Parameters Documentation

Release 0.2.1

Eilif Muller and Andrew Davison

March 02, 2014

Contents

We consider it to be best practice to cleanly separate the parameters of a model from the model itself. At the least, parameters should be defined in a separate section at the start of a file. Ideally, they should be defined in a separate file entirely. This makes version control easier, since the model code typically changes less often than the parameters, and makes it easier to track a simulation project, since the parameter sets can be stored in a database, displayed in a GUI, etc.

The **Parameters** package provides Python classes to make it easier to work with parameter sets for complex models. In particular it provides tools for

- working with parameters for models that have a deep hierarchical structure;
- specifying that a parameter value should be drawn from a random distribution;
- specifying a range of values, for example for performing a sensitivity analysis;
- specifying the physical dimensions and range of permissible values of parameters;
- defining and iterating over multiple points in a parameter space;
- validation of parameter sets against a pre-defined schema.

Contents:

CHAPTER 1

Installation

Before installing Parameters, you should install the following packages:

- NumPy
- PyYAML
- SciPy (optional - needed for certain random distributions)

You can then install Parameters using:

```
$ pip install parameters
```

CHAPTER 2

Parameters

At their simplest, individual parameters consist of a name and a value. The value is either a simple type such as a numerical value or a string, or an aggregate of such simple types, such as a set, list or array.

However, we may also wish to specify the physical dimensions of the parameter, i.e., its units, and the range of permissible values.

It is also often useful to specify an object that generates numerical values or strings, such as a random number generator, and treat that object as the parameter.

To support all these uses, we define the `Parameter` and `ParameterRange` classes, and various subclasses of the `ParameterDist` abstract class, such as `GammaDist`, `NormalDist` and `UniformDist`.

2.1 The `Parameter` class

Here are some examples of creating `Parameter` objects:

```
>>> i1 = Parameter(3)
>>> f1 = Parameter(6.2)
>>> f2 = Parameter(-65.3, "mV")
>>> s1 = Parameter("hello", name="message_to_the_world")
```

The parameter name, units, value and type can be accessed as attributes:

```
>>> i1.value
3
>>> f1.type
<type 'float'>
>>> f2.units
'mV'
>>> s1.name
'message_to_the_world'
```

`Parameter` objects are not hugely useful at the moment. The units are not used for checking dimensional consistency, for example, and `Parameter` objects are not drop-in replacements for numerical values - you must always use the `value` attribute to access the value, whereas it might be nice to define, for example, a class `IntegerParameter` which was a subclass of the built-in `int` type.

2.2 The `ParameterRange` class

When investigating the behaviour of a model or in doing sensitivity analysis, it is often useful to run a model several times using a different value for a certain parameter each time (also see the `iter_range_keys()` and similar methods of the `ParameterSet` class, below). The `ParameterRange` class supports this. Some usage examples:

```
>>> tau_m_range = ParameterRange([10.0, 15.0, 20.0], "ms", "tau_m")
>>> tau_m_range.name
'tau_m'
>>> tau_m_range.next()
10.0
>>> tau_m_range.next()
15.0
>>> [2*tau_m for tau_m in tau_m_range]
[20.0, 30.0, 40.0]
```

2.3 The `ParameterDist` classes

As with taking parameter values from a series or range, it is often useful to pick values from a particular random distribution. Three classes are available: `UniformDist`, `GammaDist` and `NormalDist`. Examples:

```
>>> ud = UniformDist(min=-1.0, max=1.0)
>>> gd = GammaDist(mean=0.5, std=1.0)
>>> nd = NormalDist(mean=-70, std=5.0)
>>> ud.next()
array([-0.56342352])
>>> gd.next(3)
array([ 0.04061142,  0.05550265,  0.23469344])
>>> nd.next(2)
array([-76.18506715, -68.71229944])
```

CHAPTER 3

Parameter sets

A problem with parameter sets for large-scale, detailed models is that the list of parameters gets very long and unwieldy, and due to the typically hierarchical nature of such models, the individual parameter names can also get very long, e.g., `v1_layer5_pyramidal_apical_dend_gbar_na`.

A solution to this is to give the parameter set a hierarchical structure as well, which allows the top-level list of parameters to be very short (e.g. `v1`, `retina` and `lgn` for a visual system simulation) since the top-level parameters are themselves parameter sets.

The simplest way to implement this in Python is using nested dicts. One disadvantage of this is that accessing deeply-nested parameters can be very verbose, e.g. `v1['layer5']['pyramidal']['apical_dend']['na']['gbar']`. A second disadvantage is that it is tedious to flatten the hierarchy when this becomes necessary, e.g. for serialisation - writing to file, etc.

For these reasons we have created a `ParameterSet` class, which:

1. allows a more convenient notation;
2. enables subsets of the parameters, lower in the hierarchy, to be passed around by themselves;
3. provides convenient methods for reading from/writing to file and for determining the differences between two different parameter sets.

An example of the notation is `v1.layer5.pyramidal.apical_dend.na.gbar`, which requires only a single `.` for each level in the hierarchy rather than two "'"s, a "`[`" and a "`]`". This is not much shorter than `v1_layer5_pyramidal_apical_dend_gbar_na` - the difference is that `v1.layer5.pyramidal` is itself a `ParameterSet` object that can be passed as an argument to the pyramidal cell object, which doesn't care about `v1.layer4.spinystellate`, let alone `retina.ganglioncell.magno.tau_m` (while `v1_layer5_pyramidal` is just a `NameError`).

3.1 The `ParameterSet` class

3.1.1 Creation

`ParameterSet` objects may be created from a dict:

```
>>> sim_params = ParameterSet({'dt': 0.11, 'tstop': 1000.0})
```

or loaded from a URL:

```
>>> exc_cell_params = ParameterSet("https://neuralensemble.org/svn/NeuroTools/trunk/doc/example.para
```

They may be nested:

```
>>> inh_cell_params = ParameterSet({'tau_m': 15.0, 'cm': 0.5})
>>> network_params = ParameterSet({'excitatory_cells': exc_cell_params, 'inhibitory_cells': inh_cell_
>>> P = ParameterSet({'sim': sim_params, 'network': network_params}, label="my_params")
```

Note that although we show here only numerical parameter values, `Parameter`, `ParameterRange` and `ParameterDist` objects, as well as strings, may also be parameter values.

Todo

describe references ('ref' and the `ParameterReference` class)

3.1.2 Navigation

Individual parameters may be accessed/set using dot notation:

```
>>> P.sim.dt
0.11
>>> P.network.inhibitory_cells.tau_m
15.0
>>> P.network.inhibitory_cells.cm = 0.75
```

or the usual dictionary access notation:

```
>>> P['network']['inhibitory_cells']['cm']
0.75
```

or mixing the two (which may be required if some of the parameter names contain spaces):

```
>>> P['network'].excitatory_cells['tau_m']
10.0
```

3.1.3 Viewing and saving

To see the entire parameter set at once, nicely formatted use the `pretty()` method:

```
>>> print P.pretty()
{
  "network": {
    "excitatory_cells": url("https://neuralensemble.org/svn/NeuroTools/trunk/doc/example.param"),
    "inhibitory_cells": {
      "tau_m": 15.0,
      "cm": 0.75,
    },
  },
  "sim": {
    "tstop": 1000.0,
    "dt": 0.11,
  },
}
```

By default, if the `ParameterSet` contains other `ParameterSets` that were loaded from URLs, these will be represented with a `url()` function in the output, but there is also the option to expand all URLs and show the full contents:

```
>>> print P.pretty(expand_urls=True)
{
  "network": {
    "excitatory_cells": {
      "tau_refrac": 0.11,
      "tau_m": 10.0,
      "cm": 0.25,
      "synI": {
        "tau": 10.0,
        "E": -75.0,
      },
      "synE": {
        "tau": 1.5,
        "E": 0.0,
      },
      "v_thresh": -57.0,
      "v_reset": -70.0,
      "v_rest": -70.0,
    },
    "inhibitory_cells": {
      "tau_m": 15.0,
      "cm": 0.75,
    },
  },
  "sim": {
    "tstop": 1000.0,
    "dt": 0.11,
  },
}
```

If a `ParameterSet` was loaded from a URL, it may be modified then saved back to the same URL, provided the protocol supports writing:

```
>>> exc_cell_params.save()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "parameters.py", line 266, in save
    raise Exception("Saving using the %s protocol is not implemented" % scheme)
Exception: Saving using the https protocol is not implemented
```

or saved to a different URL:

```
>>> exc_cell_params.save(url="file:///tmp/exc_params")
```

The file format is the same as that produced by the `pretty()` method.

3.1.4 Copying and converting

A `ParameterSet` can be used simply as a dictionary, but can also be converted explicitly to a `dict` if required:

```
>>> print sim_params.as_dict()
{'tstop': 1000.0, 'dt': 0.11}
```

[need to say something about `tree_copy()`]

3.1.5 Iteration

There are several different ways to iterate over all or part of the `ParameterSet` object. `keys()`, `values()` and `items()` work as for `dicts`. For the sake of more readable code, `names()` is provided as an alias for `keys()` and `parameters()` as an alias for `items()`:

```
>>> P.names()
['network', 'sim']
>>> exc_cell_params.parameters()
[('tau_refrac', 0.11), ('tau_m', 10.0), ('cm', 0.25),
 ('synI', {'tau': 10.0, 'E': -75.0}), ('synE', {'tau': 1.5, 'E': 0.0}),
 ('v_thresh', -57.0), ('v_reset', -70.0), ('v_rest', -70.0)]
```

To flatten nested parameter sets, i.e., the iterate recursively over all branches of the tree, the the `flatten()` method returns a `dict` with keys created by joining the names at each hierarchical level with a separator character ('.' by default):

```
>>> network_params.flatten()
{'excitatory_cells.synI.E': -75.0, 'excitatory_cells.v_rest': -70.0,
 'excitatory_cells.tau_refrac': 0.11, 'excitatory_cells.v_reset': -70.0,
 'excitatory_cells.v_thresh': -57.0, 'excitatory_cells.tau_m': 10.0,
 'excitatory_cells.synI.tau': 10.0, 'excitatory_cells.cm': 0.25,
 'inhibitory_cells.cm': 0.75, 'excitatory_cells.synE.tau': 1.5,
 'excitatory_cells.synE.E': 0.0, 'inhibitory_cells.tau_m': 15.0}
```

while the `flat()` method returns a generator which yields `(name, value)` tuples.:

```
>>> for x in network_params.flat():
...   print x
('excitatory_cells.tau_refrac', 0.11)
('excitatory_cells.tau_m', 10.0)
('excitatory_cells.cm', 0.25)
('excitatory_cells.synI.tau', 10.0)
('excitatory_cells.synI.E', -75.0)
('excitatory_cells.synE.tau', 1.5)
('excitatory_cells.synE.E', 0.0)
('excitatory_cells.v_thresh', -57.0)
('excitatory_cells.v_reset', -70.0)
('excitatory_cells.v_rest', -70.0)
('inhibitory_cells.tau_m', 15.0)
('inhibitory_cells.cm', 0.75)
```

3.2 The `ParameterTable` class

Todo

describe this

CHAPTER 4

Parameter spaces

The `ParameterSpace` class is a subclass of `ParameterSet` that is allowed to contain `ParameterRange` and `ParameterDist` objects as parameters. This turns the single point in parameter space represented by a `ParameterSet` into a set of points. For example, the following definition creates a set of six points in parameter space, which can be obtained in turn using the `iter_inner()` method:

```
>>> PS = ParameterSpace({
...         'x': 999,
...         'y': ParameterRange([10, 20]),
...         'z': ParameterRange([-1, 0, 1])
... })
>>> for P in PS.iter_inner():
...     print P
{'y': 10, 'x': 999, 'z': -1}
{'y': 20, 'x': 999, 'z': -1}
{'y': 10, 'x': 999, 'z': 0}
{'y': 20, 'x': 999, 'z': 0}
{'y': 10, 'x': 999, 'z': 1}
{'y': 20, 'x': 999, 'z': 1}
```

Putting parameter distribution objects inside a `ParameterSpace` allows an essentially infinite number of points to be generated:

```
>>> PS2 = ParameterSpace({
...     'x': UniformDist(min=-1.0, max=1.0),
...     'y': GammaDist(mean=0.5, std=1.0),
...     'z': NormalDist(mean=-70, std=5.0)
... })
>>> for P in PS2.realize_dists(n=3):
...     print P
{'y': 1.81311773668, 'x': 0.883293989399, 'z': -73.5871002759}
{'y': 0.299391158731, 'x': 0.371474054049, 'z': -68.6936045978}
{'y': 2.90108202422, 'x': -0.388218831787, 'z': -68.6681724449}
```

CHAPTER 5

Validation

CHAPTER 6

Release notes

6.1 Version 0.1

The first version of Parameters was part of NeuroTools.

6.2 Version 0.2

Version 0.2 is the first stand-alone version of Parameters. The main changes are:

- addition of parameter set validation against a predefined schema;
- Python 3 support;
- NumPy and SciPy are no longer essential, although additional functionality is available if they are installed;
- added support for YAML-format parameter files;
- added support for opening remote parameter files where you are accessing the web via a proxy server;
- added export of parameter sets in LaTeX;
- addition of references, i.e. specifying one parameter by reference to another.

6.3 Version 0.2.1

- added simple operations to references, e.g. you can specify that a given parameter is twice the value of another;
- fixed bug with complex nested references.

CHAPTER 7

Reference

7.1 ParameterSet

class parameters.**ParameterSet**(*initialiser*, *label=None*, *update_namespace=None*)

A class to manage hierarchical parameter sets.

Usage example:

```
>>> sim_params = ParameterSet({'dt': 0.1, 'tstop': 1000.0})
>>> exc_cell_params = ParameterSet("http://neuralensemble.org/svn/NeuroTools/example.params")
>>> inh_cell_params = ParameterSet({'tau_m': 15.0, 'cm': 0.5})
>>> network_params = ParameterSet({'excitatory_cells': exc_cell_params, 'inhibitory_cells': inh_
>>> P = ParameterSet({'sim': sim_params, 'network': network_params})
>>> P.sim.dt
0.1
>>> P.network.inhibitory_cells.tau_m
15.0
>>> print P.pretty()
```

as_dict()

Return a copy of the *ParameterSet* tree structure as a nested dictionary

static check_validity(*k*)

docstring missing

export(*filename*, *format='latex'*, ***kwargs*)

docstring missing

find_references()

flat()

flat_add(*name*, *value*)

Like *__setitem__*, but it will add *ParameterSet({})* objects into the namespace tree if needed.

flatten()

invalid_names = ['parameters', 'names']

non_parameter_attributes = ['_url', 'label', 'names', 'parameters', 'flat', 'flatten', 'non_parameter_attributes']

pretty(*indent=' '*, *expand_urls=False*)

Return a unicode string representing the structure of the *ParameterSet*. evaluating the string should recreate the object.

static read_from_str(*s*, *update_namespace=None*)

ParameterSet definition *s* should be a Python dict definition string, containing objects of types *int*, *float*, *str*, *list*, *dict* plus the classes defined in this module, *Parameter*, *ParameterRange*, etc. No other object types are allowed, except the function *url('some_url')* or *ref('point.delimited.path')*, e.g.:

```
{ 'a' : {'A': 3, 'B': 4},
  'b' : [1,2,3],
  'c' : 'hello world',
  'd' : url('http://example.com/my_cool_parameter_set')
  'e' : ref('level1_param_name.level2_param_name.level3_param_name') }
```

This is largely the JSON (www.json.org) format, but with extra keywords in the namespace such as *ParameterRange*, *GammaDist*, etc.

replace_references()

replace_values(***args*)

This expects its arguments to be in the form path=value, where path is a . (dot) delimited path to a parameter in the parameter tree rooted in this ParameterSet instance.

This function replaces the values of each parameter in the args with the corresponding values supplied in the arguments.

save(*url=None*, *expand_urls=False*)

Write the parameter set to a text file.

The text file syntax is open to discussion. My idea is that it should be valid Python code, preferably importable as a module.

If *url* is *None*, try to save to *self._url* (if it is not *None*), otherwise save to *url*.

tree_copy()

Return a copy of the *ParameterSet* tree structure. Nodes are not copied, but re-referenced.

update(*E*, ***F*)

docstring missing

7.2 ParameterTable

class parameters.**ParameterTable**(*initialiser*, *label=None*)

Bases: parameters.ParameterSet

A sub-class of *ParameterSet* that can represent a table of parameters.

i.e., it is limited to one-level of nesting, and each sub-dict must have the same keys. In addition to the possible initialisers for ParameterSet, a ParameterTable can be initialised from a multi-line string, e.g.:

```
>>> pt = ParameterTable('''
...     #       col1    col2    col3
...     row1     1       2       3
...     row2     4       5       6
...     row3     7       8       9
... ''')
>>> pt.row2.col3
6.0
>>> pt.column('col1')
{'row1': 1.0, 'row2': 4.0, 'row3': 7.0}
>>> pt.transpose().col3.row2
6.0
```

column(*column_label*)
Return a *ParameterSet* object containing the requested column.

column_labels()
Return a list of column labels.

columns()
Return a list of *(column_label, column)* pairs, as 2-tuples.

non_parameter_attributes = ['_url', 'label', 'names', 'parameters', 'flat', 'flatten', 'non_parameter_attributes', 're

row(*row_label*)
Return a *ParameterSet* object containing the requested row.

table_string()
Returns the table as a string, suitable for being used as the initialiser for a new *ParameterTable*.

transpose()
Return a new *ParameterTable* object with the same data as the current one but with rows and columns swapped.

7.3 ParameterSpace

class parameters.**ParameterSpace**(*initialiser*, *label=None*, *update_namespace=None*)
A collection of *ParameterSets*, representing multiple points in parameter space. Created by putting *ParameterRange* and/or *ParameterDist* objects within a *ParameterSet*.

dist_keys()
Return the list of keys for those elements which are *ParameterDists*.

get_ranges_values()
Return a dict with the keys and values of the parameters with *ParameterRanges*

Example:

```
>>> p = ParameterSpace({})
>>> p.b = ParameterRange([1,2,3])
>>> p.a = ParameterRange(['p','y','t','h','o','n'])
>>> data = p.get_ranges_values()
>>> data
{'a': ['p', 'y', 't', 'h', 'o', 'n'], 'b': [1, 2, 3]}
```

iter_inner(*copy=False*)
An iterator of the *ParameterSpace* which yields *ParameterSets* with all combinations of *ParameterRange* elements

iter_inner_range_keys(*keys*, *copy=False*)
An iterator of the *ParameterSpace* which yields *ParameterSets* with all combinations of *ParameterRange* elements which are given by the *keys* list.

Note: each newly yielded value is one and the same object so storing the returned values results in a collection of many of the lastly yielded object.

copy=True causes each yielded object to be a newly created object, but be careful because this is spawning many dictionaries!

iter_range_key(*range_key*)
An iterator of the *ParameterSpace* which yields the *ParameterSet* with the *ParameterRange* given by *range_key* replaced with each of its values

num_conditions()
Return the number of *ParameterSets* that will be returned by the *iter_inner()* method.

parameter_space_dimension_labels()
Return the dimensions and labels of the keys for those elements which are *ParameterRanges*. *range_keys* are sorted to ensure the same ordering each time.

parameter_space_index(*current_experiment*)
Return the index of the current experiment in the dimension of the parameter space i.e. parameter space dimension: [2,3] i.e. index: (1,0)

Example:

```
p = ParameterSet({})
p.b = ParameterRange([1,2,3])
p.a = ParameterRange(['p','y','t','h','o','n'])

results_dim, results_label = p.parameter_space_dimension_labels()

results = numpy.empty(results_dim)
for experiment in p.iter_inner():
    index = p.parameter_space_index(experiment)
    results[index] = 2.
```

range_keys()
Return the list of keys for those elements which are *ParameterRanges*.

realize_dists(*n=1*, *copy=False*)
For each *ParameterDist*, realize the distribution and yield the result.

If *copy==True*, causes each yielded object to be a newly created object, but be careful because this is spawning many dictionaries!

7.4 ParameterRange

class parameters.**ParameterRange**(*value*, *units=None*, *name=''*, *shuffle=False*)
A class for specifying a list of possible values for a given parameter.

The value must be an iterable. It acts like a Parameter, but .next() can be called to iterate through the values

next()

7.5 Parameter

class parameters.**Parameter**(*value*, *units=None*, *name=''*)

7.6 ParameterDist and its subclasses

class parameters.random.**ParameterDist**(***params*)
missing docstring

from_stats(*vals*, *bias=0.0*, *expand=1.0*)
missing docstring

next(*n=1*)

class parameters.random.**GammaDist**(*mean=None*, *std=None*, *repr_mode='ms'*, ***params*)
Bases: parameters.random.ParameterDist

gamma.pdf(x,a,b) = x**(a-1)*exp(-x/b)/gamma(a)/b**a

Yields strictly positive numbers. Generally the distribution is implemented by *scipy.stats.gamma.pdf(x/b,a)/b* For more info, in ipython type:

```
>>> ? scipy.stats.gamma
```

class parameters.random.**NormalDist**(*mean=0.0*, *std=1.0*)
Bases: parameters.random.ParameterDist

normal distribution with parameters mean + std

class parameters.random.**UniformDist**(*min=0.0*, *max=1.0*, *return_type=<type 'float'>*)
Bases: parameters.random.ParameterDist

uniform distribution with min,max

7.7 Validation

class parameters.validators.**ParameterSchema**(*initializer*)
A sub-class of *ParameterSet* against which other ParameterSets can be validated.

Presently, it is more or less a *ParameterSet*, with all leafs(values) which are not explicitly a subclass of the *SchemaBase* object replaced by a *Subclass(type=<leaf(value) type>)* instance.

ParameterSchema may contain arbitrary *Schema* objects subclassed from *SchemaBase* which validate leafs by the member function *validate(leaf)* returning True or false if the given leaf in the *ParameterSet* at the same path should be validated or not, e.g.:

```
LambdaSchema('isinstance(x,str)',var='x'),
*unimplemented* Timedate('%.2d-%.2m-%.2y'), etc.
*unimplemented* Email()
*unimplemented* Url()
*unimplemented* File()
*unimplemented* FileExists()
```

etc.

Example:

```
>>> schema = ParameterSchema({'age': 0, 'height': Subclass(float)})
>>> # is equivalent to
>>> schema = ParameterSchema({'age': Subclass(int), 'height': Subclass(float)})
```

See also: *SchemaBase*, *Eval*, *Subclass*

class parameters.validators.**CongruencyValidator**
A *CongruencyValidator* validates a *ParameterSet* against a *ParameterSchema* either returning *True*, or raising a *ValidationError* with the path, *SchemaBase* subclass and parameter value for which the validation failed.

The *CongruencyValidator* expects all names defined in the schema to be present in the parameter set and vice-versa, and will run validation for each item in the namespace tree.

The validation functionality is available via the "validate" member *CongruencyValidator.validate(parameter_set, parameter_schema)*

Example:

```
validator = CongruencyValidator()
try:
   validator.validate(parameter_set,parameter_schema)
except ValidationError, e:
```

See also: *ParameterSet*, *ParameterSchema*

validate(*parameter_set*, *parameter_schema*)
Validates a *ParameterSet* against a *ParameterSchema* either returning *True*, or raising a *ValidationError* with the path and *SchemaBase* subclass for which validation failed.

Expects all names defined in the schema to be present in the parameter set and vice-versa, and will run validation for each item in the namespace tree.

See also: *CongruencyValidator*.

class parameters.validators.**SchemaBase**
The base class of all "active" *Schema* objects to be placed in a *ParameterSchema*.

Schema objects define the "validate" member which accepts the to-be-validated *ParameterSet* value from the same path as the *Schema* object in the *ParameterSchema* and returns True if the value is valid, otherwise False.

validate(*leaf*)

class parameters.validators.**Subclass**(*type=None*)
Bases: parameters.validators.SchemaBase

To be used as a value in a *ParameterSchema*. Validates the same-path *ParameterSet* value if it is of the specified type.

See also: *SchemaBase*

class parameters.validators.**Eval**(*expr*, *var='leaf'*)
Bases: parameters.validators.SchemaBase

To be used as a value in a *ParameterSchema*. Validates the same-path *ParameterSet* value if the provided expression (with *leaf(value)* mapped to var in eval local namespace) evaluates to True.

See also: *SchemaBase*

CHAPTER 8

How to contribute

8.1 Reporting a bug, requesting improvements

If you find a bug in Parameters or would like to suggest an improvement, go to the issue_tracker and check whether someone else already reported the same bug or requested the same improvement. If not, click on "New Issue" and describe the problem, preferably including a code sample that reproduces the problem.

8.2 Setting up a development environment

We strongly suggest you use virtualenv to isolate your work on Parameters from your default Python installation. It is best to create two virtualenvs, one using Python 2.7, the other using Python 3.2 or later.

To run the tests, we suggest installing at least nose, NumPy and PyYAML. You may also need to install SciPy.

To obtain the Parameters source code, you will need a GitHub account. You should then fork https://github.com/NeuralEnsemble/parameters (see the GitHub documentation if you are new to GitHub) and clone your own copy of the repository:

```
git clone git@github.com:<username>/parameters.git
```

You will then need either to install Parameters in your virtualenv(s) or otherwise add it to your `PYTHONPATH`.

8.3 Running the test suite

The easiest way to run the test suite is to run **nosetests** in the root of the source directory or in the `test` subdirectory.

Tests should be run with both Python 2.7 and some version of Python >= 3.2.

8.4 Coding standards and style

Parameters is intended to support Python 2.7 and all versions of Python >= 3.2, using a single code base. For guidance on achieving this, see Porting to Python 3 and the Python 3 Porting Guide.

8.5 Contributing code

When you are happy with your changes to the code, all the tests pass with all supported versions of Python, open a pull request on Github.

8.6 Making a release

If you are the release manager for Parameters, here is a checklist for making a release:

- update the version numbers in `setup.py`, `parameters/__init__.py`, `doc/conf.py` and `doc/installation.txt`
- update `doc/changelog.txt`
- run all the tests with both Python 2 and Python 3
- `python setup.py sdist upload`
- rebuild the documentation at http://parameters.readthedocs.org/
- commit the changes, tag with release number, push to GitHub
- bump the version numbers

Note: Parameters was previously part of the NeuroTools package, but is now developed and distributed separately.