

Name	Description
size	Property that returns the chromosome size. Not writable. **Value:** `<property object at 0x8c1cd4c>`
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

9.2.3 Instance Variables

Name	Description
format	Property that contains the chromosome `struct` format.

10 Module peach.ga.crossover

Basic definitions for crossover operations and base classes.

Crossover is a very basic and important operation in genetic algorithms. It is by means of crossover among the chromosomes that population gains diversity, thus exploring more completelly the solution space and giving better answers. This sub-module provides definitions of the most common crossover operations, and provides a class that can be subclassed to construct different types of crossover for experimentation.

10.1 Variables

Name	Description
__doc__	**Value:** ...

10.2 Class Crossover

object
└
peach.ga.crossover.Crossover

Known Subclasses: peach.ga.crossover.OnePoint, peach.ga.crossover.TwoPoint, peach.ga.crossover.Uniform

Base class for crossover operators.

This class should be subclassed if you want to create your own crossover operator. The base class doesn't do much, it is only a prototype. As is done with all the base classes within this library, use the `__init__` method to configure your crossover behaviour -- if needed -- and the `__call__` method to operate over a population.

A class derived from this one should implement at least 2 methods, defined below:

__init__(self, *cnf, **kw) Initializes the object. There is no mandatory arguments, but any parameters can be used here to configure the operator. For example, a class can define a crossover rate -- this should be defined here:

```
__init__(self, rate=0.75)
```

A default value should always be offered, if possible.

__call__(self, population) The `__call__` implementation should receive a population and operate over it. Please, consult the `ga` module to see more information on populations. It should return the processed population. No recomendation on the internals of the method is made. That being said, in general the crossover operators pairs chromosomes and swap bits among them (but there is nothing to say that you can't do it differently).

Please, note that the GA implementations relies on this behaviour: it will pass a population to your `__call__` method and expects to received the result back.

10.2.1 Methods

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__init__(...)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

__new__(*T*, *S*, ...)

Return Value

```
a new object with type S, a subtype of T
```

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

10.2.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

10.3 Class OnePoint

object

peach.ga.crossover.Crossover

peach.ga.crossover.OnePoint

A one-point crossover operator.

A one-point crossover randomly selects a single point in two chromosomes and swaps the bits among them from that point until the end of the bit stream. The crossover rate is the probability that two paired chromosomes will exchange bits.

10.3.1 Methods

__init__(*self*, *rate*=`0.75`)

Initialize the crossover operator.

Parameters

`rate`: Probability that two paired chromosomes will exchange bits.

Overrides: object.__init__

__call__(*self*, *population*)

Proceeds the crossover over a population.
In one-point crossover, chromosomes from a population are randomly paired. If a uniform random number is below the `rate` given in the instantiation of the operator, then a random point is selected and bits from that point until the end of the chromosomes are exchanged.

Parameters

`population`: A list of `Chromosomes` containing the present population of the algorithm. It is processed and the results of the exchange are returned to the caller.

Return Value

The processed population, a list of `Chromosomes`.

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

10.3.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

10.3.3 Instance Variables

Name	Description
rate	Property that contains the crossover rate.

10.4 Class TwoPoint

object

peach.ga.crossover.Crossover

peach.ga.crossover.TwoPoint

A two-point crossover operator.

A two-point crossover randomly selects two points in two chromosomes and swaps the bits among them between these points. The crossover rate is the probability that two paired chromosomes will exchange bits.

10.4.1 Methods

__init__(*self*, *rate*=`0.75`)

Initialize the crossover operator.

Parameters

`rate:` Probability that two paired chromosomes will exchange bits.

Overrides: object.__init__

__call__(*self*, *population*)

Proceeds the crossover over a population.
In two-point crossover, chromosomes from a population are randomly paired. If a uniform random number is below the `rate` given in the instantiation of the operator, then random points are selected and bits between those points are exchanged.

Parameters

`population:` A list of `Chromosomes` containing the present population of the algorithm. It is processed and the results of the exchange are returned to the caller.

Return Value

The processed population, a list of `Chromosomes`.

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

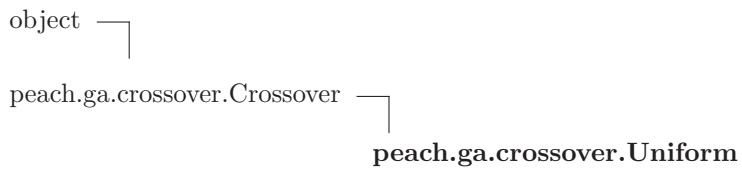
10.4.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

10.4.3 Instance Variables

Name	Description
rate	Property that contains the crossover rate.

10.5 Class Uniform



A uniform crossover operator.

A uniform crossover scans two chromosomes in a bit-to-bit fashion. According to a given crossover rate, the corresponding bits are exchanged. The crossover rate is the probability that two bits will be exchanged.

10.5.1 Methods

__init__(*self*, *rate*=`0.75`)

Initialize the crossover operator.

Parameters

`rate`: Probability that bits from two paired chromosomes will be exchanged.

Overrides: object.__init__

__call__(*self*, *population*)

Proceeds the crossover over a population.
In uniform crossover, chromosomes from a population are randomly paired, and scaned in a bit-to-bit fashion. If a uniform random number is below the `rate` given in the instantiation of the operator, then the bits under scan will be exchanged in the chromosomes.

Parameters

`population`: A list of `Chromosomes` containing the present population of the algorithm. It is processed and the results of the exchange are returned to the caller.

Return Value

The processed population, a list of `Chromosomes`.

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)
Return Value

```
a new object with type S, a subtype of T
```

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

10.5.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

10.5.3 Instance Variables

Name	Description
rate	Property that contains the crossover rate.

11 Module peach.ga.fitness

Basic definitions and base classes for definition of fitness functions for use with genetic algorithms.

Fitness is a function that rates higher the chromosomes that perform better according to the objective function. For example, if the minimum of a function needs to be found, then the fitness function should rate better the chromosomes that correspond to lower values of the objective function. This module gives support to use common Python functions as fitness functions in genetic algorithms.

The classes defined in this sub-module take a function and use some algorithm to rank a population. There are some different ranking functions, some are provided in this module. There is also a class that can be subclassed to generate other fitness methods. See the documentation of the corresponding class for more information.

11.1 Variables

Name	Description
__doc__	**Value:** ...

11.2 Class Fitness

object
└
peach.ga.fitness.Fitness

Known Subclasses: peach.ga.fitness.Ranking

Base class for fitness function classifiers.

This class is used as the base of all fitness functions. However, even if it is intended to be used as a base class, it also provides some functionality, described below.

A subclass of this class should implement at least 2 methods:

__init__(self, f) Initialization method. The initialization procedure should take at least one parameter, `f`, which is the function to be maximized. Other parameters could be used, but sensible default values should be available.

__call__(self, population) This method is called to calculate population fitness. There is no recomendation about the internals of the method, but its signature is expected as defined above. This method receives a population -- please, consult the `ga` module for more information on populations -- and should return a vector or list with the fitness value for each chromosome in the same order that they appear in the population.

This class implements the standard normalization fitness, as described in every book and article about GAs. The rank given to a chromosome is proportional to its objective function value.

11.2.1 Methods

__init__(*self*, *f*)

Initializes the operator.

Parameters

`f:` The cost function to be maximized. If you need to minimize a function, negate the return value.

Overrides: object.__init__

__call__(*self*, *population*)

Calculates the fitness for all individuals in the population.

Parameters

`population:` The population to be processed. Please, consult the `ga` module for more information on populations. This method calculates the fitness according to the traditional normalization technique.

Return Value

A vector containing the fitness value for every individual in the population, in the same order that they appear there.

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

11.2.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

11.2.3 Instance Variables

Name	Description
f	Objective function to be maximized. Handle with care -- although it can be changed, it might cause trouble if you do so.

11.3 Class Ranking

object
peach.ga.fitness.Fitness
peach.ga.fitness.Ranking

Ranking fitness for a population

Ranking gives fitness values equally spaced between 0 and 1. The fittest individual receives fitness equals to 1, the second best equals to 1 - 1/N, the third best 1 - 2/N, and so on, where N is the size of the population. It is important to note that the worst fit individual receives a fitness value of 1/N, not 0. That allows that no individuals are excluded from the selection operator.

11.3.1 Methods

__init__(*self*, *f*)

Initializes the operator.

Parameters

`f:` The cost function to be maximized. If you need to minimize a function, negate the return value.

Overrides: peach.ga.fitness.Fitness.__init__

__call__(*self*, *population*)

Calculates the fitness for all individuals in the population.

Parameters

`population`: The population to be processed. Please, consult the `ga` module for more information on populations. This method calculates the fitness according to the equally spaced ranking technique.

Return Value

A vector containing the fitness value for every individual in the population, in the same order that they appear there.

Overrides: peach.ga.fitness.Fitness.__call__

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

11.3.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

11.3.3 Instance Variables

Name	Description
f	Objective function to be maximized. Handle with care -- although it can be changed, it might cause trouble if you do so.

12 Module peach.ga.ga

Basic Genetic Algorithm (GA)

This sub-package implements a traditional genetic algorithm as described in books and papers. It consists of selecting, breeding and mutating a population of chromosomes (arrays of bits) and reinserting the fittest individual from the previous generation if the GA is elitist. Please, consult a good reference on the subject, for the subject is way too complicated to be explained here.

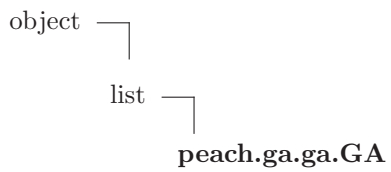
Within the algorithm implemented here, it is possible to specify and configure the selection, crossover and mutation methods using the classes in the respective sub-modules and custom methods can be implemented (check `selection`, `crossover` and `mutation` modules).

A GA object is actually a list of chromosomes. Please, refer to the documentation of the class below for more information.

12.1 Variables

Name	Description
__doc__	**Value:** ...
add	**Value:** `<ufunc 'add'>`

12.2 Class GA



A standard Genetic Algorithm

This class implements the methods to generate, initialize and evolve a population of chromosomes according to a given fitness function. A standard GA implements, in this order:

- A selection method, to choose, from this generation, which individuals will be present in the next generation;
- A crossover method, to exchange information between selected individuals to add diversity to the population;
- A mutation method, to change information in a selected individual, also to add diversity to the population;
- The reinsertion of the fittest individual, if the population is elitist (which is almost always the case).

A population is actually a list of chromosomes, and individuals can be read and set as in a normal list. Use the `[ ]` operators to access individual chromosomes but please be aware that modifying the information on the list before the end of convergence can cause unpredictable results. The population and the algorithm have also other properties, check below to see more information on them.

12.2.1 Methods

__init__(*self*, *fitness*, *fmt*, *ranges*=[], *size*=50, *selection*=<class 'peach.ga.selection.RouletteWheel'>, *crossover*=<class 'peach.ga.crossover.TwoPoint'>, *mutation*=<class 'peach.ga.mutation.BitToBit'>, *elitist*=True)

Initializes the population and the algorithm.
On the initialization of the population, a lot of parameters can be set. Those will deeply affect the results. The parameters are:

Parameters

`fitness`: A fitness function to serve as an objective function. In general, a GA is used for maximizing a function. This parameter can be a standard Python function or a `Fitness` instance.
In the first case, the GA will convert the function in a `Fitness` instance and call it internally when needed. The function should receive a tuple or vector of data according to the given `Chromosome` format (see below) and return a numeric value.
In the second case, you can use any of the fitness methods of the `fitness` sub-module, or create your own. If you want to use your own fitness method (for experimentation or simulation, for example), it must be an instance of a `Fitness` or of a subclass, or an exception will be raised. Please consult the documentation on the `fitness` sub-module.

`fmt`: A `struct`-format string. The `struct` module is a standard Python module that packs and unpacks informations in bits. These are used to inform the algorithm what types of data are to be used. For example, if you are maximizing a function of three real variables, the format should be something like `"fff"`. Any type supported by the `struct` module can be used. The GA will decode the bit array according to this format and send it as is to your fitness function -- your function *must* know what to do with them.
Alternatively, the format can be an integer. In that case, the GA will not try to decode the bit sequence. Instead, the bits are passed without modification to the objective function, which must deal with them. Notice that, if this is used this way, the `ranges` property (see below) makes no sense, so it is set to `None`. Also, no sanity checks will be performed.

`ranges`: Since messing with the bits can change substantially the values obtained can diverge a lot from the maximum point. To avoid this, you can specify a range for each of the variables. `range` defaults to `[ ]`, this means that no range checkin will be done. If given, then every variable will be checked. There are two ways to specify the ranges.
It might be a tuple of two values, `(x0, x1)`, where `x0` is the start of the interval, and `x1` its end. Obviously, `x0` should be smaller than `x1`. If `range` is given in this way, then this range will be used for every variable.
If can be specified as a list of tuples with the same format as given above. In that case, the list must have one range for every variable specified in the format and the ranges must appear in the same order as there. That is, every variable must have a range associated to it.

`size`: This is the size of the population. It defaults to 50.

`selection`: This specifies the selection method. You can use one given in the `selection` sub-module, or you can implement your own. In any case, the `selection` parameter must be an instance of `Selection` or of a subclass. Please, see the documentation on the `selection` module for more information. Defaults to `RouletteWheel`. If made `None`, then selection will not be present in the GA.

`crossover`: This specifies the crossover method. You can use one given in the `crossover` sub-module, or you can implement your own. In any case, the `crossover` parameter must be an instance of `Crossover` or of a subclass. Please, see the documentation on the `crossover` module for more information. Defaults to `TwoPoint`. If made `None`, then crossover will not be present in the GA.

_get_csize(*self*)

sanity(*self*)

Sanitizes the chromosomes in the population.
Since not every individual generated by the crossover and mutation operations might be a valid result, this method verifies if they are inside the allowed ranges (or if it is a number at all). Each invalid individual is discarded and a new one is generated.
This method has no parameters and returns no values.

fit(*self*)

Computes the fitness for each individual of the population.
This method is only an interface to the `fitness` function passed in the initialization. It calls the `Fitness` instance.
This method has no parameters and returns no values.

step(*self*)

Computes a new generation of the population, a step of the adaptation.
This method goes through all the steps of the GA, as described above. If the selection, crossover and mutation operators are defined, they are applied over the population. If the population is elitist, then the fittest individual of the past generation is reinserted.
This method has no parameters and returns no values. The GA itself can be consulted (using `[ ]`) to find the fittest individual which is the result of the process.

__add__(*x*, *y*)

x+y

__contains__(*x*, *y*)

y in x

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__delitem__(*x*, *y*)

del x[y]

__delslice__(*x*, *i*, *j*)

del x[i:j]
Use of negative indices is not supported.

__eq__(*x*, *y*)

x==y

__ge__(*x*, *y*)

x>=y

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

Overrides: object.__getattribute__

__getitem__(*x*, *y*)

x[y]

__getslice__(*x*, *i*, *j*)

x[i:j]
Use of negative indices is not supported.

__gt__(*x*, *y*)

x>y

__hash__(*x*)

hash(x)

Overrides: object.__hash__

__iadd__(*x*, *y*)

x+=y

__imul__(*x*, *y*)

x*=y

__iter__(*x*)

iter(x)

__le__(*x*, *y*)

x<=y

__len__(*x*)

len(x)

__lt__(*x*, *y*)

x<y

__mul__(*x*, *n*)

x*n

__ne__(*x*, *y*)

x!=y

__new__(*T*, *S*, *...*)

Return Value

```
a new object with type S, a subtype of T
```

Overrides: object.__new__

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

Overrides: object.__repr__

__reversed__(*L*)

return a reverse iterator over the list

__rmul__(*x*, *n*)

n*x

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__setitem__(*x*, *i*, *y*)

x[i]=y

__setslice__(*x*, *i*, *j*, *y*)

x[i:j]=y
Use of negative indices is not supported.

__str__(*x*)

str(x)

append(*L*, *object*)

append object to end

count(*L*, *value*)

return number of occurrences of value

Return Value

`integer`

extend(*L*, *iterable*)

extend list by appending elements from the iterable

index(*...*)

L.index(value, [start, [stop]]) -> integer -- return first index of value

insert(*L*, *index*, *object*)

insert object before index

pop(*L*, *index*=...)

remove and return item at index (default last)

Return Value

`item`

remove(*L*, *value*)

remove first occurrence of value

reverse(*L*)

reverse *IN PLACE*

sort(*L*, *cmp*=`None`, *key*=`None`, *reverse*=`False`)

stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

12.2.2 Properties

Name	Description
chromosome_size	This property hold the chromosome size for the population. Not writable. **Value:** `<property object at 0x8c3e5f4>`
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

12.2.3 Instance Variables

Name	Description
ranges	Holds the ranges for every variable. Although it is a writable property, care should be taken in changing parameters before ending the convergence.
elitist	If `True`, then the population is elitist.
fitness	Vector containing the computed fitness value for every individual.

13 Module peach.ga.mutation

Basic definitions and classes for operating mutation on chromosomes.

The mutation operator changes selected bits in the array corresponding to the chromosome. This operation is not as common as the others, but some genetic algorithms still implement it.

13.1 Variables

Name	Description
__doc__	**Value:** ...

13.2 Class Mutation

object
└
peach.ga.mutation.Mutation

Known Subclasses: peach.ga.mutation.BitToBit

Base class for mutation operators.

This class should be subclassed if you want to create your own mutation operator. The base class doesn't do much, it is only a prototype. As is done with all the base classes within this library, use the `__init__` method to configure your mutation behaviour -- if needed -- and the `__call__` method to operate over a population.

A class derived from this one should implement at least 2 methods, defined below:

__init__(self, *cnf, **kw) Initializes the object. There is no mandatory arguments, but any parameters can be used here to configure the operator. For example, a class can define a mutation rate -- this should be defined here:

```
__init__(self, rate=0.75)
```

A default value should always be offered, if possible.

__call__(self, population) The `__call__` implementation should receive a population and operate over it. Please, consult the `ga` module to see more information on populations. It should return the processed population. No recomendation on the internals of the method is made.

Please, note that the GA implementations relies on this behaviour: it will pass a population to your `__call__` method and expects to received the result back.

13.2.1 Methods

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__init__(*...*)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

__new__(*T*, *S*, *...*)
Return Value
`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

13.2.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

13.3 Class BitToBit

object
peach.ga.mutation.Mutation
peach.ga.mutation.BitToBit

A simple bit-to-bit mutation operator.

This operator scans every individual in the population, in a bit-to-bit fashion. If a uniformly random number is less than the mutation rate (see below), then the bit is inverted. The mutation should be

made very small, since large populations will represent a big number of bits; it should never be more than 0.5.

13.3.1 Methods

__init__(*self*, *rate*=`0.05`)

Initialize the mutation operator.

Parameters

`rate:` Probability that a single bit in an individual will be inverted.

Overrides: object.__init__

__call__(*self*, *population*)

Applies the operator over a population.
The behaviour of this operator is as described above: it scans every bit in every individual, and if a random number is less than the mutation rate, the bit is inverted.

Parameters

`population:` A list of `Chromosomes` containing the present population of the algorithm. It is processed and the results of the exchange are returned to the caller.

Return Value

The processed population, a list of `Chromosomes`.

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

13.3.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

13.3.3 Instance Variables

Name	Description
rate	Property that contains the mutation rate.

14 Module peach.ga.selection

Basic classes and definitions for selection operator.

The first step in a genetic algorithm is the selection of the fittest individuals. The selection method typically uses the fitness of the population to compute which individuals are closer to the best solution. However, instead of deterministically deciding which individuals continue to the next generation, they are randomly choosen, the chances of an individual being choosen given by its fitness value. This sub-module implements selection methods.

14.1 Variables

Name	Description
__doc__	**Value:** ...

14.2 Class Selection

object
└
peach.ga.selection.Selection

Known Subclasses: peach.ga.selection.Baker, peach.ga.selection.BinaryTournament, peach.ga.selection.RouletteWheel

Base class for selection operators.

This class should be subclassed if you want to create your own selection operator. The base class doesn't do much, it is only a prototype. As is done with all the base classes within this library, use the `__init__` method to configure your selection behaviour -- if needed -- and the `__call__` method to operate over a population.

A class derived from this one should implement at least 2 methods, defined below:

__init__(self, *cnf, **kw) Initializes the object. There is no mandatory arguments, but any parameters can be used here to configure the operator. A default value should always be offered, if possible.

__call__(self, population) The `__call__` implementation should receive a population and operate over it. Please, consult the `ga` module to see more information on populations. It should return the processed population. No recomendation on the internals of the method is made.

Please, note that the GA implementations relies on this behaviour: it will pass a population to your `__call__` method and expects to received the result back.

14.2.1 Methods

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__init__(*...*)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

__new__(*T*, *S*, *...*)
Return Value
`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

14.2.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

14.3 Class RouletteWheel

object
peach.ga.selection.Selection
peach.ga.selection.RouletteWheel

The Roulette Wheel selection method.

This method randomly chooses a new population with the same size of the original population. An individual is choosen with a probability proportional to its fitness value, independent of what fitness method was used. This is usually abstracted as a roulette wheel in texts about the subject. Please,

note that the selection is done *in loco*, that is, although the new population is returned, it is not a new list -- it is the same list as before, but with values changed.

14.3.1 Methods

__call__(*self*, *population*)

Selects the population.

Parameters

`population:` The list of chromosomes that should be operated over. The given list is modified, so be aware that the old generation will not be available after stepping the GA.

Return Value

The new population.

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__init__(...)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

14.3.2 Properties

Name	Description
__class__	**Value:** <attribute '__class__' of 'object' objects>

14.4 Class BinaryTournament

object

peach.ga.selection.Selection

peach.ga.selection.BinaryTournament

The Binary Tournament selection method.

This method randomly chooses a new population with the same size of the original population. Two individuals are choosen at random and they "battle", the fittest surviving for the next generation. Please, note that the selection is done *in loco*, that is, although the new population is returned, it is not a new list -- it is the same list as before, but with values changed.

14.4.1 Methods

__call__(*self*, *population*)

Selects the population.

Parameters

`population`: The list of chromosomes that should be operated over. The given list is modified, so be aware that the old generation will not be available after stepping the GA.

Return Value

The new population.

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__init__(...)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

__new__(*T*, *S*, ...)
Return Value
`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

14.4.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

14.5 Class Baker

object
peach.ga.selection.Selection
peach.ga.selection.Baker

The Baker selection method.

This method is very similar to the Roulette Wheel, but instead or randomly choosing every new member on the next generation, only the first probability is randomized. The others are determined as equally spaced numbers from 0 to 1, from this number. Please, note that the selection is done *in loco*, that is, although the new population is returned, it is not a new list -- it is the same list as before, but with values changed.

14.5.1 Methods

__call__(*self*, *population*)

Selects the population.

Parameters

`population`: The list of chromosomes that should be operated over. The given list is modified, so be aware that the old generation will not be available after stepping the GA.

Return Value

The new population.

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__init__(...)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

14.5.2 Properties

Name	Description
__class__	**Value:** <attribute '__class__' of 'object' objects>

15 Package peach.nn

This package implements support for neural networks. Consult:

af A list of activation functions for use with neurons and a base class to implement different activation functions;

base Basic definitions of the objects used with neural networks;

lrule Learning rules;

nn Implementation of different classes of neural networks;

15.1 Modules

- **af**: Base activation functions and base class
(Section 16, p. 109)
- **base**: Basic definitions for layers of neurons.
(Section 17, p. 131)
- **lrules**: Learning rules for neural networks and base classes for custom learning.
(Section 18, p. 135)
- **nn**: Basic topologies of neural networks.
(Section 19, p. 153)

15.2 Variables

Name	Description
__doc__	**Value:** ...

16 Module peach.nn.af

Base activation functions and base class

Activation functions define if a neuron is activated or not. There are a lot of different definitions for activation functions in the literature, and this sub-package implements some of them. An activation function is defined by its response and its derivative. Being conveniently defined as classes, it is possible to define a custom derivative method.

In this package, also, there is a base class that should be subclassed if you want to define your own activation function. This class, however, can be instantiated with a standard Python function as an initialization parameter, and it is adjusted to work with the internals of the package.

If the base class is instantiated, then the function should take a real number as input, and return a real number. The response of the function determines if the neuron is activated or not.

16.1 Variables

Name	Description
__doc__	**Value:** ...

16.2 Class Activation

object
└
peach.nn.af.Activation

Known Subclasses: peach.nn.af.ArcTan, peach.nn.af.Linear, peach.nn.af.Sigmoid, peach.nn.af.Ramp, peach.nn.af.Signum, peach.nn.af.Threshold, peach.nn.af.TanH

Base class for activation functions.

This class can be used as base for activation functions. A subclass should have at least three methods, described below:

__init__ This method should be used to configure the function. In general, some parameters to change the behaviour of a simple function is passed. In a subclass, the `__init__` method should call the mother class initialization procedure.

__call__ The `__call__` interface is the function call. It should receive a *vector* of real numbers and return a *vector* of real numbers. Using the capabilities of the `numpy` module will help a lot. In case you don't know how to use, maybe instantiating this class instead will work better (see below).

derivative This method implements the derivative of the activation function. It is used in the learning methods. If one is not provided (but remember to call the superclass `__init__` so that it is created).

16.2.1 Methods

__init__(*self*, *f*=`None`, *df*=`None`)

Initializes the activation function.
Instantiating this class creates and adjusts a standard Python function to work with layers of neurons.

Parameters

`f:` The activation function. It can be created as a lambda function or any other method, but it should take a real value, corresponding to the activation potential of a neuron, and return a real value, corresponding to its activation. Defaults to `None`, if none is given, the identity function is used.

`df:` The derivative of the above function. It can be defined as above, or not given. If not given, an estimate is calculated based on the given function. Defaults to `None`.

Overrides: object.__init__

__call__(*self*, *x*)

Call interface to the object.
This method applies the activation function over a vector of activation potentials, and returns the results.

Parameters

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

derivative(*self*, *x*, *dx*=`5e-05`)

An estimate of the derivative of the activation function.
This method estimates the derivative using difference equations. This is a simple estimate, but efficient nonetheless.

Parameters

`x:` A real number or vector of real numbers representing the point over which the derivative is to be calculated.

`dx:` The value of the interval of the estimate. The smaller this number is, the better. However, if made too small, the precision is not enough to avoid errors. This defaults to 5e-5, which is the values that gives the best results.

Return Value

The value of the derivative over the given point.

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)
Return Value
`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

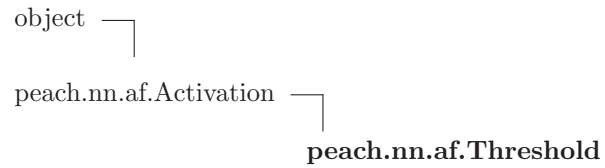
16.2.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

16.2.3 Instance Variables

Name	Description
d	An alias to the derivative of the function.

16.3 Class Threshold



Threshold activation function.

16.3.1 Methods

__init__(*self*, *threshold*=`0.0`, *amplitude*=`1.0`)

Initializes the object.

Parameters

`threshold`: The threshold value. If the value of the input is lower than this, the function is 0, otherwise, it is the given `amplitude`.

`amplitude`: The maximum value of the function.

Overrides: peach.nn.af.Activation.__init__

__call__(*self*, *x*)

Call interface to the object.
This method applies the activation function over a vector of activation potentials, and returns the results.

Parameters

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative. Technically, this function doesn't have a derivative, but making it equals to 1, this can be used in learning algorithms.

Parameters

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

16.3.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

16.3.3 Instance Variables

Name	Description
d	An alias to the derivative of the function.

16.4 Class Threshold

object

peach.nn.af.Activation

peach.nn.af.Threshold

Threshold activation function.

16.4.1 Methods

__init__(*self*, *threshold*=`0.0`, *amplitude*=`1.0`)

Initializes the object.

Parameters

`threshold`: The threshold value. If the value of the input is lower than this, the function is 0, otherwise, it is the given `amplitude`.

`amplitude`: The maximum value of the function.

Overrides: peach.nn.af.Activation.__init__

__call__(*self*, *x*)

Call interface to the object.
This method applies the activation function over a vector of activation potentials, and returns the results.

Parameters

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative. Technically, this function doesn't have a derivative, but making it equals to 1, this can be used in learning algorithms.

Parameters

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

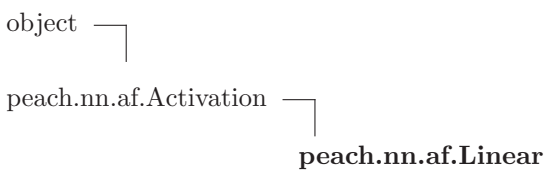
16.4.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

16.4.3 Instance Variables

Name	Description
d	An alias to the derivative of the function.

16.5 Class Linear



Identity activation function

16.5.1 Methods

__init__(*self*)

Initializes the function

Overrides: peach.nn.af.Activation.__init__

__call__(*self*, *x*)

Call interface to the object.
This method applies the activation function over a vector of activation potentials, and returns the results.

Parameters

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative.

Parameters

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

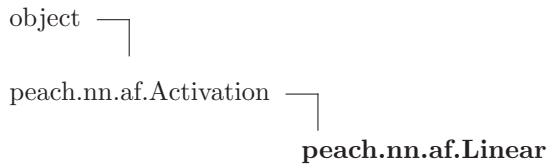
16.5.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

16.5.3 Instance Variables

Name	Description
d	An alias to the derivative of the function.

16.6 Class Linear



Identity activation function

16.6.1 Methods

__init__(*self*)

Initializes the function

Overrides: peach.nn.af.Activation.__init__

__call__(*self*, *x*)

Call interface to the object.
This method applies the activation function over a vector of activation potentials, and returns the results.

Parameters

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative.

Parameters

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

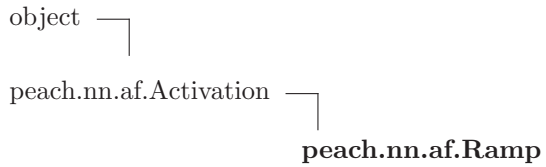
16.6.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

16.6.3 Instance Variables

Name	Description
d	An alias to the derivative of the function.

16.7 Class Ramp



Ramp activation function

16.7.1 Methods

__init__(*self*, *p0*=`(-0.5, 0.0)`, *p1*=`(0.5, 1.0)`)

Initializes the object.
Two points are needed to set this function. They are used to determine where the ramp begins and where it ends.

Parameters

`p0`: The starting point, given as a tuple `(x0, y0)`. For values of the input below `x0`, the function returns `y0`. Defaults to `(-0.5, 0.0)`.

`p1`: The ending point, given as a tuple `(x1, y1)`. For values of the input above `x1`, the function returns `y1`. Defaults to `(0.5, 1.0)`.

Overrides: peach.nn.af.Activation.__init__

__call__(*self*, *x*)

Call interface to the object.
This method applies the activation function over a vector of activation potentials, and returns the results.

Parameters

`x`: A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative.

Parameters

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

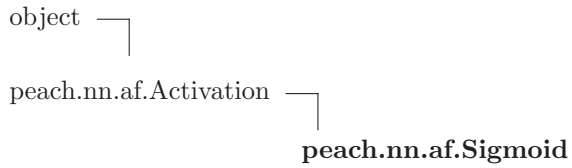
16.7.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

16.7.3 Instance Variables

Name	Description
d	An alias to the derivative of the function.

16.8 Class Sigmoid



Sigmoid activation function

16.8.1 Methods

__init__(*self*, *a*=`1.0`, *x0*=`0.0`)

Initializes the object.

Parameters

`a:` The slope of the function in the center `x0`. Defaults to 1.0.

`x0:` The center of the sigmoid. Defaults to 0.0.

Overrides: peach.nn.af.Activation.__init__

__call__(*self*, *x*)

Call interface to the object.
This method applies the activation function over a vector of activation potentials, and returns the results.

Parameters

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative.

Parameters

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

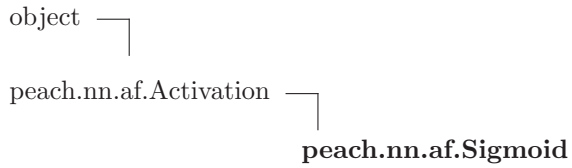
16.8.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

16.8.3 Instance Variables

Name	Description
d	An alias to the derivative of the function.

16.9 Class Sigmoid



Sigmoid activation function

16.9.1 Methods

__init__(*self*, *a*=`1.0`, *x0*=`0.0`)

Initializes the object.

Parameters

`a:` The slope of the function in the center `x0`. Defaults to 1.0.

`x0:` The center of the sigmoid. Defaults to 0.0.

Overrides: peach.nn.af.Activation.__init__

__call__(*self*, *x*)

Call interface to the object.
This method applies the activation function over a vector of activation potentials, and returns the results.

Parameters

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative.

Parameters

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

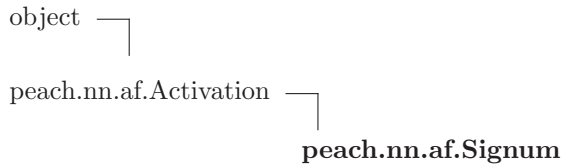
16.9.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

16.9.3 Instance Variables

Name	Description
d	An alias to the derivative of the function.

16.10 Class Signum



Signum activation function

16.10.1 Methods

__init__(*self*)

Initializes the object.

Overrides: peach.nn.af.Activation.__init__

__call__(*self*, *x*)

Call interface to the object.
This method applies the activation function over a vector of activation potentials, and returns the results.

Parameters

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative. Technically, this function doesn't have a derivative, but making it equals to 1, this can be used in learning algorithms.

Parameters

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)
Return Value
`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

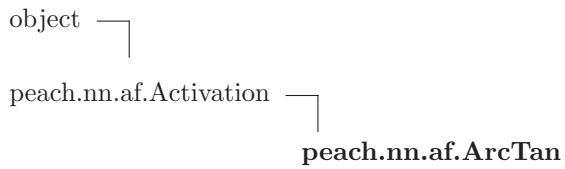
16.10.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

16.10.3 Instance Variables

Name	Description
d	An alias to the derivative of the function.

16.11 Class ArcTan



Inverse tangent activation function

16.11.1 Methods

__init__(*self*, *a*=`1.0`, *x0*=`0.0`)

Initializes the object

Parameters

`a:` The slope of the function in the center `x0`. Defaults to 1.0.

`x0:` The center of the sigmoid. Defaults to 0.0.

Overrides: peach.nn.af.Activation.__init__

__call__(*self*, *x*)

Call interface to the object.
This method applies the activation function over a vector of activation potentials, and returns the results.

Parameters

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative.

Parameters

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

```
a new object with type S, a subtype of T
```

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

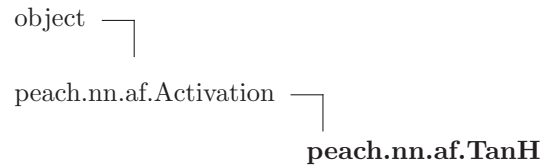
16.11.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

16.11.3 Instance Variables

Name	Description
d	An alias to the derivative of the function.

16.12 Class TanH



Hyperbolic tangent activation function

16.12.1 Methods

__init__(*self*, *a*=`1.0`, *x0*=`0.0`)

Initializes the object

Parameters

`a:` The slope of the function in the center `x0`. Defaults to 1.0.

`x0:` The center of the sigmoid. Defaults to 0.0.

Overrides: peach.nn.af.Activation.__init__

__call__(*self*, *x*)

Call interface to the object.
This method applies the activation function over a vector of activation potentials, and returns the results.

Parameters

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The activation function applied over the input vector.

Overrides: peach.nn.af.Activation.__call__

derivative(*self*, *x*)

The function derivative.

Parameters

`x:` A real number or a vector of real numbers representing the activation potential of a neuron or a layer of neurons.

Return Value

The derivative of the activation function applied over the input vector.

Overrides: peach.nn.af.Activation.derivative

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

16.12.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

16.12.3 Instance Variables

Name	Description
d	An alias to the derivative of the function.

17 Module peach.nn.base

Basic definitions for layers of neurons.

This subpackage implements the basic classes used with neural networks. A neural network is basically implemented as a layer of neurons. To speed things up, a layer is implemented as a array, where each line represents the weight vector of a neuron. Further definitions and algorithms are based on this definition.

17.1 Variables

Name	Description
__doc__	**Value:** ...
_BIAS	This constant vector is defined to implement in a fast way the bias of a neuron, as an input of value 1, stacked over the real input to the neuron. **Value:** `array([[ 1.]])`

17.2 Class Layer

object
└
peach.nn.base.Layer

Known Subclasses: peach.nn.nn.SOM

Base class for neural networks.

This class implements a layer of neurons. It is represented by a array of real values. Each line of the array represents the weight vector of a single neuron. If the neurons on the layer are biased, then the first element of the weight vector is the bias weight, and the bias input is always valued 1. Also, to each layer is associated an activation function, that determines if the neuron is fired or not. Please, consult the module `af` to see more about activation functions.

In general, this class shoulb be subclassed if you want to use neural nets. But, as neural nets are very different one from the other, check carefully the documentation to see if the attributes, properties and methods are suited to your task.

17.2.1 Methods

__call__(*self*, *x*)

The feedforward method to the layer.
The `__call__` interface should be called if the answer of the neuron to a given input vector `x` is desired. *This method has collateral effects*, so beware. After the calling of this method, the `v` and `y` properties are set with the activation potential and the answer of the neurons, respectivelly.

Parameters

`x`: The input vector to the layer.

Return Value

The vector containing the answer of every neuron in the layer, in the respective order.

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__getbias(*self*)

__getinputs(*self*)

__getitem__(*self*, *n*)

The `[ ]` get interface.
The input to this method is forwarded to the `weights` property. That means that it will return the respective line/element of the weight array.

Parameters

`n:` A slice object containing the elements referenced. Since it is forwarded to an array, it behaves exactly as one.

Return Value

The element or elements in the referenced indices.

__getphi(*self*)

__getshape(*self*)

__getsize(*self*)

__getv(*self*)

__getweights(*self*)

__gety(*self*)

__hash__(*x*)

hash(x)

__init__(*self*, *shape*, *phi*=<class 'peach.nn.af.Linear'>, *bias*=False)

Initializes the layer.
A layer is represented by a array where each line is the weight vector of a single neuron. The first element of the vector is the bias weight, in case the neuron is biased. Associated with the layer is an activation function defined in an appropriate way.

Parameters

`shape`: Stablishes the size of the layer. It must be a two-tuple of the format `(m, n)`, where `m` is the number of neurons in the layer, and `n` is the number of inputs of each neuron. The neurons in the layer all have the same number of inputs.

`phi`: The activation function. It can be an `Activation` object (please, consult the `af` module) or a standard Python function. In this case, it must receive a single real value and return a single real value which determines if the neuron is activated or not. Defaults to `Linear`.

`bias`: If `True`, then the neurons on the layer are biased. That means that an additional weight is added to each neuron to represent the bias. If `False`, no modification is made.

Overrides: object.__init__

__new__(*T*, *S*, *...*)
Return Value
`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__setitem__(*self*, *n*, *w*)

The `[ ]` set interface.
The inputs to this method are forwarded to the `weights` property. That means that it will set the respective line/element of the weight array.

Parameters

`n`: A slice object containing the elements referenced. Since it is forwarded to an array, it behaves exactly as one.

`w`: A value or array of values to be set in the given indices.

__setphi(*self*, *phi*)

__setweights__(*self*, *m*)

__str__(*x*)

str(x)

17.2.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`
bias	True if the neuron is biased. Not writable. **Value:** `<property object at 0x8cf98ec>`
inputs	Number of inputs for each neuron in the layer. Not writable. **Value:** `<property object at 0x8cf989c>`
phi	The activation function. It can be set with an `Activation` instance or a standard Python function. If a standard function is given, it must receive a real value and return a real value that is the activation value of the neuron. In that case, it is adjusted to work accordingly with the internals of the layer. **Value:** `<property object at 0x8cf993c>`
shape	Shape of the layer, given in the format of a tuple `(m, n)`, where `m` is the number of neurons in the layer, and `n` is the number of inputs in each neuron. Not writable. **Value:** `<property object at 0x8cf98c4>`
size	Number of neurons in the layer. Not writable. **Value:** `<property object at 0x8cf9874>`
v	The activation potential of the neuron. Not writable, and only available after the neuron is fed some input. **Value:** `<property object at 0x8cf9964>`
weights	A `numpy` array containing the synaptic weights of the network. Each line is the weight vector of a neuron. It is writable, but the new weight array must be the same shape of the neuron, or an exception is raised. **Value:** `<property object at 0x8cf9914>`
y	The activation value of the neuron. Not writable, and only available after the neuron is fed some input. **Value:** `<property object at 0x8cf998c>`

18 Module peach.nn.lrules

Learning rules for neural networks and base classes for custom learning.

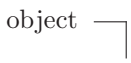
This sub-package implements learning methods commonly used with neural networks. There are a lot of different topologies and different learning methods for each one. It is very difficult to find a consistent framework for defining learning methods, in consequence. This method defines some base classes that are coupled with the neural networks that they are supposed to work with. Also, based on these classes, some of the traditional methods are implemented.

If you want to implement a different learning method, you must subclass the correct base class. Consult the classes below. Also, pay attention to how the implementation is expected to behave. Since learning algorithms are usually somewhat complex, care should be taken to make everything work accordingly.

18.1 Variables

Name	Description
__doc__	**Value:** ...
_BIAS	This constant vector is defined to implement in a fast way the bias of a neuron, as an input of value 1, stacked over the real input to the neuron. **Value:** `array([[ 1.]])`

18.2 Class FFLearning



peach.nn.lrules.FFLearning

Known Subclasses: peach.nn.lrules.BackPropagation, peach.nn.lrules.LMS

Base class for FeedForwarding Multilayer neural networks.

As a base class, this class doesn't do anything. You should subclass this class if you want to implement a learning method for multilayer networks.

A learning method for a neural net of this kind must deal with a `FeedForward` instance. A `FeedForward` object is a list of `Layers` (consulting the documentation of these classes is important!). Each layer is a bidimensional array, where each line represents the synaptic weights of a single neuron. So, a multilayer network is actually a three-dimensional array, if you will. Usually, though, learning methods for this kind of net propagate some measure of the error from the output back to the input (the `BackPropagation` method, for instance).

A class implementing a learning method should have at least two methods:

__init__ The `__init__` method should initialize the object. It is in general used to configure some property of the learning algorithm, such as the learning rate.

__call__ The `__call__` interface is how the method should interact with the neural network. It should have the following signature:

```
__call__(self, nn, x, d)
```

where `nn` is the `FeedForward` instance to be modified *in loco*, `x` is the input vector and `d` is the desired response of the net for that particular input vector. It should return nothing.

18.2.1 Methods

__call__(*self*, *nn*, *x*, *d*)

The `__call__` interface.
Read the documentation for this class for more information. A call to the class should have the following parameters:

Parameters

`nn`: A `FeedForward` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x`: The input vector from the training set.

`d`: The desired response for the given input vector.

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__init__(...)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

18.2.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

18.3 Class LMS

object

peach.nn.lrules.FFLearning

peach.nn.lrules.LMS

The Least-Mean-Square (LMS) learning method.

The LMS method is a very simple method of learning, thoroughly described in virtually every book about the subject. Please, consult a good book on neural networks for more information. This implementation tries to use the `numpy` routines as much as possible for better efficiency.

18.3.1 Methods

__init__(*self*, *lrate*=`0.05`)

Initializes the object.

Parameters

`lrate`: Learning rate to be used in the algorithm. Defaults to 0.05.

Overrides: object.__init__

__call__(*self*, *nn*, *x*, *d*)

The `__call__` interface.
The learning implementation. Read the documentation for the base class for more information. A call to the class should have the following parameters:

Parameters

`nn`: A `FeedForward` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x`: The input vector from the training set.

`d`: The desired response for the given input vector.

Overrides: peach.nn.lrules.FFLearning.__call__

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)
Return Value

```
a new object with type S, a subtype of T
```

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

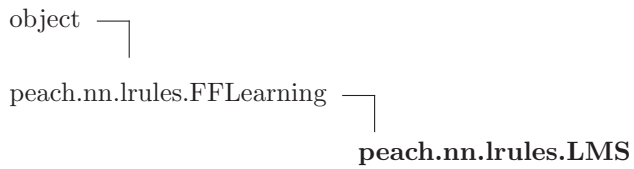
18.3.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

18.3.3 Instance Variables

Name	Description
lrate	Learning rate used in the algorithm.

18.4 Class LMS



The Least-Mean-Square (LMS) learning method.

The LMS method is a very simple method of learning, thoroughly described in virtually every book about the subject. Please, consult a good book on neural networks for more information. This implementation tries to use the `numpy` routines as much as possible for better efficiency.

18.4.1 Methods

__init__(*self*, *lrate*=`0.05`)

Initializes the object.

Parameters

`lrate`: Learning rate to be used in the algorithm. Defaults to 0.05.

Overrides: object.__init__

__call__(*self*, *nn*, *x*, *d*)

The `__call__` interface.
The learning implementation. Read the documentation for the base class for more information. A call to the class should have the following parameters:

Parameters

`nn`: A `FeedForward` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x`: The input vector from the training set.

`d`: The desired response for the given input vector.

Overrides: peach.nn.lrules.FFLearning.__call__

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)
Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

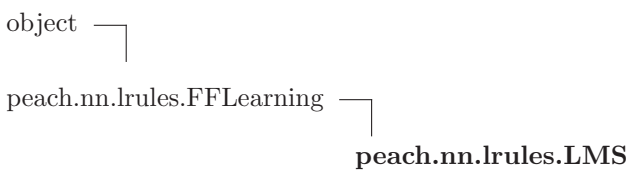
18.4.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

18.4.3 Instance Variables

Name	Description
lrate	Learning rate used in the algorithm.

18.5 Class LMS



The Least-Mean-Square (LMS) learning method.

The LMS method is a very simple method of learning, thoroughly described in virtually every book about the subject. Please, consult a good book on neural networks for more information. This implementation tries to use the `numpy` routines as much as possible for better efficiency.

18.5.1 Methods

__init__(*self*, *lrate*=`0.05`)

Initializes the object.

Parameters

`lrate:` Learning rate to be used in the algorithm. Defaults to 0.05.

Overrides: object.__init__

__call__(*self*, *nn*, *x*, *d*)

The `__call__` interface.
The learning implementation. Read the documentation for the base class for more information. A call to the class should have the following parameters:

Parameters

`nn:` A `FeedForward` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x:` The input vector from the training set.

`d:` The desired response for the given input vector.

Overrides: peach.nn.lrules.FFLearning.__call__

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)
Return Value

`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

18.5.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

18.5.3 Instance Variables

Name	Description
lrate	Learning rate used in the algorithm.

18.6 Class BackPropagation

object

peach.nn.lrules.FFLearning

peach.nn.lrules.BackPropagation

The BackPropagation learning method.

The backpropagation method is a very simple method of learning, thoroughly described in virtually every book about the subject. Please, consult a good book on neural networks for more information. This implementation tries to use the `numpy` routines as much as possible for better efficiency.

18.6.1 Methods

__init__(*self*, *lrate*=`0.05`)

Initializes the object.

Parameters

`lrate`: Learning rate to be used in the algorithm. Defaults to 0.05.

Overrides: object.__init__

__call__(*self*, *nn*, *x*, *d*)

The `__call__` interface.
The learning implementation. Read the documentation for the base class for more information. A call to the class should have the following parameters:

Parameters

`nn`: A `FeedForward` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x`: The input vector from the training set.

`d`: The desired response for the given input vector.

Overrides: peach.nn.lrules.FFLearning.__call__

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)
Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

18.6.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

18.6.3 Instance Variables

Name	Description
lrate	Learning rate used in the algorithm.

18.7 Class SOMLearning

object
└
peach.nn.lrules.SOMLearning

Known Subclasses: peach.nn.lrules.Competitive, peach.nn.lrules.Cooperative, peach.nn.lrules.WinnerTakesAll

Base class for Self-Organizing Maps.

As a base class, this class doesn't do anything. You should subclass this class if you want to implement a learning method for self-organizing maps.

A learning method for a neural net of this kind must deal with a `SOM` instance. A `SOM` object is a `Layer` (consulting the documentation of these classes is important!).

A class implementing a learning method should have at least two methods:

__init__ The `__init__` method should initialize the object. It is in general used to configure some property of the learning algorithm, such as the learning rate.

__call__ The `__call__` interface is how the method should interact with the neural network. It should have the following signature:

```
__call__(self, nn, x)
```

where `nn` is the `SOM` instance to be modified *in loco*, and `x` is the input vector. It should return nothing.

18.7.1 Methods

__call__(*self*, *nn*, *x*, *d*)

The `__call__` interface.
Read the documentation for this class for more information. A call to the class should have the following parameters:

Parameters

`nn`: A `SOM` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x`: The input vector from the training set.

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__init__(...)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

__new__(*T*, *S*, ...)

Return Value

```
a new object with type S, a subtype of T
```

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

18.7.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

18.8 Class WinnerTakesAll

object

peach.nn.lrules.SOMLearning

peach.nn.lrules.WinnerTakesAll

Purely competitive learning method without learning rate adjust.

A winner-takes-all strategy detects the winner on the self-organizing map and adjusts it in the direction of the input vector, scaled by the learning rate. Its tendency is to cluster around the gravity center of the points in the training set.

18.8.1 Methods

__init__(*self*, *lrate*=`0.05`)

Initializes the object.

Parameters

`lrate`: Learning rate to be used in the algorithm. Defaults to 0.05.

Overrides: object.__init__

__call__(*self*, *nn*, *x*)

The `__call__` interface.
The learning implementation. Read the documentation for the base class for more information. A call to the class should have the following parameters:

Parameters

`nn`: A `SOM` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x`: The input vector from the training set.

Overrides: peach.nn.lrules.SOMLearning.__call__

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

18.8.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

18.8.3 Instance Variables

Name	Description
lrate	Learning rate used with the algorithm.

18.9 Class WinnerTakesAll

object

peach.nn.lrules.SOMLearning

peach.nn.lrules.WinnerTakesAll

Purely competitive learning method without learning rate adjust.

A winner-takes-all strategy detects the winner on the self-organizing map and adjusts it in the direction of the input vector, scaled by the learning rate. Its tendency is to cluster around the gravity center of the points in the training set.

18.9.1 Methods

__init__(*self*, *lrate*=`0.05`)

Initializes the object.

Parameters

`lrate`: Learning rate to be used in the algorithm. Defaults to 0.05.

Overrides: object.__init__

__call__(*self*, *nn*, *x*)

The `__call__` interface.
The learning implementation. Read the documentation for the base class for more information. A call to the class should have the following parameters:

Parameters

`nn`: A `SOM` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x`: The input vector from the training set.

Overrides: peach.nn.lrules.SOMLearning.__call__

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)

Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

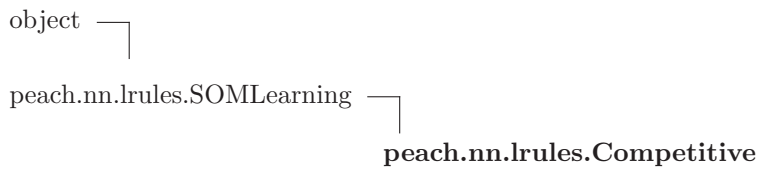
18.9.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

18.9.3 Instance Variables

Name	Description
lrate	Learning rate used with the algorithm.

18.10 Class Competitive



Competitive learning with time adjust of the learning rate.

A competitive strategy detects the winner on the self-organizing map and adjusts it in the direction of the input vector, scaled by the learning rate. Its tendency is to cluster around the gravity center of the points in the training set. As time passes, the learning rate grows smaller, this allows for better adjustment of the synaptic weights.

18.10.1 Methods

__init__(*self*, *lrate*=`0.05`, *tl*=`1000.0`)

Initializes the object.

Parameters

`lrate:` Learning rate to be used in the algorithm. Defaults to 0.05.

`tl:` Time constant that measures how many iterations will be needed to reduce the learning rate to a small value. Defaults to 1000.

Overrides: object.__init__

__call__(*self*, *nn*, *x*)

The `__call__` interface.
The learning implementation. Read the documentation for the base class for more information. A call to the class should have the following parameters:

Parameters

`nn:` A `SOM` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x:` The input vector from the training set.

Overrides: peach.nn.lrules.SOMLearning.__call__

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)
Return Value
`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

18.10.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

18.11 Class Cooperative

object
peach.nn.lrules.SOMLearning
peach.nn.lrules.Cooperative

Cooperative learning with time adjust of the learning rate and neighborhood function to propagate cooperation

A cooperative strategy detects the winner on the self-organizing map and adjusts it in the direction of the input vector, scaled by the learning rate. Its tendency is to cluster around the gravity center of

the points in the training set. As time passes, the learning rate grows smaller, this allows for better adjustment of the synaptic weights.

Also, a neighborhood is defined on the winner. Neurons close to the winner are also updated in the direction of the input vector, although with a smaller scale determined by the neighborhood function. A neighborhood function is 1. at 0., and decreases monotonically as the distance increases.

There are issues with this class! -- some of the class capabilities are yet to be developed.

18.11.1 Methods

__init__(*self*, *lrate*=`0.05`, *tl*=`1000`, *tn*=`1000`)

Initializes the object.

Parameters

`lrate`: Learning rate to be used in the algorithm. Defaults to 0.05.

`tl`: Time constant that measures how many iterations will be needed to reduce the learning rate to a small value. Defaults to 1000.

`tn`: Time constant that measures how many iterations will be needed to shrink the neighborhood. Defaults to 1000.

Overrides: object.__init__

__call__(*self*, *nn*, *x*)

The `__call__` interface.
The learning implementation. Read the documentation for the base class for more information. A call to the class should have the following parameters:

Parameters

`nn`: A `SOM` neural network instance that is going to be modified by the learning algorithm. The modification is made *in loco*, that is, the synaptic weights of `nn` should be modified in place, and not returned from this function.

`x`: The input vector from the training set.

Overrides: peach.nn.lrules.SOMLearning.__call__

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)
Return Value

`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

18.11.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

19 Module peach.nn.nn

Basic topologies of neural networks.

This sub-package implements various neural network topologies, see the complete list below. These topologies are implemented using the `Layer` class of the `base` sub-package. Please, consult the documentation of that module for more information on layers of neurons. The neural nets implemented here don't derive from the `Layer` class, instead, they have instance variables to take control of them. Thus, there is no base class for networks. While subclassing the classes of this module is usually safe, it is recomended that a new kind of net is developed from the ground up.

19.1 Functions

randn(*d0*, *d1*, *dn*, *...*)

Returns zero-mean, unit-variance Gaussian random numbers in an array of shape (d0, d1, ..., dn).
Note: This is a convenience function. If you want an interface that takes a tuple as the first argument use numpy.random.standard_normal(shape_tuple).

19.2 Variables

Name	Description
__doc__	**Value:** `...`
arctan	**Value:** `<ufunc 'arctan'>`
cosh	**Value:** `<ufunc 'cosh'>`
exp	**Value:** `<ufunc 'exp'>`
pi	**Value:** `3.14159265359`
tanh	**Value:** `<ufunc 'tanh'>`

19.3 Class FeedForward

Classic completely connected neural network.

A feedforward neural network is implemented as a list of layers, each layer being a `Layer` object (please consult the documentation on the `base` module for more information on layers). The layers are completely connected, which means that every neuron in one layers is connected to every other neuron in the following layer.

There is a number of learning methods that are already implemented, but in general, any learning class derived from `FFLearning` can be used. No other kind of learning can be used. Please, consult the documentation on the `lrules` (*learning rules*) module.

19.3.1 Methods

__init__(*self*, *layers*, *phi*=<class 'peach.nn.af.Linear'>, *lrule*=<class 'peach.nn.lrules.BackPropagation'>, *bias*=False)

Initializes a feedforward neural network.
A feedforward network is implemented as a list of layers, completely connected.

Parameters

`layers`: A list of integers containing the shape of the network. The first element of the list is the number of inputs of the network (or, as somebody prefer, the number of input neurons); the number of outputs is the number of neurons in the last layer. Thus, at least two numbers should be given.

`phi`: The activation functions to be used with each layer of the network. Please consult the `Layer` documentation in the `base` module for more information. This parameter can be a single function or a list of functions. If only one function is given, then the same function is used in every layer. If a list of functions is given, then the layers use the functions in the sequence given. Note that heterogeneous networks can be created that way. Defaults to `Linear`.

`lrule`: The learning rule used. Only `FFLearning` objects (instances of the class or of the subclasses) are allowed. Defaults to `BackPropagation`. Check the `lrules` documentation for more information.

`bias`: If `True`, then the neurons are biased.

Return Value

`new list`

Overrides: list.__init__

__getnlayers(*self*)

__getbias(*self*)

__gety(*self*)

__getphi(*self*)

__setphi(*self*, *phis*)

__call__(*self*, *x*)

The feedforward method of the network.
The `__call__` interface should be called if the answer of the neuron network to a given input vector `x` is desired. *This method has collateral effects*, so beware. After the calling of this method, the `y` property is set with the activation potential and the answer of the neurons, respectivelly.

Parameters

`x`: The input vector to the network.

Return Value

The vector containing the answer of every neuron in the last layer, in the respective order.

learn(*self*, *x*, *d*)

Applies one example of the training set to the network.
Using this method, one iteration of the learning procedure is made with the neurons of this network. This method presents one example (not necessarilly of a training set) and applies the learning rule over the network. The learning rule is defined in the initialization of the network, and some are implemented on the `lrules` method. New methods can be created, consult the `lrules` documentation but, for `FeedForward` instances, only `FFLearning` learning is allowed.
Also, notice that *this method only applies the learning method!* The network should be fed with the same input vector before trying to learn anything first. Consult the `feed` and `train` methods below for more ways to train a network.

Parameters

`x`: Input vector of the example. It should be a column vector of the correct dimension, that is, the number of input neurons.

`d`: The desired answer of the network for this particular input vector. Notice that the desired answer should have the same dimension of the last layer of the network. This means that a desired answer should be given for every output of the network.

Return Value

The error obtained by the network.

feed(*self*, *x*, *d*)

Feed the network and applies one example of the training set to the network.
Using this method, one iteration of the learning procedure is made with the neurons of this network. This method presents one example (not necessarilly of a training set) and applies the learning rule over the network. The learning rule is defined in the initialization of the network, and some are implemented on the `lrules` method. New methods can be created, consult the `lrules` documentation but, for `FeedForward` instances, only `FFLearning` learning is allowed.
Also, notice that *this method feeds the network* before applying the learning rule. Feeding the network has collateral effects, and some properties change when this happens. Namely, the `y` property is set. Please consult the `__call__` interface.

Parameters

`x`: Input vector of the example. It should be a column vector of the correct dimension, that is, the number of input neurons.

`d`: The desired answer of the network for this particular input vector. Notice that the desired answer should have the same dimension of the last layer of the network. This means that a desired answer should be given for every output of the network.

Return Value

The error obtained by the network.

train(*self*, *train_set*, *imax*=`2000`, *emax*=`1e-05`, *randomize*=`False`)

Presents a training set to the network.
This method automatizes the training of the network. Given a training set, the examples are shown to the network (possibly in a randomized way). A maximum number of iterations or a maximum admitted error should be given as a stop condition.

Parameters

`train_set`: The training set is a list of examples. It can have any size and can contain repeated examples. In fact, the definition of the training set is open. Each element of the training set, however, should be a two-tuple `(x, d)`, where `x` is the input vector, and `d` is the desired response of the network for this particular input. See the `learn` and `feed` for more information.

`imax`: The maximum number of iterations. Examples from the training set will be presented to the network while this limit is not reached. Defaults to 2000.

`emax`: The maximum admitted error. Examples from the training set will be presented to the network until the error obtained is lower than this limit. Defaults to 1e-5.

`randomize`: If this is `True`, then the examples are shown in a randomized order. If `False`, then the examples are shown in the same order that they appear in the `train_set` list. Defaults to `False`.

__add__(*x*, *y*)

x+y

__contains__(*x*, *y*)

y in x

__delattr__(...)

x.__delattr__('name') <==> del x.name

__delitem__(*x*, *y*)

del x[y]

__delslice__(*x*, *i*, *j*)

del x[i:j]
Use of negative indices is not supported.

__eq__(*x*, *y*)

x==y

__ge__(*x*, *y*)

x>=y

__getattribute__(...)

x.__getattribute__('name') <==> x.name

Overrides: object.__getattribute__

__getitem__(*x*, *y*)

x[y]

__getslice__(*x*, *i*, *j*)

x[i:j]
Use of negative indices is not supported.

__gt__(*x*, *y*)

x>y

__hash__(*x*)

hash(x)

Overrides: object.__hash__

__iadd__(*x*, *y*)

x+=y

__imul__(*x*, *y*)

x*=y

__iter__(*x*)

iter(x)

__le__(*x*, *y*)

x<=y

__len__(*x*)

len(x)

__lt__(*x*, *y*)

x<y

__mul__(*x*, *n*)

x*n

__ne__(*x*, *y*)

x!=y

__new__(*T*, *S*, *...*)

Return Value

```
a new object with type S, a subtype of T
```

Overrides: object.__new__

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

Overrides: object.__repr__

__reversed__(*L*)

return a reverse iterator over the list

__rmul__(*x*, *n*)

n*x

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__setitem__(*x*, *i*, *y*)

x[i]=y

__setslice__(*x*, *i*, *j*, *y*)

x[i:j]=y
Use of negative indices is not supported.

__str__(*x*)

str(x)

append(*L*, *object*)

append object to end

count(*L*, *value*)

return number of occurrences of value

Return Value
`integer`

extend(*L*, *iterable*)

extend list by appending elements from the iterable

index(*...*)

L.index(value, [start, [stop]]) -> integer -- return first index of value

insert(*L*, *index*, *object*)

insert object before index

pop(*L*, *index*=...)

remove and return item at index (default last)

Return Value
`item`

remove(*L*, *value*)

remove first occurrence of value

reverse(*L*)

reverse *IN PLACE*

sort(*L*, *cmp*=`None`, *key*=`None`, *reverse*=`False`)

stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

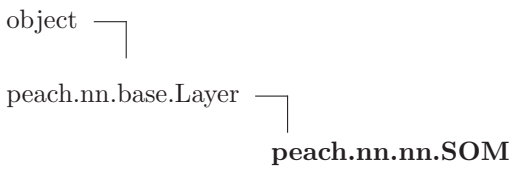
19.3.2 Properties

Name	Description
nlayers	Number of layers of the neural network. Not writable. **Value:** `<property object at 0x8d164dc>`
bias	A tuple containing the bias of each layer. Not writable. **Value:** `<property object at 0x8d16504>`
y	A list of activation values for each neuron in the last layer of the network, ie., the answer of the network. This property is available only after the network is fed some input. **Value:** `<property object at 0x8d1652c>`

continued on next page

Name	Description
phi	Activation functions for every layer in the network. It is a list of `Activation` objects, but can be set with only one function. In this case, the same function is used for every layer. **Value:** `<property object at 0x8d16554>`
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

19.4 Class SOM



A Self-Organizing Map (SOM).

A self-organizing map is a type of neural network that is trained via unsupervised learning. In particular, the self-organizing map finds the neuron closest to an input vector -- this neuron is the winning neuron, and it is the answer of the network. Thus, the SOM is usually used for classification and pattern recognition.

The SOM is a single-layer network, so this class subclasses the `Layer` class. But some of the properties of a `Layer` object are not available or make no sense in this context.

19.4.1 Methods

__init__(*self*, *shape*, *lrule*=`<class 'peach.nn.lrules.Competitive'>`)

Initializes a self-organizing map.
A self-organizing map is implemented as a layer of neurons. There is no connection among the neurons. The answer to a given input is the neuron closer to the given input. `phi` (the activation function) `v` (the activation potential) and `bias` are not used.

Parameters

`shape`: Stablishes the size of the SOM. It must be a two-tuple of the format `(m, n)`, where `m` is the number of neurons in the layer, and `n` is the number of inputs of each neuron. The neurons in the layer all have the same number of inputs.

`lrule`: The learning rule used. Only `SOMLearning` objects (instances of the class or of the subclasses) are allowed. Defaults to `Competitive`. Check the `lrules` documentation for more information.

Overrides: peach.nn.base.Layer.__init__

__gety__(*self*)

__call__(*self*, *x*)

The response of the network to a given input.
The `__call__` interface should be called if the answer of the neuron network to a given input vector `x` is desired. *This method has collateral effects*, so beware. After the calling of this method, the `y` property is set with the activation potential and the answer of the neurons, respectivelly.

Parameters

`x:` The input vector to the network.

Return Value

The winning neuron.

Overrides: peach.nn.base.Layer.__call__

learn(*self*, *x*)

Applies one example of the training set to the network.
Using this method, one iteration of the learning procedure is made with the neurons of this network. This method presents one example (not necessarilly of a training set) and applies the learning rule over the network. The learning rule is defined in the initialization of the network, and some are implemented on the `lrules` method. New methods can be created, consult the `lrules` documentation but, for `SOM` instances, only `SOMLearning` learning is allowed.
Also, notice that *this method only applies the learning method!* The network should be fed with the same input vector before trying to learn anything first. Consult the `feed` and `train` methods below for more ways to train a network.

Parameters

`x:` Input vector of the example. It should be a column vector of the correct dimension, that is, the number of input neurons.

Return Value

The error obtained by the network.

feed(*self*, *x*)

Feed the network and applies one example of the training set to the network.
Using this method, one iteration of the learning procedure is made with the neurons of this network. This method presents one example (not necessarilly of a training set) and applies the learning rule over the network. The learning rule is defined in the initialization of the network, and some are implemented on the `lrules` method. New methods can be created, consult the `lrules` documentation but, for `SOM` instances, only `SOMLearning` learning is allowed.
Also, notice that *this method feeds the network* before applying the learning rule. Feeding the network has collateral effects, and some properties change when this happens. Namely, the `y` property is set. Please consult the `__call__` interface.

Parameters

`x:` Input vector of the example. It should be a column vector of the correct dimension, that is, the number of input neurons.

Return Value

The error obtained by the network.

train(*self*, *train_set*, *imax*=`2000`, *emax*=`1e-05`, *randomize*=`False`)

Presents a training set to the network.
This method automatizes the training of the network. Given a training set, the examples are shown to the network (possibly in a randomized way). A maximum number of iterations or a maximum admitted error should be given as a stop condition.

Parameters

`train_set`: The training set is a list of examples. It can have any size and can contain repeated examples. In fact, the definition of the training set is open. Each element of the training set, however, should be a input vector of the correct dimensions, See the `learn` and `feed` for more information.

`imax`: The maximum number of iterations. Examples from the training set will be presented to the network while this limit is not reached. Defaults to 2000.

`emax`: The maximum admitted error. Examples from the training set will be presented to the network until the error obtained is lower than this limit. Defaults to 1e-5.

`randomize`: If this is `True`, then the examples are shown in a randomized order. If `False`, then the examples are shown in the same order that they appear in the `train_set` list. Defaults to `False`.

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__getitem__(*self*, *n*)

The `[ ]` get interface.
The input to this method is forwarded to the `weights` property. That means that it will return the respective line/element of the weight array.

Parameters

`n`: A slice object containing the elements referenced. Since it is forwarded to an array, it behaves exactly as one.

Return Value

The element or elements in the referenced indices.

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__setitem__(*self*, *n*, *w*)

The `[ ]` set interface.
The inputs to this method are forwarded to the `weights` property. That means that it will set the respective line/element of the weight array.

Parameters

`n:` A slice object containing the elements referenced. Since it is forwarded to an array, it behaves exactly as one.

`w:` A value or array of values to be set in the given indices.

__str__(*x*)

str(x)

19.4.2 Properties

Name	Description
y	The winning neuron for a given input, the answer of the network. This property is available only after the network is fed some input. **Value:** `<property object at 0x8d1661c>`
__class__	**Value:** `<attribute '__class__' of 'object' objects>`
bias	True if the neuron is biased. Not writable. **Value:** `<property object at 0x8cf98ec>`
inputs	Number of inputs for each neuron in the layer. Not writable. **Value:** `<property object at 0x8cf989c>`
phi	The activation function. It can be set with an `Activation` instance or a standard Python function. If a standard function is given, it must receive a real value and return a real value that is the activation value of the neuron. In that case, it is adjusted to work accordingly with the internals of the layer. **Value:** `<property object at 0x8cf993c>`
shape	Shape of the layer, given in the format of a tuple `(m, n)`, where `m` is the number of neurons in the layer, and `n` is the number of inputs in each neuron. Not writable. **Value:** `<property object at 0x8cf98c4>`

continued on next page

Name	Description
size	Number of neurons in the layer. Not writable. **Value:** `<property object at 0x8cf9874>`
v	The activation potential of the neuron. Not writable, and only available after the neuron is fed some input. **Value:** `<property object at 0x8cf9964>`
weights	A `numpy` array containing the synaptic weights of the network. Each line is the weight vector of a neuron. It is writable, but the new weight array must be the same shape of the neuron, or an exception is raised. **Value:** `<property object at 0x8cf9914>`

20 Package peach.optm

This package implements deterministic optimization methods. Consult:

optm Basic definitions and interface with the optimization methods;

linear Basic methods for one variable optimization;

multivar Gradient, Newton and othe multivariable optimization methods;

quasinewton Quasi-Newton methods;

stochastic General stochastic methods;

sa Simulated Annealing methods;

Every optimizer works in pretty much the same way. Instantiate the respective class, using as parameter the cost function to be optimized and some other parameters. Use `step()` to perform one iteration of the method, use the `__call__()` method to perform the search until the stop conditions are met. See each method for details.

20.1 Modules

- **linear**: This package implements basic one variable only optimizers.
  (Section 21, p. 166)
- **multivar**: This package implements basic multivariable optimizers, including gradient and Newton searches.
  (Section 22, p. 175)
- **optm**: Basic definitons and base class for optimizers
  (Section 23, p. 182)
- **quasinewton**: This package implements basic quasi-Newton optimizers.
  (Section 24, p. 186)
- **sa**: This package implements two versions of simulated annealing optimization.
  (Section 25, p. 194)
- **stochastic**: General methods of stochastic optimization.
  (Section 26, p. 201)

20.2 Variables

Name	Description
__doc__	**Value:** ...

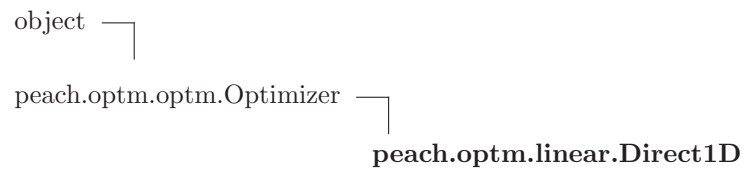
21 Module peach.optm.linear

This package implements basic one variable only optimizers.

21.1 Variables

Name	Description
__doc__	**Value:** ...

21.2 Class Direct1D



1-D direct search.

This methods 'oscilates' around the function minimum, reducing the updating step until it achieves the maximum error or the maximum number of steps. This is a very inefficient method, and should be used only at times where no other methods are able to converge (eg., if a function has a lot of discontinuities).

21.2.1 Methods

__init__(*self*, *f*, *dx*=`0.5`, *emax*=`1e-08`, *imax*=`1000`)

Initializes the optimizer.
To create an optimizer of this type, instantiate the class with the parameters given below:

Parameters

`f:` A one variable only function to be optimized. The function should have only one parameter and return the function value.

`dx:` The initial step of the search. Defaults to 0.5

`emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: peach.optm.optm.Optimizer.__init__

step(*self*, *x*)

One step of the search.
In this method, the result of the step is highly dependent of the steps executed before, as the search step is updated at each call to this method.

Parameters

`x:` The value from where the new estimate should be calculated. This can of course be the result of a previous iteration of the algorithm.

Return Value

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.step

__call__(*self*, *x*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error.

Parameters

`x:` The value from where the search must start.

Return Value

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.__call__

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

21.2.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

21.3 Class Interpolation

object

peach.optm.optm.Optimizer

peach.optm.linear.Interpolation

Optimization by quadractic interpolation.

This methods takes three estimates and finds the parabolic function that fits them, and returns as a new estimate the vertex of the parabola. The procedure can be repeated until a good approximation is found.

21.3.1 Methods

__init__(*self*, *f*, *emax*=`1e-05`, *imax*=`1000`)

Initializes the optimizer.
To create an optimizer of this type, instantiate the class with the parameters given below:

Parameters

`f:` A one variable only function to be optimized. The function should have only one parameter and return the function value.

`dx:` The initial step of the search. Defaults to 0.5

`emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: peach.optm.optm.Optimizer.__init__

step(*self*, *x*)

One step of the search.
In this method, the result of the step is dependent only of the given estimated, so it can be used for different kind of investigations on the same cost function.

Parameters

`x:` A triple `(x0, x1, x2)`, with `x0 < x1 < x2` of estimates on the cost function. From these values the new estimate is calculated. This can of course be the result of a previous iteration of the algorithm.

Return Value

This method returns a tuple `(x, e)`, where `x` is the updated triplet of estimates of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.step

__call__(*self*, *x*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error.

Parameters

`x:` The initial triplet of values from where the search must start.

Return Value

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.__call__

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

21.3.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

21.4 Class GoldenRule

object

peach.optm.optm.Optimizer

peach.optm.linear.GoldenRule

Optimizer by the Golden Section Rule

This optimizer uses the golden rule to section an interval in search of the minimum. Using a simple heuristic, the interval is refined until an interval small enough to satisfy the error requirements is found.

21.4.1 Methods

__init__(*self*, *f*, *emax*=`1e-05`, *imax*=`1000`)

Initializes the optimizer.
To create an optimizer of this type, instantiate the class with the parameters given below:

Parameters

`f:` A one variable only function to be optimized. The function should have only one parameter and return the function value.

`dx:` The initial step of the search. Defaults to 0.5

`emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: peach.optm.optm.Optimizer.__init__

step(*self*, *x*)

One step of the search.
In this method, the result of the step is dependent only of the given estimated, so it can be used for different kind of investigations on the same cost function.

Parameters

`x:` A duple `(x0, x1)`, with `x0 < x1` of estimates on the cost function. From these values the new estimate is calculated. This can of course be the result of a previous iteration of the algorithm.

Return Value

This method returns a tuple `(x, e)`, where `x` is the updated duple of estimates of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.step

__call__(*self*, *x*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error.

Parameters

`x:` The initial duple of values from where the search must start.

Return Value

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.__call__

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

21.4.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

21.5 Class Fibonacci

object

peach.optm.optm.Optimizer

peach.optm.linear.Fibonacci

Optimization by the Golden Rule Section, estimated by Fibonacci numbers.

This optimizer uses the golden rule to section an interval in search of the minimum. Using a simple heuristic, the interval is refined until an interval small enough to satisfy the error requirements is found. The golden section is estimated at each step using Fibonacci numbers. This can be useful in situations where only integer numbers should be used.

21.5.1 Methods

__init__(*self*, *f*, *emax*=`1e-05`, *imax*=`1000`)

Initializes the optimizer.
To create an optimizer of this type, instantiate the class with the parameters given below:

Parameters

- `f:` A one variable only function to be optimized. The function should have only one parameter and return the function value.
- `dx:` The initial step of the search. Defaults to 0.5
- `emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.
- `imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: peach.optm.optm.Optimizer.__init__

step(*self*, *x*)

One step of the search.
In this method, the result of the step is highly dependent of the steps executed before, as the estimate of the golden ratio is updated at each call to this method.

Parameters

`x:` A duple `(x0, x1)`, with `x0 < x1` of estimates on the cost function. From these values the new estimate is calculated. This can of course be the result of a previous iteration of the algorithm.

Return Value

This method returns a tuple `(x, e)`, where `x` is the updated duple of estimates of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.step

__call__(*self*, *x*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error.

Parameters

`x:` The initial duple of values from where the search must start.

Return Value

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.__call__

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

21.5.2 Properties

Name	Description
__class__	**Value:** <attribute '__class__' of 'object' objects>

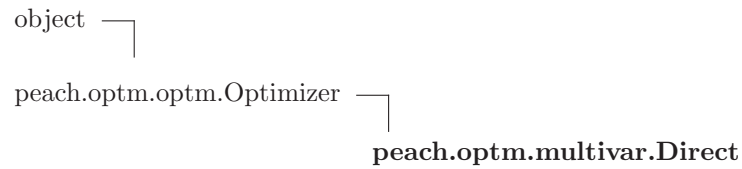
22 Module peach.optm.multivar

This package implements basic multivariable optimizers, including gradient and Newton searches.

22.1 Variables

Name	Description
__doc__	**Value:** ...

22.2 Class Direct



Multidimensional direct search

This optimization method is a generalization of the 1D method, using variable swap as search direction. This results in a very simplistic and inefficient method that should be used only when any other method fails.

22.2.1 Methods

__init__(*self*, *f*, *h*=`0.5`, *emax*=`1e-08`, *imax*=`1000`)

Initializes the optimizer.
To create an optimizer of this type, instantiate the class with the parameters given below:

Parameters

`f:` A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.

`dx:` The initial step of the search. Defaults to 0.5

`emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: peach.optm.optm.Optimizer.__init__

step(*self*, *x*)

One step of the search.
In this method, the result of the step is highly dependent of the steps executed before, as the search step is updated at each call to this method.
One characteristic of this method is that it uses the dimensions of the input vector to initialize the updating matrix -- this is necessary to mantain coherence with the interface of other methods. The matrix dimensions are calculated and used in the first call to this method and used in future calls. In general, there is no need to worry about this, since everything is taken care of automatically. But, if future calls to this method are done with incoherent dimensions, then an exception will be raised.

Parameters

`x:` The value from where the new estimate should be calculated. This can of course be the result of a previous iteration of the algorithm.

Return Value

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.step

__call__(*self*, *x*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error.

Parameters

`x:` The value from where the search must start.

Return Value

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.__call__

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

22.2.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

22.3 Class Gradient

object

peach.optm.optm.Optimizer

peach.optm.multivar.Gradient

Gradient search

This method uses the fact that the gradient of a function points to the direction of largest increase in the function (in general called *uphill* direction). So, the contrary direction (*downhill*) is used as search direction.

22.3.1 Methods

__init__(*self*, *f*, *df*=`None`, *h*=`0.1`, *emax*=`1e-05`, *imax*=`1000`)

Initializes the optimizer.
To create an optimizer of this type, instantiate the class with the parameters given below:

Parameters

`f:` A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.

`df:` A function to calculate the gradient vector of the cost function `f`. Defaults to `None`, if no gradient is supplied, then it is estimated from the cost function using Euler equations.

`h:` Convergence step. This method does not takes into consideration the possibility of varying the convergence step, to avoid Stiefel cages.

`emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: peach.optm.optm.Optimizer.__init__

step(*self*, *x*)

One step of the search.
In this method, the result of the step is dependent only of the given estimated, so it can be used for different kind of investigations on the same cost function.

Parameters

`x:` The value from where the new estimate should be calculated. This can of course be the result of a previous iteration of the algorithm.

Return Value

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.step

__call__(*self*, *x*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error.

Parameters

`x:` The initial triplet of values from where the search must start.

Return Value

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.__call__

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)
Return Value
`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

22.3.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

22.4 Class Newton

object
peach.optm.optm.Optimizer
peach.optm.multivar.Newton

Newton search

This is a very effective method to find minimum points in functions. In a very basic fashion, this method corresponds to using Newton root finding method on f'(x). Converges *very* fast if the cost function is quadratic of simmilar to it.

22.4.1 Methods

__init__(*self*, *f*, *df*=`None`, *hf*=`None`, *h*=`0.1`, *emax*=`1e-05`, *imax*=`1000`)

Initializes the optimizer.
To create an optimizer of this type, instantiate the class with the parameters given below:

Parameters

`f`: A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.

`df`: A function to calculate the gradient vector of the cost function `f`. Defaults to `None`, if no gradient is supplied, then it is estimated from the cost function using Euler equations.

`hf`: A function to calculate the hessian matrix of the cost function `f`. Defaults to `None`, if no hessian is supplied, then it is estimated from the cost function using Euler equations.

`h`: Convergence step. This method does not takes into consideration the possibility of varying the convergence step, to avoid Stiefel cages.

`emax`: Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax`: Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: peach.optm.optm.Optimizer.__init__

step(*self*, *x*)

One step of the search.
In this method, the result of the step is dependent only of the given estimated, so it can be used for different kind of investigations on the same cost function.

Parameters

`x`: The value from where the new estimate should be calculated. This can of course be the result of a previous iteration of the algorithm.

Return Value

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.step

__call__(*self*, *x*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error.

Parameters

`x`: The initial triplet of values from where the search must start.

Return Value

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.__call__

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)
Return Value
`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

22.4.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

23 Module peach.optm.optm

Basic definitons and base class for optimizers

This sub-package exports some auxiliary functions to work with cost functions, namely, a function to calculate gradient vectors and hessian matrices, which are extremely important in optimization.

Also, a base class, `Optimizer`, for all optimizers. Sub-class this class if you want to create your own optmizer, and follow the interface. This will allow easy configuration of your own scripts and comparison between methods.

23.1 Functions

gradient(*f*, *dx*=`1e-05`)

Creates a function that calculates the gradient vector of a scalar field.
This function takes as a parameter a scalar function and creates a new function that is able to calculate the derivative (in case of single variable functions) or the gradient vector (in case of multivariable functions. Please, note that this function takes as a parameter a *function*, and returns as a result *another function*. Calling the returned function on a point will give the gradient vector of the original function at that point:

```
>>> def f(x):
        return x^2

>>> df = gradient(f)
>>> df(1)
2
```

In the above example, `df` is a generated function which will return the result of the expression `2*x`, the derivative of the original function. In the case `f` is a multivariable function, it is assumed that its argument is a line vector.

Parameters

`f:` Any function, one- or multivariable. The function must be an scalar function, though there is no checking at the moment the function is created. If `f` is not an scalar function, an exception will be raised at the moment the returned function is used.

`dx:` Optional argument that gives the precision of the calculation. It is recommended that `dx = sqrt(D)`, where `D` is the machine precision. It defaults to `1e-5`, which usually gives a good estimate.

Return Value

A new function which, upon calling, gives the derivative or gradient vector of the original function on the analised point. The parameter of the returned function is a real number or a line vector where the gradient should be calculated.

hessian(*f*, *dx*=`1e-05`)

Creates a function that calculates the hessian matrix of a scalar field.
This function takes as a parameter a scalar function and creates a new function that is able to calculate the second derivative (in case of single variable functions) or the hessian matrix (in case of multivariable functions. Please, note that this function takes as a parameter a *function*, and returns as a result *another function*. Calling the returned function on a point will give the hessian matrix of the original function at that point:

```
>>> def f(x):
        return x^4

>>> ddf = hessian(f)
>>> ddf(1)
12
```

In the above example, `ddf` is a generated function which will return the result of the expression `12*x**2`, the second derivative of the original function. In the case `f` is a multivariable function, it is assumed that its argument is a line vector.

Parameters

`f:` Any function, one- or multivariable. The function must be an scalar function, though there is no checking at the moment the function is created. If `f` is not an scalar function, an exception will be raised at the moment the returned function is used.

`dx:` Optional argument that gives the precision of the calculation. It is recommended that `dx = sqrt(D)`, where `D` is the machine precision. It defaults to `1e-5`, which usually gives a good estimate.

Return Value

A new function which, upon calling, gives the second derivative or hessian matrix of the original function on the analised point. The parameter of the returned function is a real number or a line vector where the hessian should be calculated.

23.2 Variables

Name	Description
__doc__	**Value:** ...

23.3 Class Optimizer

object
└ **peach.optm.optm.Optimizer**

Known Subclasses: peach.optm.stochastic.CrossEntropy, peach.optm.quasinewton.BFGS, peach.optm.quasinewton.DFP, peach.optm.quasinewton.SR1, peach.optm.multivar.Direct, peach.optm.multivar.Gradient, peach.optm.multivar.Newton, peach.optm.linear.Direct1D, peach.optm.linear.Fibonacci, peach.optm.linear.GoldenRule, peach.optm.linear.Interpolation, peach.optm.sa.ContinuousSA, peach.optm.sa.DiscreteSA

Base class for all optimizers.

This class does nothing, and shouldn't be instantiated. Its only purpose is to serve as a template (or interface) to implemented optimizers. To create your own optimizer, subclass this.

This class defines 3 methods that should be present in any subclass. They are defined here:

__init__ Initializes the optimizer. There are three usual parameters in this method, which signature should be:

```
__init__(self, f, ..., emax=1e-8, imax=1000)
```

where:
- `f` is the cost function to be minimized;
- `...` represent additional configuration of the optimizer, and it is dependent of the technique implemented;
- `emax` is the maximum allowed error. The default value above is only a suggestion;
- `imax` is the maximum number of iterations of the method. The default value above is only a suggestions.

step This method should take an estimate and calculate the next, possibly better, estimate. Notice that the next estimate is strongly dependent of the method, the optimizer state and configuration, and two calls to this method with the same estimate might not give the same results. The method signature is:

```
step(self, x)
```

and the implementation should keep track of all the needed parameters. The method should return a tuple `(x, e)` with the new estimate of the solution and the estimate of the error.

__call__ This method should take an estimate and iterate the optimizer until one of the stop criteria is met: either less than the maximum error or more than the maximum number of iterations. Error is usually calculated as an estimate using the previous estimate, but any technique might be used. Use a counter to keep track of the number of iterations. The method signature is:

```
__call__(self, x)
```

and the implementation should keep track of all the needed parameters. The method should return a tuple `(x, e)` with the final estimate of the solution and the estimate of the error.

23.3.1 Methods

__call__(*self*, *x*)

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__init__(*self*, *f*=`None`, *emax*=`1e-08`, *imax*=`1000`)
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

Overrides: object.__init__ extit(inherited documentation)

__new__(*T*, *S*, *...*)
Return Value
`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

step(*self*, *x*)

23.3.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

24 Module peach.optm.quasinewton

This package implements basic quasi-Newton optimizers. Newton optimizer is very efficient, except that inverse matrices need to be calculated at each convergence step. These methods try to estimate the hessian inverse iteratively, thus increasing performance.

24.1 Variables

Name	Description
__doc__	**Value:** ...

24.2 Class DFP

object
└
peach.optm.optm.Optimizer
└
peach.optm.quasinewton.DFP

DFP (*Davidon-Fletcher-Powell*) search

24.2.1 Methods

__init__(*self*, *f*, *df*=`None`, *B*=`None`, *h*=`0.1`, *emax*=`1e-05`, *imax*=`1000`)

Initializes the optimizer.
To create an optimizer of this type, instantiate the class with the parameters given below:

Parameters

`f`: A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.

`df`: A function to calculate the gradient vector of the cost function `f`. Defaults to `None`, if no gradient is supplied, then it is estimated from the cost function using Euler equations.

`B`: A first estimate of the inverse hessian. Note that, differently from the Newton method, the elements in this matrix are numbers, not functions. So, it is an estimate at a given point, and its values *should* be coherent with the first estimate (that is, `B` should be the inverse of the hessian evaluated at the first estimate), or else the algorithm might diverge. Defaults to `None`, if none is given, it is estimated. Note that, given the same reasons as before, the estimate of `B` is deferred to the first calling of the `step` method, where it is handled automatically.

`h`: Convergence step. This method does not takes into consideration the possibility of varying the convergence step, to avoid Stiefel cages.

`emax`: Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax`: Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: peach.optm.optm.Optimizer.__init__

step(*self*, *x*)

One step of the search.
In this method, the result of the step is dependent of parameters calculated before (namely, the estimate of the inverse hessian), so it is not recomended that different investigations are used with the same optimizer in the same cost function.

Parameters

`x:` The value from where the new estimate should be calculated. This can of course be the result of a previous iteration of the algorithm.

Return Value

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.step

__call__(*self*, *x*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error.

Parameters

`x:` The initial triplet of values from where the search must start.

Return Value

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.__call__

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

24.2.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

24.3 Class BFGS

object

peach.optm.optm.Optimizer

peach.optm.quasinewton.BFGS

BFGS (*Broyden-Fletcher-Goldfarb-Shanno*) search

24.3.1 Methods

__init__(*self*, *f*, *df*=`None`, *B*=`None`, *h*=`0.1`, *emax*=`1e-05`, *imax*=`1000`)

Initializes the optimizer.
To create an optimizer of this type, instantiate the class with the parameters given below:

Parameters

`f`: A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.

`df`: A function to calculate the gradient vector of the cost function `f`. Defaults to `None`, if no gradient is supplied, then it is estimated from the cost function using Euler equations.

`B`: A first estimate of the inverse hessian. Note that, differently from the Newton method, the elements in this matrix are numbers, not functions. So, it is an estimate at a given point, and its values *should* be coherent with the first estimate (that is, `B` should be the inverse of the hessian evaluated at the first estimate), or else the algorithm might diverge. Defaults to `None`, if none is given, it is estimated. Note that, given the same reasons as before, the estimate of `B` is deferred to the first calling of the `step` method, where it is handled automatically.

`h`: Convergence step. This method does not takes into consideration the possibility of varying the convergence step, to avoid Stiefel cages.

`emax`: Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax`: Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: peach.optm.optm.Optimizer.__init__

step(*self*, *x*)

One step of the search.
In this method, the result of the step is dependent of parameters calculated before (namely, the estimate of the inverse hessian), so it is not recomended that different investigations are used with the same optimizer in the same cost function.

Parameters

`x`: The value from where the new estimate should be calculated. This can of course be the result of a previous iteration of the algorithm.

Return Value

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.step

__call__(*self*, *x*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error.

Parameters

`x:` The initial triplet of values from where the search must start.

Return Value

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.__call__

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

24.3.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

24.4 Class SR1

object

peach.optm.optm.Optimizer

peach.optm.quasinewton.SR1

SR1 (*Symmetric Rank 1*)search method

24.4.1 Methods

__init__(*self*, *f*, *df*=`None`, *B*=`None`, *h*=`0.1`, *emax*=`1e-05`, *imax*=`1000`)

Initializes the optimizer.
To create an optimizer of this type, instantiate the class with the parameters given below:

Parameters

`f:` A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.

`df:` A function to calculate the gradient vector of the cost function `f`. Defaults to `None`, if no gradient is supplied, then it is estimated from the cost function using Euler equations.

`B:` A first estimate of the inverse hessian. Note that, differently from the Newton method, the elements in this matrix are numbers, not functions. So, it is an estimate at a given point, and its values *should* be coherent with the first estimate (that is, `B` should be the inverse of the hessian evaluated at the first estimate), or else the algorithm might diverge. Defaults to `None`, if none is given, it is estimated. Note that, given the same reasons as before, the estimate of `B` is deferred to the first calling of the `step` method, where it is handled automatically.

`h:` Convergence step. This method does not takes into consideration the possibility of varying the convergence step, to avoid Stiefel cages.

`emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: peach.optm.optm.Optimizer.__init__

step(*self*, *x*)

One step of the search.
In this method, the result of the step is dependent of parameters calculated before (namely, the estimate of the inverse hessian), so it is not recomended that different investigations are used with the same optimizer in the same cost function.

Parameters

`x:` The value from where the new estimate should be calculated. This can of course be the result of a previous iteration of the algorithm.

Return Value

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.step

__call__(*self*, *x*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error.

Parameters

`x:` The initial triplet of values from where the search must start.

Return Value

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.__call__

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

24.4.2 Properties

Name	Description
__class__	**Value:** <attribute '__class__' of 'object' objects>

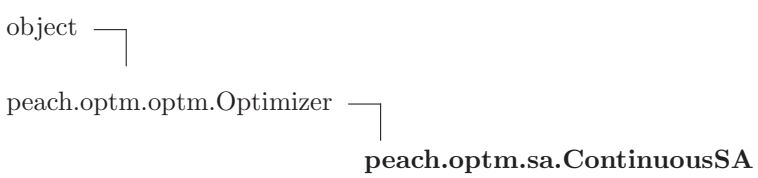
25 Module peach.optm.sa

This package implements two versions of simulated annealing optimization. One works with numeric data, and the other with a codified bit string. This last method can be used in discrete optimization problems.

25.1 Variables

Name	Description
__doc__	**Value:** ...

25.2 Class ContinuousSA



Simulated Annealing continuous optimization.

This is a simulated annealing optimizer implemented to work with vectors of continuous variables (obviouslly, implemented as floating point numbers). In general, simulated annealing methods searches for neighbors of one estimate, which makes a lot more sense in discrete problems. While in this class the method is implemented in a different way (to deal with continuous variables), the principle is pretty much the same -- the neighbor is found based on a gaussian neighborhood.

A simulated annealing algorithm adapted to deal with continuous variables has an enhancement that can be used: a gradient vector can be given and, in case the neighbor is not accepted, the estimate is updated in the downhill direction.

25.2.1 Methods

__init__(*self*, *f*, *df*=`None`, *T0*=`1000.0`, *rt*=`0.95`, *h*=`0.05`, *emax*=`1e-08`, *imax*=`1000`)

Initializes the optimizer.
To create an optimizer of this type, instantiate the class with the parameters given below:

Parameters

`f`: A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.

`df`: A function to calculate the gradient vector of the cost function `f`. Defaults to `None`, if no gradient is supplied, then it is estimated from the cost function using Euler equations.

`T0`: Initial temperature of the system. The temperature is, of course, an analogy. Defaults to 1000.

`rt`: Temperature decreasing rate. The temperature must slowly decrease in simulated annealing algorithms. In this implementation, this is controlled by this parameter. At each step, the temperature is multiplied by this value, so it is necessary that `0 < rt < 1`. Defaults to 0.95, smaller values make the temperature decay faster, while larger values make the temperature decay slower.

`h`: Convergence step. In the case that the neighbor estimate is not accepted, a simple gradient step is executed. This parameter is the convergence step to the gradient step.

`emax`: Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax`: Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: peach.optm.optm.Optimizer.__init__

step(*self*, *x*)

One step of the search.
In this method, a neighbor of the given estimate is chosen at random, using a gaussian neighborhood. It is accepted as a new estimate if it performs better in the cost function *or* if the temperature is high enough. In case it is not accepted, a gradient step is executed.

Parameters

`x`: The value from where the new estimate should be calculated. This can of course be the result of a previous iteration of the algorithm.

Return Value

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.step

__call__(*self*, *x*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error.

Parameters

`x:` The initial triplet of values from where the search must start.

Return Value

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.__call__

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

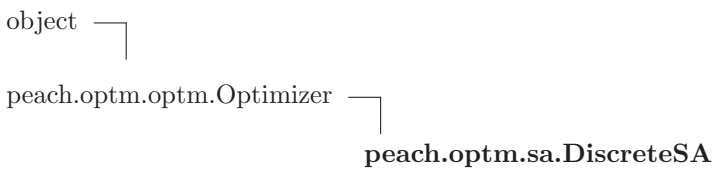
__str__(*x*)

str(x)

25.2.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

25.3 Class DiscreteSA



Simulated Annealing discrete optimization.

This is a simulated annealing optimizer implemented to work with vectors of discrete variables, which can be floating point or integer numbers, characters or anything allowed by the `struct` module of the Python standard library. The neighborhood of an estimate is calculated by inverting a number of bits randomly according to a given rate. Given the nature of this implementation, no alternate convergence can be used in the case of rejection of an estimate.

25.3.1 Methods

__init__(*self*, *f*, *fmt*, *ranges*=[], *T0*=1000.0, *rt*=0.95, *nb*=1, *emax*=1e-08, *imax*=1000)

Initializes the optimizer.
To create an optimizer of this type, instantiate the class with the parameters given below:

Parameters

`f:` A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.

`fmt:` A `struct`-module string with the format of the data used. Please, consult the `struct` documentation, since what is explained there is exactly what is used here. For example, if you are going to use the optimizer to deal with three-dimensional vectors of continuous variables, the format would be something like:

```
fmt = 'fff'
```

Default value is an empty string. Notice that this is implemented as a `bitarray`, so this module must be present.
It is strongly recommended that integer numbers are used! Floating point numbers can be simulated with long integers. The reason for this is that random bit sequences can have no representation as floating point numbers, and that can make the algorithm not perform adequatelly.

`ranges:` Ranges of values allowed for each component of the input vector. If given, ranges are checked and a new estimate is generated in case any of the components fall beyond the value. `range` can be a tuple containing the inferior and superior limits of the interval; in that case, the same range is used for every variable in the input vector. `range` can also be a list of tuples of the same format, inferior and superior limits; in that case, the first tuple is assumed as the range allowed for the first variable, the second tuple is assumed as the range allowed for the second variable and so on.

`T0:` Initial temperature of the system. The temperature is, of course, an analogy. Defaults to 1000.

`rt:` Temperature decreasing rate. The temperature must slowly decrease in simulated annealing algorithms. In this implementation, this is controlled by this parameter. At each step, the temperature is multiplied by this value, so it is necessary that `0 < rt < 1`. Defaults to 0.95, smaller values make the temperature decay faster, while larger values make the temperature decay slower.

`nb:` The number of bits to be randomly choosen to be inverted in the calculation of the neighbor. Be very careful while choosing this parameter. While very large optimizations can benefit from a big value here, it is not recommended that more than one bit per variable is inverted at each step -- otherwise, the neighbor might fall very far from the present estimate, which can make the algorithm not work accordingly. This defaults to 1, that is, at each step, only one bit will be inverted at most.

`emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.

`imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: peach.optm.optm.Optimizer.__init__

step(*self*, *x*)

One step of the search.
In this method, a neighbor of the given estimate is obtained from the present estimate by choosing `nb` bits and inverting them. It is accepted as a new estimate if it performs better in the cost function *or* if the temperature is high enough. In case it is not accepted, the previous estimate is mantained.

Parameters

`x:` The value from where the new estimate should be calculated. This can of course be the result of a previous iteration of the algorithm.

Return Value

This method returns a tuple `(x, e)`, where `x` is the updated estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.step

__call__(*self*, *x*)

Transparently executes the search until the minimum is found. The stop criteria are the maximum error or the maximum number of iterations, whichever is reached first. Note that this is a `__call__` method, so the object is called as a function. This method returns a tuple `(x, e)`, with the best estimate of the minimum and the error.

Parameters

`x:` The initial triplet of values from where the search must start.

Return Value

This method returns a tuple `(x, e)`, where `x` is the best estimate of the minimum, and `e` is the estimated error.

Overrides: peach.optm.optm.Optimizer.__call__

__delattr__(*...*)

x.__delattr__('name') <==> del x.name

__getattribute__(*...*)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, *...*)

Return Value

`a new object with type S, a subtype of T`

__reduce__(*...*)

helper for pickle

__reduce_ex__(*...*)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(*...*)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

25.3.2 Properties

Name	Description
__class__	**Value:** <attribute '__class__' of 'object' objects>

26 Module peach.optm.stochastic

General methods of stochastic optimization.

26.1 Variables

Name	Description
__doc__	**Value:** ...

26.2 Class CrossEntropy

Multidimensional search based on cross-entropy technique.

In cross-entropy, a set of N possible solutions is randomly generated at each interaction. To converge the solutions, the best M solutions are selected and its statistics are calculated. A new set of solutions are randomly generated from these statistics.

26.2.1 Methods

__init__(*self*, *f*, *M*=`30`, *N*=`60`, *emax*=`1e-08`, *imax*=`1000`)

Initializes the optimizer.
To create an optimizer of this type, instantiate the class with the parameters given below:

Parameters

- `f:` A multivariable function to be optimized. The function should have only one parameter, a multidimensional line-vector, and return the function value, a scalar.
- `M:` Size of the solution set used to calculate the statistics to generate the next set of solutions
- `N:` Total size of the solution set.
- `emax:` Maximum allowed error. The algorithm stops as soon as the error is below this level. The error is absolute.
- `imax:` Maximum number of iterations, the algorithm stops as soon this number of iterations are executed, no matter what the error is at the moment.

Overrides: peach.optm.optm.Optimizer.__init__

step(*self*)

One step of the search (*NOT IMPLEMENTED YET*)
In this method, the solution set is searched for the M best solutions. Mean and variance of these solutions is calculated, and these values are used to randomly generate, from a gaussian distribution, a set of N new solutions.

Overrides: peach.optm.optm.Optimizer.step

__call__(*self*, *x*)

__delattr__(...)

x.__delattr__('name') <==> del x.name

__getattribute__(...)

x.__getattribute__('name') <==> x.name

__hash__(*x*)

hash(x)

__new__(*T*, *S*, ...)
Return Value
`a new object with type S, a subtype of T`

__reduce__(...)

helper for pickle

__reduce_ex__(...)

helper for pickle

__repr__(*x*)

repr(x)

__setattr__(...)

x.__setattr__('name', value) <==> x.name = value

__str__(*x*)

str(x)

26.2.2 Properties

Name	Description
__class__	**Value:** `<attribute '__class__' of 'object' objects>`

Index