



# **SIMO API documentation**

*Release 0.5.0*

**Simosol Oy**

October 19, 2009



# CONTENTS

<b>1</b>	<b>xmlobject.py</b>	<b>3</b>
1.1	class XmlObject(object): . . . . .	3
<b>2</b>	<b>operationmapping.py</b>	<b>9</b>
2.1	class OperationMappingDef(XmlObject): . . . . .	9
<b>3</b>	<b>text2data.py</b>	<b>11</b>
3.1	class ConversionMappingDef(XmlObject): . . . . .	11
3.2	class ValueConv(Persistent): . . . . .	16
3.3	class Mapping(Persistent): . . . . .	16
3.4	class LevelDef(Persistent): . . . . .	16
3.5	class ConversionMapping(Persistent): . . . . .	17
<b>4</b>	<b>operationconversion.py</b>	<b>19</b>
4.1	class OperationConversionDef(XmlObject): . . . . .	19
<b>5</b>	<b>lexicon.py</b>	<b>25</b>
5.1	class Lexicon(object): . . . . .	27
<b>6</b>	<b>lexiconlevel.py</b>	<b>33</b>
6.1	class LexiconLevel(object): . . . . .	33
<b>7</b>	<b>lexiconvariable.py</b>	<b>35</b>
7.1	class LexiconVariable(object): . . . . .	35
<b>8</b>	<b>aggregationmodel.py</b>	<b>37</b>
8.1	class AggregationModel(object): . . . . .	38
8.2	class AggregationModelParam(object): . . . . .	38
8.3	class AggregationModelOperand(object): . . . . .	41
<b>9</b>	<b>cashflowmodel.py</b>	<b>45</b>
9.1	class CashFlowModel(object): . . . . .	45
9.2	class CashFlowModelParam(object): . . . . .	46
<b>10</b>	<b>cashflowtable.py</b>	<b>47</b>
10.1	class CashFlowTable(object): . . . . .	48
10.2	class Trend(object): . . . . .	49
10.3	class CashFlowTable(object): . . . . .	50
<b>11</b>	<b>geotable.py</b>	<b>53</b>
11.1	class GeoTable(object): . . . . .	54
11.2	class GeoTableParam(object): . . . . .	55
<b>12</b>	<b>managementmodel.py</b>	<b>57</b>
12.1	class ManagementModel(object): . . . . .	58

12.2	class ModelParamManagementModel(object):	58
<b>13</b>	<b>model.py</b>	<b>61</b>
13.1	class Model(object):	62
13.2	class POModel(Model):	63
13.3	class Limit(object):	63
<b>14</b>	<b>modelbase.py</b>	<b>65</b>
14.1	class ModelbaseDef(XmlObject):	65
<b>15</b>	<b>operationmodel.py</b>	<b>67</b>
15.1	class OperationModel(Model):	68
15.2	class Classifier(object):	71
15.3	class OperationModelParam(object):	71
<b>16</b>	<b>parametertable.py</b>	<b>75</b>
16.1	class ParameterTable(object):	76
16.2	class ParamTableTarget(object):	76
<b>17</b>	<b>predictionmodel.py</b>	<b>77</b>
17.1	class PredictionModel(POModel):	78
17.2	class ResultVariable(object):	79
17.3	class PredictionModelParam(Persistent):	79
<b>18</b>	<b>table.py</b>	<b>81</b>
18.1	def table2array(table):	81
18.2	def table2dict(table):	81
18.3	def parse_classifiers(ns, elem):	81
18.4	def parse_tables(ns, elem):	83
<b>19</b>	<b>conditionparser.py</b>	<b>85</b>
19.1	class ConditionParser(object):	85
<b>20</b>	<b>modelchain.py</b>	<b>91</b>
20.1	class ModelChainDef(XmlObject):	91
20.2	class ModelChainCollection(Persistent):	96
20.3	class ModelChain(Persistent):	97
<b>21</b>	<b>task.py</b>	<b>99</b>
21.1	class Task(object):	99
<b>22</b>	<b>exprparser.py</b>	<b>101</b>
22.1	class ExpressionParser(object):	101
<b>23</b>	<b>objfunc.py</b>	<b>109</b>
23.1	class ObjectiveFunction(object):	109
23.2	class SubObjective(object):	109
23.3	class UFunc(object):	109
<b>24</b>	<b>opttask.py</b>	<b>111</b>
24.1	class OptimizationTaskDef(XmlObject):	112
24.2	class OptimizationTask(object):	112
<b>25</b>	<b>aggrdef.py</b>	<b>115</b>
25.1	class AggregationOutputDef(XmlObject):	116
25.2	class AggregationOutput(object):	116
<b>26</b>	<b>exprdef.py</b>	<b>117</b>
26.1	class ExpressionOutputDef(XmlObject):	117
26.2	class ExpressionOutput(object):	118

<b>27</b>	<b>outputconstraint.py</b>	<b>119</b>
27.1	class OutputConstraintDef(XmlObject): . . . . .	119
<b>28</b>	<b>randomvalues.py</b>	<b>121</b>
28.1	class RandomValueDef(XmlObject): . . . . .	122
28.2	class RandomValue(object): . . . . .	122
28.3	class ErrorModel(object): . . . . .	122
28.4	class RandomVariable(object): . . . . .	122
28.5	class RandomError(object): . . . . .	123
28.6	class ClassificationError(object): . . . . .	123
28.7	class RemovalProbablity(object): . . . . .	123
28.8	class Sigmoid(object): . . . . .	124
28.9	class UniformDist(object): . . . . .	124
28.10	class NormalDist(object): . . . . .	124
28.11	class LognormalDist(object): . . . . .	124
28.12	class WeibullDist(object): . . . . .	124
28.13	class BetaDist(object): . . . . .	124
28.14	class Systematic(object): . . . . .	124
<b>29</b>	<b>simcontrol.py</b>	<b>125</b>
29.1	class SimControlDef(XmlObject): . . . . .	125
<b>30</b>	<b>lexicontranslationtable.py</b>	<b>129</b>
30.1	class LexiconTransTableDef(XmlObject): . . . . .	129
30.2	class LexiconTransTable(object): . . . . .	130
<b>31</b>	<b>messagetranslationtable.py</b>	<b>133</b>
31.1	class MsgTransTableDef(XmlObject): . . . . .	133
31.2	class MsgTransTable(object): . . . . .	134
<b>32</b>	<b>db.py</b>	<b>135</b>
32.1	class SQLiteDatabase(object): . . . . .	135
32.2	class DataDB(SQLiteDatabase): . . . . .	136
32.3	class OperationDB(SQLiteDatabase): . . . . .	149
32.4	class LoggerDB(SQLiteDatabase): . . . . .	151
<b>33</b>	<b>db.py</b>	<b>155</b>
33.1	class SimoDB(object): . . . . .	157
<b>34</b>	<b>importdata.py</b>	<b>163</b>
34.1	class DataImporter(object): . . . . .	163
<b>35</b>	<b>importops.py</b>	<b>169</b>
35.1	class OperationImporter(object): . . . . .	169
<b>36</b>	<b>brancher.py</b>	<b>173</b>
36.1	class Brancher(object): . . . . .	173
<b>37</b>	<b>handler.py</b>	<b>179</b>
37.1	class Handler(object): . . . . .	179
<b>38</b>	<b>ind2id.py</b>	<b>197</b>
38.1	class Ind2Id(object): . . . . .	197
<b>39</b>	<b>linkage.py</b>	<b>199</b>
39.1	class Linkage(object): . . . . .	199
<b>40</b>	<b>matrix.py</b>	<b>203</b>
40.1	class Matrix(object): . . . . .	203

<b>41</b>	<b>hero.py</b>	<b>207</b>
41.1	class Hero(Optimizer): . . . . .	207
<b>42</b>	<b>jlp.py</b>	<b>211</b>
42.1	class JLP(Optimizer): . . . . .	211
<b>43</b>	<b>optimizer.py</b>	<b>217</b>
43.1	class Optimizer(object): . . . . .	217
<b>44</b>	<b>tabusearch.py</b>	<b>219</b>
44.1	class TabuSearch(Optimizer): . . . . .	219
<b>45</b>	<b>ologger.py</b>	<b>223</b>
45.1	class OLogger(object): . . . . .	223
<b>46</b>	<b>omatrix.py</b>	<b>225</b>
46.1	def calculate_NPV(value, discountrate, presentdate, eventdate): . . . . .	225
46.2	def __init__(self): . . . . .	225
46.3	def construct_data(self, db): . . . . .	226
46.4	def _get_matrix_dimensions(self): . . . . .	228
46.5	def _create_empty_matrices(self): . . . . .	228
46.6	def _process_exprs(self, exprs, db, exprtype, subobj=True): . . . . .	228
46.7	def _fill_matrices(self, db): . . . . .	228
46.8	def _collect_values(self, expr, matrix): . . . . .	228
46.9	def _evaluate_operand_constraints(self, data, matrix, expr): . . . . .	231
46.10	def solution_utility(self, solution, iteration): . . . . .	231
46.11	def _value2utility(self, iso, value, maxvalue): . . . . .	231
46.12	def solution_feasibility(self, solution, iteration): . . . . .	231
46.13	def compare_utilities(self, best, candidate): . . . . .	231
46.14	def get_random_solution(self, iteration): . . . . .	232
<b>47</b>	<b>postfixeval.py</b>	<b>233</b>
47.1	class PostfixEvaluator: . . . . .	233
<b>48</b>	<b>aggr.py</b>	<b>237</b>
48.1	class OutputAggr(Output): . . . . .	237
<b>49</b>	<b>branching.py</b>	<b>243</b>
49.1	class OutputOpres(Output): . . . . .	243
<b>50</b>	<b>by_level.py</b>	<b>247</b>
50.1	class OutputByLevel(Output): . . . . .	247
<b>51</b>	<b>expression.py</b>	<b>251</b>
51.1	def calculate_NPV(value, discountrate, presentdate, eventdate): . . . . .	251
51.2	def __init__(self, datadb, data_type, id_list, main_level, result_type, output_formats, output_filenames, output_constraint=None, default_decimal_places=1, archiving=False, re- sult_padding=True, aggregation_def=None, expression_def=None, opres_vars=None): . . . . .	251
51.3	def run(self): . . . . .	252
51.4	def _run_expr(self, expr_dict): . . . . .	252
51.5	def _execute_var(self, discount_rate, time_unit, time_length, var): . . . . .	252
51.6	def _analyse_data(self): . . . . .	252
51.7	def _process_exprs(self, exprs): . . . . .	252
51.8	def _create_tables(self): . . . . .	253
51.9	def _fill_tables(self): . . . . .	253
51.10	def _collect_values(self, expr, arr, forced_level=None): . . . . .	253
51.11	def _get_values_from_db(self, operand, dates, norm, opresconst=None, forced_level=None): . . . . .	253
51.12	def _eval_data_constraint(self, dates, arr, operand, i_e, norm): . . . . .	253
51.13	def _eval_opres_constraint(self, dates, arr, operand, i_e, norm): . . . . .	253
51.14	def _set_value(self, arr, it, uid, br, i_e, date, value): . . . . .	254

51.15	def _construct_strings(self): . . . . .	254
<b>52</b>	<b>inline.py</b>	<b>255</b>
52.1	class OutputInline(Output): . . . . .	255
<b>53</b>	<b>opres.py</b>	<b>259</b>
53.1	class OutputOpres(Output): . . . . .	259
<b>54</b>	<b>out.py</b>	<b>261</b>
54.1	class Out(object): . . . . .	261
<b>55</b>	<b>output.py</b>	<b>267</b>
55.1	class Output(object): . . . . .	267
<b>56</b>	<b>smt.py</b>	<b>271</b>
56.1	class OutputSMT(Output): . . . . .	271
<b>57</b>	<b>condeval.py</b>	<b>275</b>
57.1	class ConditionEvaluator(object): . . . . .	275
<b>58</b>	<b>sim.py</b>	<b>279</b>
58.1	class Simulator(object): . . . . .	279
<b>59</b>	<b>aggregationcaller.py</b>	<b>299</b>
59.1	class AggregationModelCaller(Caller): . . . . .	299
<b>60</b>	<b>caller.py</b>	<b>305</b>
60.1	class Caller(object): . . . . .	305
<b>61</b>	<b>geotablecaller.py</b>	<b>307</b>
61.1	class GeoTableCaller(Caller): . . . . .	308
<b>62</b>	<b>managementcaller.py</b>	<b>313</b>
62.1	class ManagementModelCaller(Caller): . . . . .	313
<b>63</b>	<b>operationcaller.py</b>	<b>315</b>
63.1	class OperationModelCaller(Caller): . . . . .	315
<b>64</b>	<b>operationmemory.py</b>	<b>325</b>
64.1	class OperationMemory(object): . . . . .	325
<b>65</b>	<b>paramtablecaller.py</b>	<b>329</b>
65.1	def get_indices(tind, table, sim, depthind): . . . . .	329
65.2	def _limit_range(table, searchrange, values, i): . . . . .	329
65.3	def __init__(self, logger, logname): . . . . .	330
<b>66</b>	<b>predictioncaller.py</b>	<b>331</b>
66.1	class PredictionModelCaller(Caller): . . . . .	331
<b>67</b>	<b>predictionmemory.py</b>	<b>335</b>
67.1	class PredictionModelMemory(object): . . . . .	335
<b>68</b>	<b>aggregationarg.py</b>	<b>341</b>
68.1	class AggregationArg(object): . . . . .	341
<b>69</b>	<b>operationarg.py</b>	<b>343</b>
69.1	class OperationArg(object): . . . . .	343
<b>70</b>	<b>predictionarg.py</b>	<b>351</b>
70.1	class PredictionArg(object): . . . . .	351

<b>71 runnerconfig.py</b>	<b>357</b>
<b>72 logger.py</b>	<b>359</b>
<b>73 utils.py</b>	<b>361</b>
73.1 def secs2str(seconds): . . . . .	361
<b>74 Dev notes</b>	<b>363</b>
74.1 Testing . . . . .	363



System building modules (XML document to Python object conversion):

Data import, simulation, optimization and reporting descriptions are constructed with the builder package. It's responsible for parsing the xml documents, validating them and constructing the class instances of them, and storing both the xml documents, xml schema documents and Python class instances in a ZODB object database.

Each xml schema has a corresponding builder module and Python class responsible for parsing and validating the xml documents conforming the the schema document:



# XMLOBJECT.PY

## 1.1 class XmlObject(object):

Base class for SIMO XML definitions has two properties: xml and schema. The typedef schema will be included into the schema document if the schema has an include-element. When the schema is set, a parser is generated which is used to parse and validate the xml document when setting the xml property. The textual representations are returned schema and xml properties.

Successfully setting the xml property generates the object representation of the xml as well. This is taken care of by the subclass implementations of the `__xml_to_obj` method.

### 1.1.1 def \_\_init\_\_(self, typedefschema):

A class instance is initialized with a type definition schema:

```
>>> from simo.builder.xmlobject import XmlObject
>>> s = '''<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
...     <xs:simpleType name="ListOfIntDouble">
...         <xs:list>
...             <xs:simpleType>
...                 <xs:union memberTypes="xs:int xs:double"/>
...             </xs:simpleType>
...         </xs:list>
...     </xs:simpleType>
... </xs:schema>'''
>>> xo = XmlObject(s)
>>> xo.typedef_schema[:10]
'<xs:schema'
>>> xo.all_ok
False
```

### 1.1.2 def \_\_set\_schema(self, schematext):

Invalid schema leads to None value for schema property:

```
>>> s = '''<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
...     <xs:element name="test">
...         <xs:element name="notvalid"/>
...     </xs:element>
... </xs:schema>'''
>>> xo.schema = s # doctest: +ELLIPSIS
Traceback (most recent call last):
...

```

```
XMLSchemaParseError: Element '{http://www.w3.org/2001/XMLSchema}element': The content is not valid
>>> xo.schema
```

Valid schema document is stored as text into the schema property:

```
>>> s = '''<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
...     <xs:element name="test"/>
... </xs:schema>'''
>>> xo.schema = s
>>> xo.schema[:10]
'<xs:schema'
>>> xo.all_ok
False
```

A schema containing an include will inherit the contents of the typedef\_schema, the schema text will retain the include element, but the parser will have the types defined in the typedef document:

```
>>> s = '''<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
...     <xs:include schemaLocation="../dir/dir.2/Typedefs_SIMO.xsd"/>
...     <xs:element name="test" type="ListOfIntDouble"/>
... </xs:schema>'''
>>> xo.schema = s
>>> 'xs:include' in xo.schema
True
```

The include must be for the document Typedefs\_SIMO.xsd, other references will generate an error:

```
>>> snot = '''<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
...     <xs:include schemaLocation="some_other_def.xsd"/>
...     <xs:element name="test" type="ListOfIntDouble"/>
... </xs:schema>'''
>>> xo.schema = snot
Traceback (most recent call last):
...
XMLSchemaParseError: failed to load external entity "some_other_def.xsd"
>>> xo.schema = s
```

### 1.1.3 def \_\_set\_xml(self, xmldata):

With valid schema and xml a dictionary entry is created with the given name as key and the xml text as data:

```
>>> xo.xml = ('testdata', '<test>1 2 3</test>', None)
>>> xo.xml
{'testdata': '<test>1 2 3</test>'}
>>> xo.ok['testdata']
True
>>> xo.all_ok
True
```

With invalid xml, no entry is created in the xml dictionary:

```
>>> xo.xml = ('notvalid', '<nottest>test</nottest>', None)
Traceback (most recent call last):
...
ValueError: Element 'nottest': No matching global declaration available for the validation root.
>>> xo.xml = ('notvalideither', '<test>a b c</test>', None)
Traceback (most recent call last):
...
ValueError: Element 'test': 'a' is not a valid value of the local union type.
>>> xo.xml
```

```
{'testdata': '<test>1 2 3</test>'}
>>> xo.ok['testdata']
True
>>> xo.all_ok
True
```

Changing the schema invalidates the state of the typedef-schema-xml combination, unless the set schema is identical to the current one. However, the xml content is kept intact:

```
>>> s = xo.schema
>>> xo.schema = s
>>> xo.xml
{'testdata': '<test>1 2 3</test>'}
>>> xo.ok['testdata']
True
>>> s1 = '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"><xs:element name="testtoo"/></xs:schema>'
>>> xo.schema = s1
>>> xo.xml
{'testdata': '<test>1 2 3</test>'}
>>> xo.ok['testdata']
False
>>> xo.all_ok
False
```

#### 1.1.4 def xml\_to\_obj(self, root, lexicon):

Converts the XML representation to an object instance. For those XML documents that are tied to the SIMO lexicon definition; i.e., containing variable and data level definitions, the lexicon object instance is used to validate the XML document content against the lexicon content.

If the XML to object conversion is unsuccessful, the ok property for the XML name will be set to False as well as the all\_ok property. If it is successful, the ok property for the XML name will be set to True, and all the other entries in the ok property will be scanned for False values. If none are found, the all\_ok property will be set to True as well.

Each subclass of XmlObject will provide an implementation for this method.

#### 1.1.5 def reparse(self, lexicon=None):

Used to explicitly parse the schema and xml content again after typedef or schema changes:

```
>>> s = '''<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
...     <xs:element name="test" type="xs:string"/>
... </xs:schema>'''
>>> xo.schema = s
>>> tds = '''<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
...     <xs:simpleType name="doopidoo">
...         <xs:list itemType="xs:string"/>
...     </xs:simpleType>
... </xs:schema>'''
>>> xo.typedef_schema = tds
>>> xo.all_ok
False
>>> xo.ok['testdata']
False
>>> xo.reparse()
>>> xo.ok['testdata']
True
>>> xo.all_ok
```

```
True
>>> 'doopidoo' in xo.typedef_schema
True
>>> 'xs:string' in xo.schema
True
```

### 1.1.6 def level\_ind(self, level):

Checks whether a lexicon is defined, and whether the level can be found in the lexicon. Logs an error message if either of these fail and returns None, else returns the level index number for data matrix:

```
>>> xo.lexicon

>>> xo.errors
set([])
>>> xo.from_module = 'testmodule'
>>> xo.level_ind('comp_unit')

>>> xo.errors
set(["No lexicon set when validating level for testmodule 'testdata'"])
>>> class Lexicon(object):
...     def get_level_ind(self, level):
...         if level == 'comp_unit': return 1
...         else: return None
...     def get_variable_ind(self, level, var, active=False):
...         if level == 'comp_unit' and var == 'TS':
...             return (1, 1)
...         else:
...             return (None, None)
>>> xo.lexicon = Lexicon()
>>> xo.clear_warnings_and_errors()
>>> xo.level_ind('comp_unit')
1
>>> xo.errors
set([])
>>> xo.level_ind('non existing')

>>> xo.errors
set(["Level 'non existing' not found in lexicon for testmodule 'testdata'"])
```

### 1.1.7 def variable\_ind(self, level, var):

Checks whether a lexicon is defined, and whether the variable at the given level can be found in the lexicon. Logs an error message if either of these fail and returns (None, None), else returns a tuple of level index and variable index for the data matrix:

```
>>> xo.variable_ind('comp_unit', 'TS')
(1, 1)
>>> xo.variable_ind('comp_unit', 'X')
(None, None)
>>> xo.errors
set(["Variable 'X' not found at level 'comp_unit' in lexicon for testmodule 'testdata'", "Level '"])
```

### 1.1.8 def elem\_namespace(elemtag):

Returns the namespace of an XML element:

```
>>> from lxml import etree
>>> root = etree.fromstring('<root xmlns="http://www.simo-project.org/simo"><tag/></root>')
>>> xo.elem_namespace(root.tag)
'http://www.simo-project.org/simo'
```

### 1.1.9 def elem\_name(elemtag):

Returns the tag name of an XML element:

```
>>> xo.elem_name(root.tag)
'root'
```

### 1.1.10 def add\_warning(self, msg):

Appends the message into the list contained in the warnings attribute.

### 1.1.11 def add\_error(self, msg):

Adds the error message into the set contained in the errors attribute.

### 1.1.12 def clear\_warnings\_and\_errors(self):

```
>>> xo.clear_warnings_and_errors()
>>> xo.warnings
[]
>>> xo.errors
set([])
```





# OPERATIONMAPPING.PY

## 2.1 class OperationMappingDef(XmlObject):

### 2.1.1 def \_\_init\_\_(self, typedef, schema, xmldata):

Initializes the operation mapping by processing the schema and xml documents:

```
>>> from simo.builder.importers.operation2modelchains import \
...     OperationMappingDef
>>> tdf = open('../..simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../..simulator/xml/schemas/operation2modelchains.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = u'''<SIMO_operation_mapping xmlns="http://www.simo-project.org/simo"
... xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
... xsi:schemaLocation="http://www.simo-project.org/simo
... ../schemas/operation_mapping.xsd">
... <operation>
...     <from>
...         <name>thinning</name>
...     </from>
...     <to>
...         <model_chain>Calculate thinning limits</model_chain>
...         <model_chain>Forced low thinning</model_chain>
...     </to>
... </operation>
... <operation>
...     <from>
...         <name>first_thinning</name>
...     </from>
...     <to>
...         <model_chain>Calculate thinning limits</model_chain>
...     </to>
... </operation>
... </SIMO_operation_mapping>'''
>>> class Lexicon(object):
...     def level_ind(self, level):
...         return 1
...     def variable_ind(self, level, var, active=False):
...         return (None, None)
>>> omd = OperationMappingDef(typedef)
>>> omd.schema = schema
>>> try:
...     omd.xml = ('testxml', xml, Lexicon())
... except ValueError, e:
```

```
...     print e
>>> omd.errors
set([])
>>> omd.xml['testxml'][:23]
u'<SIMO_operation_mapping'
```

### 2.1.2 def xml\_to\_obj(self, root, lexicon):

```
>>> om = omd.obj['testxml']
>>> from pprint import pprint
>>> pprint(om.operation_mapping)
{'first_thinning': ['Calculate thinning limits'],
 'thinning': ['Calculate thinning limits', 'Forced low thinning']}
>>> om.forced_op_chains
set(['Calculate thinning limits', 'Forced low thinning'])
>>> omd.errors
set([])
>>> omd.warnings
[]
```

# TEXT2DATA.PY

## 3.1 class ConversionMappingDef(XmlObject):

### 3.1.1 def \_\_init\_\_(self, typedef, schema, xmldata):

Initializes the conversion mapping by processing the schema and xml documents:

```
>>> from simo.builder.importers.text2data import ConversionMappingDef
>>> tdf = open('../..simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../..simulator/xml/schemas/text2data.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = u'''<conversion_mapping xmlns="http://www.simo-project.org/simo"
...     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
...     xsi:schemaLocation="http://www.simo-project.org/simo
...     ../schemas/conversion_mapping.xsd">
...     <data_levels>
...         <level>
...             <name>comp_unit</name>
...             <id_rowpos>
...                 <pos>0</pos>
...             </id_rowpos>
...             <rowtype_rowpos>1</rowtype_rowpos>
...             <rowtype_value>1</rowtype_value>
...             <date_rowpos>7</date_rowpos>
...             <sublevel>
...                 <name>stratum</name>
...                 <id_rowpos>
...                     <pos>2</pos>
...                 </id_rowpos>
...                 <rowtype_rowpos>2</rowtype_rowpos>
...                 <rowtype_value>2</rowtype_value>
...                 <sublevel>
...                     <name>tree</name>
...                     <id_rowpos>
...                         <pos>1</pos>
...                     </id_rowpos>
...                     <link_id_rowpos>
...                         <level>comp_unit</level>
...                         <pos>1</pos><!--should be 0-->
...                     </link_id_rowpos>
...                     <rowtype_value>3</rowtype_value>
...                 </sublevel>
...             </sublevel>
...         </level>
```

```
... </data_levels>
... <none_value_indicator>' ' -1</none_value_indicator>
... <comment_prefix>! #</comment_prefix>
... <object_rejection>
...   <SIMO_variable>
...     <name>MAIN_GROUP</name>
...     <reject_criterion oper="in">
...       <enum>4 5 6 7 8</enum>
...     </reject_criterion>
...   </SIMO_variable>
... </object_rejection>
... <defaults>
...   <variable>
...     <level>comp_unit</level>
...     <name>USE_RESTRICTION_HARVEST</name>
...     <value>0</value>
...   </variable>
...   <variable>
...     <level>comp_unit</level>
...     <name>USE_RESTRICTION_SILVIC</name>
...     <value>0</value>
...   </variable>
... </defaults>
... <variable>
...   <name>
...     <from>Y-koordinaatti</from>
...     <to>LAT</to>
...   </name>
...   <row_type>1</row_type>
...   <row_position>9</row_position>
...   <from_datatype>double</from_datatype>
...   <none_to_value/>
...   <numerical>
...     <conversion_factor>0.001</conversion_factor>
...   </numerical>
... </variable>
... <variable>
...   <name>
...     <from>Pääryhmä</from>
...     <to>MAIN_GROUP</to>
...   </name>
...   <row_type>1</row_type>
...   <row_position>13</row_position>
...   <from_datatype>int</from_datatype>
...   <none_to_value/>
...   <categorical>
...     <value_mapping>
...       <value>
...         <from>1</from>
...         <to>1</to>
...       </value>
...       <value>
...         <from>2</from>
...         <to>2</to>
...       </value>
...     </value_mapping>
...   </categorical>
... </variable>
... <variable>
...   <name>
...     <from>Inventointipäivä</from>
...     <to>inventory_date</to>
```

```

...     </name>
...     <row_type>1</row_type>
...     <row_position>15</row_position>
...     <from_datatype>date</from_datatype>
...     <none_to_value/>
...     <date>
...         <epoch_year>gregorian</epoch_year>
...     </date>
... </variable>
... <variable>
...     <name>
...         <from>Tekstivariable</from>
...         <to>text_test</to>
...     </name>
...     <row_type>1</row_type>
...     <row_position>16</row_position>
...     <from_datatype>string</from_datatype>
...     <none_to_value/>
...     <text/>
... </variable>
... <variable>
...     <name>
...         <from>invalid</from>
...         <to>invalid</to>
...     </name>
...     <row_type>99</row_type>
...     <row_position>1</row_position>
...     <from_datatype>double</from_datatype>
...     <none_to_value/>
...     <numerical>
...         <conversion_factor>1</conversion_factor>
...     </numerical>
... </variable>
... </conversion_mapping>'''
>>> class Lexicon(object):
...     def get_level_ind(self, level):
...         if level=='comp_unit':
...             return 2
...         else:
...             return 3
...     def get_variable_ind(self, level, var, active=False):
...         if var == 'MAIN_GROUP':
...             return (1, 1)
...         elif var == 'LAT':
...             return (1, 2)
...         elif var == 'inventory_date':
...             return (1, 3)
...         elif var == 'text_test':
...             return (1, 4)
...         elif var == 'USE_RESTRICTION_HARVEST':
...             return (1, 5)
...         elif var == 'USE_RESTRICTION_SILVIC':
...             return (1, 6)
...         else:
...             return (None, None)
>>> cmd = ConversionMappingDef(typedef)
>>> cmd.schema = schema
>>> try:
...     cmd.xml = ('testxml', xml, Lexicon())
... except ValueError, e:
...     print e
errors in xml to object conversion

```

```
>>> cmd.errors # doctest: +NORMALIZE_WHITESPACE
set(["Link id position (1) is the same as data id position.
    This conflict will result in import errors and data corruption
    for conversion mapping 'testxml'",
    "invalid rowtype for variable 'invalid' for conversion mapping
    'testxml'",
    "Data level 'comp_unit' is not the child level of simulation level
    in lexicon for conversion mapping 'testxml'"])
>>> cmd.xml['testxml'][:19]
u'<conversion_mapping'
```

### 3.1.2 def xml\_to\_obj(self, root, lexicon):

```
>>> cm = cmd.obj['testxml']
>>> urs = cm.defaults['comp_unit']['USE_RESTRICTION_SILVIC']
>>> print urs['varind'], urs['value']
(1, 6) 0
>>> urh = cm.defaults['comp_unit']['USE_RESTRICTION_HARVEST']
>>> print urh['varind'], urh['value']
(1, 5) 0
>>> len(cm.defaults['comp_unit'].keys())
2
>>> len(cm.level_list)
3
>>> ldef = cm.level_list[0]
>>> ldef.level_name
'comp_unit'
>>> ldef.rowtype_value
[1]
>>> ldef.id_pos
[0]
>>> ldef.id_delimiter

>>> ldef.linkid_pos

>>> ldef.rowtype_pos
1
>>> ldef.new_object_row
1
>>> ldef.date_pos
7
>>> ldef = cm.level_list[2]
>>> ldef.linkid_pos
('comp_unit', 1)
>>> ldef.rowtype_pos

>>> cm.none_val
['', '-1']
>>> cm.comment_prefix
['!', '#']
>>> cm.object_rejection
{'MAIN_GROUP': [{'oper': 'in', 'criteria': [4, 5, 6, 7, 8]}]}
>>> cm.attributes
{'comp_unit': ['LAT', 'MAIN_GROUP', 'inventory_date', 'text_test']}
>>> numvar = cm.mapping[1][9]
>>> numvar.var_type
'numerical'
>>> isinstance(numvar.map, list)
True
>>> numvar.map[0].var_ind
```

```

(1, 2)
>>> numvar.map[0].to_var
'LAT'
>>> numvar.map[0].to_val

>>> numvar.map[0].conv_fact
0.001
>>> numvar.map[0].epoch_year

>>> numvar.from_data_type
'double'

>>> textvar = cm.mapping[1][16]
>>> textvar.var_type
'text'
>>> isinstance(textvar.map, list)
True
>>> textvar.map[0].var_ind
(1, 4)
>>> textvar.map[0].to_var
'text_test'
>>> textvar.map[0].to_val

>>> textvar.map[0].conv_fact

>>> textvar.map[0].epoch_year

>>> textvar.from_data_type
'string'

>>> catvar = cm.mapping[1][13]
>>> catvar.var_type
'categorical'
>>> isinstance(catvar.map, dict)
True
>>> catvar.map[1][0].var_ind
(1, 1)
>>> catvar.map[1][0].to_var
'MAIN_GROUP'
>>> catvar.map[1][0].to_val
1
>>> catvar.map[1][0].conv_fact

>>> catvar.map[1][0].epoch_year

>>> catvar.from_data_type
'int'
>>> datevar = cm.mapping[1][15]
>>> datevar.var_type
'date'
>>> isinstance(datevar.map, list)
True
>>> datevar.map[0].var_ind
(1, 3)
>>> datevar.map[0].to_val

>>> datevar.map[0].to_var
'inventory_date'
>>> datevar.map[0].conv_fact

>>> datevar.map[0].epoch_year
'gregorian'

```

```
>>> datevar.from_data_type
'date'
>>> cm.validator

>>> cmd.errors # doctest: +NORMALIZE_WHITESPACE
set(["Link id position (1) is the same as data id position.
    This conflict will result in import errors and data corruption
    for conversion mapping 'testxml'",
    "invalid rowtype for variable 'invalid' for conversion mapping
    'testxml'",
    "Data level 'comp_unit' is not the child level of simulation level
    in lexicon for conversion mapping 'testxml'"])
>>> cmd.warnings #doctest: +NORMALIZE_WHITESPACE
["Different row type indicator positions given for different data levels.
    This may cause data import errors for conversion mapping 'testxml'",
    "Different row type indicator positions given for different data levels.
    This may cause data import errors for conversion mapping 'testxml'"]
```

## 3.2 class ValueConv(Persistent):

Value conversion definition for imported data values.

## 3.3 class Mapping(Persistent):

Mapping definition for imported data; given for each variable and contains the variable type and the map between the original and imported values.

Attributes:

- `vartype`: categorical, numerical or date
- `map`: either a dictionary (categorical variable; key is the original value, value is a list of ValueConv objects) or list (numerical, date) of ValueConv objects
- `from_data_type`: the data type the data comes in as

## 3.4 class LevelDef(Persistent):

Data level definition for imported data.

Attributes:

- `levelname`: the name of the data level
- `rowtypevalue`: a list of row type values mapping to this level
- `idpos`: the index number(s) of data field(s) containing the id
- `delim`: delimiter for creating an id for object, if the id consists of multiple values
- `linkidpos`: the index number of the data field containing the id of the parent data object
- **newobjectrow**: the value of the row type values that triggers the creation of a new data object on the data level



### 3.5 class ConversionMapping(Persistent):

Data conversion definition for data import.

Attributes:

- name
- level\_list: list of LevelDef instances
- none\_val: list of original values treated as None values
- object\_rejection: dictionary of conditions of object rejection
- comment\_prefix: when an original row is treated as comment
- attributes: dictionary of level variable names by level name
- mapping: a dictionary by row type number and row position of Mapping instances
- validator: ConversionMappingDef instance used during the object construction to provide lexicon validation, set to None at the end

#### 3.5.1 def \_\_init\_\_(self, ns, root, validator):

Parses the XML data into a class instance

#### 3.5.2 def \_check\_rowtype\_pos(self):

Sanity check for the rowtypepos definitions which should be equal for all rowtypes. A warning is generated if they are not.

#### 3.5.3 def \_convert\_var(self, elem, ns):

Construct mapping definitions for single variable.

#### 3.5.4 def \_extract\_levels(self, ns, elem):

Parse level information from an XML element.

#### 3.5.5 def \_set\_rejection\_criteria(self, elem, ns):

Set criteria for rejecting input for an object.

```
def _create_categorical_mapping(self, elem, ns, rowtype, rowpos, vartype, fromdt, defval, level, toname):
    =====
```

Import data value to SIMO data value mappings for categorical variables.

#### 3.5.6 def \_extract\_id\_const(self, ns, elem):

Parse level id\_rowpos element information. This information is used for constructing object ids

#### 3.5.7 def \_parse\_linkids(self, ns, elem):

Parse link\_id\_rowpos element into link structure which links individual objects to a single top level object.



# OPERATIONCONVERSION.PY

## 4.1 class OperationConversionDef(XmlObject):

### 4.1.1 def \_\_init\_\_(self, typedef, schema, xmldata):

Initializes the output constraint by processing the schema and xml documents:

```
>>> from simo.builder.importers.operation2modelchains import \
...     OperationMappingDef
>>> tdf = open('../..simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../..simulator/xml/schemas/operation2modelchains.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = u'''<SIMO_operation_mapping
...     xmlns="http://www.simo-project.org/simo"
...     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
...     xsi:schemaLocation="http://www.simo-project.org/simo
...     ../schemas/operation2modelchains.xsd">
...     <operation>
...         <from>
...             <name>thinning</name>
...         </from>
...         <to>
...             <model_chain>Calculate thinning limits</model_chain>
...             <model_chain>Forced low thinning</model_chain>
...             <model_chain>Update Tree after forced thinning</model_chain>
...         </to>
...     </operation>
...     <operation>
...         <from>
...             <name>first_thinning</name>
...         </from>
...         <to>
...             <model_chain>Calculate thinning limits</model_chain>
...         </to>
...     </operation>
... </SIMO_operation_mapping>'''
>>> class Lexicon(object):
...     def get_level_ind(self, level):
...         return 1
...     def get_variable_ind(self, level, var, active=False):
...         return (None, None)
>>> omd = OperationMappingDef(typedef)
>>> omd.schema = schema
>>> try:
```

```
...     omd.xml = ('testxml', xml, Lexicon())
... except ValueError, e:
...     print e
>>> om = omd.obj['testxml']
>>> from simo.builder.importers.text2operation import OperationConversionDef
>>> tdf = open('../simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../simulator/xml/schemas/text2operation.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = u'''<operation_conversion
... xmlns="http://www.simo-project.org/simo"
... xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
... xsi:schemaLocation="http://www.simo-project.org/simo
... ../schemas/operation_conversion.xsd">
... <operation_level>comp_unit</operation_level>
... <none_value_indicator>0</none_value_indicator>
... <id_row_position>
...   <pos>0</pos>
... </id_row_position>
... <operation_type>
...   <row_position>2</row_position>
...   <mapping>
...     <type>
...       <from>1</from>
...       <to>harvest</to>
...     </type>
...     <type>
...       <from>2</from>
...       <to>silviculture</to>
...     </type>
...   </mapping>
... </operation_type>
... <operation_name>
...   <row_position>3</row_position>
...   <no_operation_value>99</no_operation_value>
...   <mapping>
...     <!-- Perushakkuut -->
...     <operation>
...       <type>1</type>
...       <from>2</from>
...       <to>first_thinning</to>
...     </operation>
...     <operation>
...       <type>1</type>
...       <from>21</from>
...       <to>first_thinning</to>
...       <parameters>
...         <parameter>
...           <name>TARGET_N</name>
...           <level>comp_unit</level>
...           <value>500</value>
...         </parameter>
...         <parameter>
...           <name>POINTLESS</name>
...           <level>comp_unit</level>
...           <value>-10</value>
...         </parameter>
...       </parameters>
...     </operation>
...   </mapping>
... </operation_name>
... </operation>'''
```

```

...         <type>2</type>
...         <from>970</from>
...         <to>pruning</to>
...     </operation>
... </mapping>
... </operation_name>
... <operation_timing>
...     <row_position>5</row_position>
...     <time_step>
...         <unit>year</unit>
...         <mapping>
...             <step>
...                 <from>1</from>
...                 <to>2</to>
...             </step>
...             <step>
...                 <from>2</from>
...                 <to>7</to>
...             </step>
...         </mapping>
...     </time_step>
... </operation_timing>
... <variable_mapping>
...     <variable>
...         <name>
...             <from>Uudistuspuulaji</from>
...             <to>REGEN_SP</to>
...         </name>
...         <level>comp_unit</level>
...         <row_position>13</row_position>
...         <from_datatype>int</from_datatype>
...         <none_to_value/>
...         <categorical>
...             <value_mapping>
...                 <value>
...                     <from>81 82</from>
...                     <to>1</to>
...                 </value>
...                 <value>
...                     <from>91</from>
...                     <to>2</to>
...                 </value>
...             </value_mapping>
...         </categorical>
...     </variable>
... </variable_mapping>
... </operation_conversion>'''
>>> class Lexicon(object):
...     def get_level_ind(self, level):
...         return 1
...     def get_variable_ind(self, level, var, active=False):
...         if var=='POINTLESS':
...             return (None, None)
...         else:
...             return (1, 1)
>>> ocd = OperationConversionDef(typedef)
>>> ocd.schema = schema
>>> try:
...     ocd.xml = ('testxml', xml, Lexicon(), om)
... except ValueError, e:
...     print e
errors in xml to object conversion

```

```
>>> ocd.errors # doctest: +NORMALIZE_WHITESPACE
set(["Variable 'POINTLESS' not found at level 'comp_unit' in lexicon for
    operation conversion 'testxml'",
    "Operation 'pruning' not defined in operation mapping for operation
    conversion 'testxml'"])
>>> ocd.xml['testxml'][:21]
u'<operation_conversion'
```

Conversion definition with no operation type definitions and timing type of date:

```
>>> xml2 = u'''<operation_conversion
... xmlns="http://www.simo-project.org/simo"
... xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
... xsi:schemaLocation="http://www.simo-project.org/simo
... ../schemas/operation_conversion.xsd">
... <operation_level>comp_unit</operation_level>
... <none_value_indicator>0</none_value_indicator>
... <id_row_position>
...   <pos>0 1</pos>
...   <delimiter>-</delimiter>
... </id_row_position>
... <operation_name>
...   <row_position>3</row_position>
...   <no_operation_value>99</no_operation_value>
...   <mapping>
...     <operation>
...       <from>2</from>
...       <to>first_thinning</to>
...     </operation>
...   </mapping>
... </operation_name>
... <operation_timing>
...   <row_position>5</row_position>
...   <date/>
... </operation_timing>
... </operation_conversion>'''
>>> try:
...     ocd.xml = ('testxml2', xml2, Lexicon(), om)
... except ValueError, e:
...     print e
```

#### 4.1.2 def xml\_to\_obj(self, root, lexicon):

Note that the timing type of 'date' has no implementation yet:

```
>>> oc = ocd.obj['testxml']
>>> oc.validator
>>> oc.op2mc.operation_mapping['first_thinning']
['Calculate thinning limits']
>>> oc.operation_conv.operation_level
'comp_unit'
>>> oc.operation_conv.no_operation_indicator
['99']
>>> oc.operation_conv.none_value_indicator
['0']
>>> oc.operation_conv.comment_prefix
>>> oc.operation_conv.id_rowpos
[0]
>>> oc.operation_conv.type_rowpos
2
```

```

>>> oc.operation_conv.operation_rowpos
3
>>> oc.operation_conv.timing_rowpos
5
>>> t = oc.operation_conv.timing
>>> t.type
'step'
>>> t.step_conversion['2']
6
>>> t.step_unit
'year'
>>> oc.operation_conv.types[None]
>>> oc.operation_conv.types['1']
'harvest'
>>> oc.operation_conv.types['2']
'silviculture'
>>> keys = oc.operation_conv.operations.keys()
>>> for key in keys:
...     print 'Key:', key
...     print oc.operation_conv.operations[key].type
...     print oc.operation_conv.operations[key].from_name
...     print oc.operation_conv.operations[key].to_name
Key: ('1', '2')
harvest
2
first_thinning
Key: ('1', '21')
harvest
21
first_thinning
>>> params = oc.operation_conv.operations[('1', '21')].parameters
>>> len(params)
1
>>> params[0].value
500.0
>>> params[0].variable
'TARGET_N'
>>> params[0].level
'comp_unit'
>>> params[0].variable_ind
(1, 1)
>>> oc.operation_conv.timing.type
'step'
>>> oc.operation_conv.timing.step_conversion
{'1': 1, '2': 6}
>>> oc.operation_conv.timing.step_unit
'year'
>>> mapping = oc.operation_conv.variables[13]
>>> mapping.var_type
'categorical'
>>> vc = mapping.map[81][0]
>>> vc.level
'comp_unit'
>>> vc.to_var
'REGEN_SP'
>>> vc.to_val
1
>>> ocd.warnings
[]

```

The 'no operation types' and 'date timing' variant:

```
>>> oc = ocd.obj['testxml2']
>>> oc.operation_conv.id_rowpos
[0, 1]
>>> oc.operation_conv.id_delim
'_'
>>> t = oc.operation_conv.timing
>>> t.type
'date'
>>> oc.operation_conv.types
{None: None}
```



# LEXICON.PY

```
>>> from simo.builder.lexicon.lexicon import LexiconDef
>>> tdf = open('../../simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../../simulator/xml/schemas/lexicon.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = '''<rootlevel xmlns="http://www.simo-project.org/simo"
...     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
...     xsi:schemaLocation="http://www.simo-project.org/simo
...     ../schemas/lexicon.xsd">
...     <name>simulation</name>
...     <num_vars>
...         <variable>
...             <name>DIAM_CLASS_WIDTH</name>
...             <description>Width of a single class...</description>
...         </variable>
...     </num_vars>
...     <cat_vars/>
...     <sublevels>
...         <sublevel>
...             <name>comp_unit</name>
...             <type>static</type>
...             <num_vars>
...                 <variable>
...                     <name>BA</name>
...                     <description>Basal area...</description>
...                 </variable>
...             </num_vars>
...             <cat_vars>
...                 <variable>
...                     <name>SC</name>
...                     <description>Site class...</description>
...                     <values>
...                         <enum>
...                             <value>1</value>
...                             <description>Very good</description>
...                         </enum>
...                     </values>
...                 </variable>
...             </cat_vars>
...             <text_vars>
...                 <variable>
...                     <name>Stand label</name>
...                     <description>Id text for the stand</description>
...                 </variable>
...             </text_vars>
...         </sublevels>
```

```
...         <sublevel>
...             <name>stratum</name>
...             <type>dynamic</type>
...             <num_vars>
...                 <variable>
...                     <name>N</name>
...                     <description>Number of stems...</description>
...                 </variable>
...             </num_vars>
...             <cat_vars>
...                 <variable>
...                     <name>SP</name>
...                     <description>Tree species...</description>
...                     <values>
...                         <enum>
...                             <value>1</value>
...                             <description>Pine</description>
...                         </enum>
...                         <enum>
...                             <value>2</value>
...                             <description>Spruce</description>
...                         </enum>
...                     </values>
...                 </variable>
...             </cat_vars>
...             <sublevels>
...                 <sublevel>
...                     <name>tree</name>
...                     <type>dynamic</type>
...                     <num_vars>
...                         <variable>
...                             <name>d</name>
...                             <description>Diameter...</description>
...                         </variable>
...                     </num_vars>
...                     <cat_vars/>
...                 </sublevel>
...             </sublevels>
...         </sublevel>
...     </sublevels>
... </sublevel>
... <sublevel>
...     <name>sample_plot</name>
...     <type>static</type>
...     <num_vars>
...         <variable>
...             <name>BA</name>
...             <description>Basal area...</description>
...         </variable>
...     </num_vars>
...     <cat_vars/>
...     <sublevels>
...         <sublevel>
...             <name>sample_tree</name>
...             <type>dynamic</type>
...             <num_vars>
...                 <variable>
...                     <name>h</name>
...                     <description>Height...</description>
...                 </variable>
...             </num_vars>
...             <cat_vars/>
...         </sublevel>
...     </sublevels>
... </sublevel>
```

```

...         </sublevel>
...         </sublevels>
...     </sublevel>
... </sublevels>
... </rootlevel>'''
>>> ldef = LexiconDef(typedef)
>>> ldef.schema = schema
>>> try:
...     ldef.xml = xml
... except ValueError, e:
...     print e
xml name and xml content must be passed
>>> ldef.xml = ('Test lexicon', xml)
>>> ldef.xml['Test lexicon'][0:10]
'<rootlevel'

```

## 5.1 class Lexicon(object):

Class responsible for providing data level and attribute validation for all the other simobuilder classes. The validation is based on the lexicon XML-document, which should define all the data levels, their hierarchy and all the allowed attributes for each level.

### Properties:

- `variable_ind`: get variable index dictionary (name to index)
- `level_ind`: get level index dictionary (name to index)
- `variable_name`: get variable name dictionary (index to name)
- `level_name`: get level name dictionary (index to name)
- `hierarchy`: get hierarchy dictionary
- `variables`: get variable checklist
- `models`: get and set models in model checklist

### 5.1.1 def \_\_init\_\_(self):

The main task of Lexicon is to keep track of the correspondence between the names of levels and attributes and their indices in the data matrix. It also keeps record of the hierarchy between the data levels.

The lexicon instance has two internal lexicon representations; the master lexicon which holds the complete SIMO main lexicon containing all the data levels and their attributes, and the instance lexicon which is limited to the data levels and attributes used in the particular simulation instance.

The instance lexicon is constructed stepwise when model chains etc. are being processed.

Changing the master lexicon will trigger object validation for all other classes tied to the lexicon:

```

>>> lex = ldef.obj["Test lexicon"]
>>> for x, y in lex.level_ind.items(): print x, y # doctest: +ELLIPSIS
tree 3
simulation 0
sample_tree 5
sample_plot 4
stratum 2
comp_unit 1

>>> for x, y in lex.variable_ind.items():
...     print x, y # doctest: +ELLIPSIS

```

```
tree {'d': None}
simulation {'DIAM_CLASS_WIDTH': None}
sample_tree {'h': None}
sample_plot {'BA': None}
stratum {'SP': None, 'N': None}
comp_unit {'SC': None, 'Stand label': None, 'BA': None}

>>> for x, y in lex.level_name.items():
...     print x, y # doctest: +ELLIPSIS
0 simulation
1 comp_unit
2 stratum
3 tree
4 sample_plot
5 sample_tree

>>> for x, y in lex.variable_name.items():
...     print x, y # doctest: +ELLIPSIS
0 {}
1 {}
2 {}
3 {}
4 {}
5 {}

>>> for x, y in lex.hierarchy.items():
...     print x, y # doctest: +ELLIPSIS
0 {'ordinal': 0, 'lineage': set([0, 1]), 'children': [1, 4], 'parent': None}
1 {'ordinal': 1, 'lineage': set([0]), 'children': [2], 'parent': 0}
2 {'ordinal': 2, 'lineage': set([0]), 'children': [3], 'parent': 1}
3 {'ordinal': 3, 'lineage': set([0]), 'children': None, 'parent': 2}
4 {'ordinal': 1, 'lineage': set([1]), 'children': [5], 'parent': 0}
5 {'ordinal': 2, 'lineage': set([1]), 'children': None, 'parent': 4}

>>> for x, y in lex.active_variables.items():
...     print x, y # doctest: +ELLIPSIS
tree {'d': False}
simulation {'DIAM_CLASS_WIDTH': False}
sample_tree {'h': False}
sample_plot {'BA': False}
stratum {'SP': False, 'N': False}
comp_unit {'SC': False, 'Stand label': False, 'BA': False}
>>> lex.get_variable_ind('tree', 'd')
(3, 0)
>>> lex.active_variables['tree']
{'d': True}
>>> lex.get_variable_ind('simulation', 'DIAM_CLASS_WIDTH')
(0, 0)
>>> lex.get_variable_ind('comp_unit', 'SC')
(1, 0)

>>> cd = lex.get_content_def()
>>> cd[('simulation', 0)]
[('DIAM_CLASS_WIDTH', 0)]
>>> cd[('comp_unit', 1)]
[('SC', 0)]

>>> lex.models
{}
>>> lex.add_model('prediction', 'testmodel')
>>> lex.add_model('aggregation', 'testmodel')
>>> lex.add_model('prediction', 'testmodel')
```

```

>>> lex.add_model('operation', 'testmodel1')
0
>>> lex.add_model('operation', 'testmodel2')
1
>>> lex.add_model('operation', 'testmodel2')
1
>>> lex.models['prediction']
set(['testmodel'])
>>> lex.models['aggregation']
set(['testmodel'])
>>> lex.models['operation']
set(['testmodel2', 'testmodel1'])

>>> l = lex.levels
>>> t = l['tree']
>>> var = t.variables['d']
>>> var.desc
'Diameter...'
>>> s = l['stratum']
>>> s.categorical
set(['SP'])
>>> s.numerical
set(['N'])
>>> sp = s.variables['SP']
>>> sp.name
'SP'
>>> sp.unit
>>> sp.maximum
>>> sp.minimum
>>> sp.desc
'Tree species...'
>>> sp.values
{1.0: 'Pine', 2.0: 'Spruce'}

>>> lex.reset_active_variables_and_models()
>>> lex.active_variables['tree']
{'d': False}
>>> lex.models
{}
>>> lex.operation_ind
{}

```

### 5.1.2 def \_parse\_level(self, elem, levelind, ordinal, line):

Parse a single level and store level information into lexicon.

### 5.1.3 def \_store\_level(self, level, levelind, ordinal, lineage):

Store variable indices, level indices etc. from a single level.

### 5.1.4 def get\_level\_ind(self, level):

Validate level against lexicon:

```

>>> lex.get_level_ind('comp_unit')
1
>>> lex.get_level_ind('compunit')

```

### 5.1.5 def get\_variable\_ind(self, level, variable):

Validate variable name and level against lexicon; create a new ind if one doesn't already exist:

```
>>> lex.get_variable_ind('simulation', 'DIAM_CLASS_WIDTH')
(0, 0)
>>> lex.get_variable_ind('stratum', 'SP')
(2, 0)
>>> lex.get_variable_ind('stratum', 'N', True)
(None, None)
>>> lex.get_variable_ind('stratum', 'N')
(2, 1)
>>> lex.get_variable_ind('stratum', 'SP', True)
(2, 0)
>>> lex.get_variable_ind('no-level', 'SP')
(None, None)
>>> lex.get_variable_ind('stratum', 'no-variable')
(None, None)
```

### 5.1.6 def get\_level\_name(self, level):

Get level name:

```
>>> lex.get_level_name(2)
'stratum'
>>> lex.get_level_name(10)
```

### 5.1.7 def get\_variable\_name(self, level, variable):

Get variable name:

```
>>> lex.get_variable_name(1, 0)
'SC'
>>> lex.get_variable_name(10, 1)
>>> lex.get_variable_name(1, 10)
```

### 5.1.8 def is\_parent(self, parentcandidate, childcandidate):

Check whether parentcandidate is a parent level of childcandidate level:

```
>>> lex.is_parent(0, 0)
False
>>> lex.is_parent(0, 1)
True
>>> lex.is_parent(0, 3)
True
>>> lex.is_parent(0, 5)
True
>>> lex.is_parent(1, 3)
True
>>> lex.is_parent(1, 5)
False
```

### 5.1.9 def is\_child(self, childcandidate, parentcandidate):

Check whether childcandidate is a child level of parentcandidate level:

```
>>> lex.is_child(0, 0)
False
>>> lex.is_child(1, 0)
True
>>> lex.is_child(3, 0)
True
>>> lex.is_child(5, 0)
True
>>> lex.is_child(3, 1)
True
>>> lex.is_child(5, 1)
False
```

### 5.1.10 Final checks:

```
>>> for x, y in lex.variable_ind.items():
...     print x, y # doctest: +ELLIPSIS
tree {'d': 0}
simulation {'DIAM_CLASS_WIDTH': 0}
sample_tree {'h': None}
sample_plot {'BA': None}
stratum {'SP': 0, 'N': 1}
comp_unit {'SC': 0, 'Stand label': None, 'BA': None}

>>> for x, y in lex.variable_name.items():
...     print x, y # doctest: +ELLIPSIS
0 {0: 'DIAM_CLASS_WIDTH'}
1 {0: 'SC'}
2 {0: 'SP', 1: 'N'}
3 {0: 'd'}
4 {}
5 {}
```





# LEXICONLEVEL.PY

## 6.1 class `LexiconLevel(object)`:

Class for defining a single level of lexicon

### 6.1.1 `def __init__(self, ns, elem, validator)`:

Parse an xml snippet containing level definition and construct a level instance.:

```
>>> from simo.builder.lexicon.lexiconlevel import LexiconLevel
>>> from lxml import etree
>>> xml = u'''<sublevel>
...     <name>comp_unit</name>
...     <type>static</type>
...     <num_vars>
...         <variable>
...             <name>BA</name>
...             <description>Basal area</description>
...         </variable>
...     </num_vars>
...     <cat_vars>
...         <variable>
...             <name>SC</name>
...             <description>Site class</description>
...             <values>
...                 <enum>
...                     <value>1</value>
...                     <description>Pine</description>
...                 </enum>
...                 <enum>
...                     <value>2</value>
...                     <description>Spruce</description>
...                 </enum>
...             </values>
...         </variable>
...     </cat_vars>
...     <text_vars>
...         <variable>
...             <name>StandLabel</name>
...             <description>Id text for the stand</description>
...         </variable>
...     </text_vars>
... </sublevel>'''
>>> elem = etree.fromstring(xml)
>>> level = LexiconLevel('', elem, None)
>>> level.name
```

```
'comp_unit'
>>> level.type
'static'
>>> level.numerical # doctest: +ELLIPSIS
set(['BA'])
>>> level.categorical # doctest: +ELLIPSIS
set(['SC'])
>>> level.textual
set(['StandLabel'])
>>> var = level.variables['SC']
>>> var.name
'SC'
>>> var.minimum
>>> var.maximum
>>> var.values
{1.0: 'Pine', 2.0: 'Spruce'}
>>> var.desc
'Site class'
```

# LEXICONVARIABLE.PY

## 7.1 class LexiconVariable(object):

Class for defining a single variable in the lexicon

### 7.1.1 def \_\_init\_\_(self, ns, elem, validator):

Parse an xml snippet containing variable definition and construct a variable instance:

```
>>> from simo.builder.lexicon.lexiconvariable import LexiconVariable
>>> from lxml import etree
>>> xml = u'''<variable>
...     <name>assortment</name>
...     <unit>NA</unit>
...     <min_value>1</min_value>
...     <max_value>2</max_value>
...     <description>Timber assortment</description>
...     <values>
...         <enum>
...             <value>1</value>
...             <description>Log</description>
...         </enum>
...         <enum>
...             <value>2</value>
...             <description>Pulp</description>
...         </enum>
...     </values>
... </variable>'''
>>> elem = etree.fromstring(xml)
>>> lvar = LexiconVariable('', elem, None)
>>> lvar.name
'assortment'
>>> lvar.unit
'NA'
>>> lvar.minimum
1.0
>>> lvar.maximum
2.0
>>> lvar.desc
'Timber assortment'
>>> lvar.values
{1.0: 'Log', 2.0: 'Pulp'}
```



# AGGREGATIONMODEL.PY

```
>>> class Lexicon(object):
...     def __init__(self):
...         self.models = {}
...     def get_variable_ind(self, level, variable, active=False):
...         if level == 'comp_unit': return (0,1)
...         elif level == 'stratum': return (1,1)
...         else: return (2,1)
...     def get_level_ind(self, level):
...         if level == 'comp_unit': return 0
...         elif level == 'stratum': return 1
...         else: return 2
...     def get_variable_name(self, level, variable):
...         return 'VARNAME'
...     def get_level_name(self, level):
...         return 'LEVEL%d' % level
...     def is_parent(self, child, parent):
...         return False
...     def add_model(self, mtype, mname):
...         if mtype not in self.models:
...             self.models[mtype] = set()
...             self.models[mtype].add(mname)
>>> from simo.builder.modelbase.modelbase import ModelbaseDef
>>> tdf = open('../simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../simulator/xml/schemas/aggregation_modelbase.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = u'''<aggregation_base xmlns="http://www.simo-project.org/simo"
...         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...             <model>
...                 <name>sum</name>
...                 <description>desc...</description>
...                 <implemented_at>simoaggr.py</implemented_at>
...                 <vector_implementation>true</vector_implementation>
...             </model>
...         </aggregation_base>'''
>>> mb = ModelbaseDef(typedef)
>>> try:
...     mb.schema = schema
... except ValueError, e:
...     print e
schema type and schema text must be passed
>>> mb.schema = ('aggregation', schema)
>>> try:
...     mb.xml = ('aggregation', 'testxml', xml)
... except ValueError, e:
...     print e # doctest: +NORMALIZE_WHITESPACE
```

```
xml name, xml content, lexicon instance, model directory path and model
type must be passed
>>> mb.xml = ('testxml', xml, Lexicon(), '', 'aggregation')
>>> mb.xml['aggregation']['testxml'][:17]
u'<aggregation_base'
```

## 8.1 class AggregationModel(object):

Aggregation model

Properties:

- type: get model type 'aggregation'
- name: get model name as string
- dirs: get model library directory listing
- function: get model function object
- library: get model library object
- wrapper: get model library wrapper object

### 8.1.1 def \_\_init\_\_(self, ns, elem, validator, dirs):

Parses the XML data into model instance attributes:

```
>>> am = mb.obj['aggregation']['testxml']['sum']
>>> am.name
'sum'
>>> am.language
'python'
```

## 8.2 class AggregationModelParam(object):

Aggregation model parameters

Attributes:

- target\_ind: model target indice (level, variable)
- through\_level: model aggregation level
- operands: aggregation model operands
- condition: aggregation model condition
- model: aggregation model object

### 8.2.1 def \_\_init\_\_(self, ns, elem, validator, model):

Extract aggregation model parameters defined in the model chain

```
>>> from minimock import Mock
>>> execfile('builder/modelbase/test/mocktask.py')
>>> from lxml import etree
>>> from simo.builder.modelbase.aggregationmodel import AggregationModelParam
>>> xml = u'''<aggregation>
...     <target>BA</target>
```

```

...         <level>stratum</level>
...         <through_level>stratum</through_level>
...         <operands>
...             <operand>
...                 <variable>BA_u</variable>
...                 <level>stratum</level>
...             </operand>
...             <operand>
...                 <variable>B_BA</variable>
...                 <level>stratum</level>
...             </operand>
...         </operands>
...     </aggregation>'''
>>> elem = etree.fromstring(xml)
>>> ns = ''
>>> model = Mock('AggregationModel')
>>> model.vector_implementation = False
>>> mp = AggregationModelParam(ns, elem, task, model)
>>> mp.target_ind
(2, 1)
>>> mp.through_level
2
>>> len(mp.operands)
2

```

Extract aggregation parameters with a condition

```

>>> xml = u'''<aggregation>
...     <target>BA_L</target>
...     <through_level>comp_unit</through_level>
...     <operands>
...         <operand>
...             <variable>ba_m2</variable>
...             <level>tree</level>
...         </operand>
...         <operand>
...             <variable>n</variable>
...             <level>tree</level>
...         </operand>
...     </operands>
...     <condition>tree:d gt tree:d</condition>
... </aggregation>'''
>>> elem = etree.fromstring(xml)
>>> task.chain_level = 'tree'
>>> mp = AggregationModelParam(ns, elem, task, model)
>>> mp.target_ind
(3, 1)
>>> mp.through_level
1
>>> len(mp.operands)
2
>>> mp.condition # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
[('variable', (3, 1, True)), ('variable', (3, 1, True)),
 ('eq', <function gte at ...>)]

```

Extract aggregation model with target vector, i.e. multiple targets instead of a single variable:

```

>>> model.vector_implementation = True
>>> xml = u'''<aggregation>
...     <target> BA N D_gM
...                 H_gM
... </target>

```

```
...         <level>comp_unit</level>
...         <operands>
...             <operand>
...                 <value> 0    0    0    0 </value>
...             </operand>
...         </operands>
...     </aggregation>'''
>>> elem = etree.fromstring(xml)
>>> task.chain_level = 'comp_unit'
>>> mp = AggregationModelParam(ns, elem, task, model)
>>> mp.target_ind
(1, [1, 1, 1, 1])
>>> mp.scalar
False
>>> task.validator.errors
set([])
```

Extract aggregation parameters with invalid values

```
>>> xml = u'''<aggregation>
...     <target>BA</target>
...     <level>stratum</level>
...     <through_level>stratum</through_level>
...     <operands>
...         <operand>
...             <variable>BA_u</variable>
...             <level>stratum</level>
...         </operand>
...         <operand>
...             <variable>B_BA</variable>
...             <level>tree</level>
...         </operand>
...     </operands>
... </aggregation>'''
>>> elem = etree.fromstring(xml)
>>> task.chain_level = 'stratum'
>>> mp = AggregationModelParam(ns, elem, task, model)
>>> task.validator.errors
set(["Aggregation operands at different levels for task 'test task'"])

>>> xml = u'''<aggregation>
...     <target>JUST_THINNED</target>
...     <operands>
...         <operand>
...             <value>0</value>
...         </operand>
...     </operands>
... </aggregation>'''
>>> elem = etree.fromstring(xml)
>>> task.chain_level = 'stratum'
>>> mp = AggregationModelParam(ns, elem, task, model)
>>> oper = mp.operands[0]
>>> oper.value
0.0
>>> oper.type
'value'
```

Process aggregation definition for model without vector implementation:

```
>>> xml = u'''<aggregation>
...     <target> BA N  D_gM H_gM </target>
...     <level>comp_unit</level>
```



```

...         <operands>
...             <operand>
...                 <value> 0    0    0    0</value>
...             </operand>
...         </operands>
...     </aggregation>'''
>>> elem = etree.fromstring(xml)
>>> task.chain_level = 'comp_unit'
>>> task.validator.errors = set([])
>>> model.vector_implementation = False
>>> model.name = 'mock_model'
>>> mp = AggregationModelParam(ns, elem, task, model)
>>> task.validator.errors
set(["Multiple targets not allowed for aggregation model 'mock_model', model lacks vector implementation"])

```

Process aggregation definition with unequal number of target variables and operands:

```

>>> xml = u'''<aggregation>
...     <target> BA N   D_gM H_gM </target>
...     <level>comp_unit</level>
...     <operands>
...         <operand>
...             <value> 0    0    0    0    0</value>
...         </operand>
...     </operands>
... </aggregation>'''
>>> elem = etree.fromstring(xml)
>>> task.chain_level = 'comp_unit'
>>> task.validator.errors = set([])
>>> model.vector_implementation = True
>>> mp = AggregationModelParam(ns, elem, task, model)
>>> task.validator.errors
set(["Unequal value vector and target variable vector lengths!"])

```

Process invalid aggregation definition with mixed scalar and vector definitions:

```

>>> xml = u'''<aggregation>
...     <target> BA N   D_gM H_gM </target>
...     <level>comp_unit</level>
...     <operands>
...         <operand>
...             <value>0</value>
...         </operand>
...     </operands>
... </aggregation>'''
>>> elem = etree.fromstring(xml)
>>> task.chain_level = 'comp_unit'
>>> task.validator.errors = set([])
>>> mp = AggregationModelParam(ns, elem, task, model)
>>> task.validator.errors
set(["Target variable and operands must be either both scalar or both vector!"])

```

## 8.3 class AggregationModelOperand(object):

Aggregation model operand

Attributes:

- type: u'variable' or u'value'

### 8.3.1 def \_\_init\_\_(self, ns, elem, validator):

Extract aggregation model operand defined in a model chain:

```
>>> from simo.builder.modelbase.aggregationmodel import AggregationModelOperand
>>> xml = u'''<operand>
...     <variable default="0.0">ba</variable>
...     <weight default="1.0">n</weight>
...     <level>tree</level>
... </operand>'''
>>> elem = etree.fromstring(xml)
>>> op = AggregationModelOperand(ns, elem, task.validator)
>>> op.type
'variable'
>>> op.variable
(3, 1)
>>> op.weight
(3, 1)
>>> op.default_value
0.0
>>> op.default_weight
1.0
```

Extract aggregation model operand with variable vector:

```
>>> xml = u'''<operand>
...     <variable>d h n sp</variable>
...     <weight> ba ba ba ba</weight>
...     <level>tree</level>
... </operand>'''
>>> elem = etree.fromstring(xml)
>>> op = AggregationModelOperand(ns, elem, task.validator, False, 4)
>>> op.type
'variable'
>>> op.level
'tree'
>>> op.variable
(3, [1, 1, 1, 1])
>>> op.weight
(3, [1, 1, 1, 1])
```

Extract aggregation model operand with value vector:

```
>>> xml = u'''<operand>
...     <value> 1 2
...     3 4 </value>
... </operand>'''
>>> elem = etree.fromstring(xml)
>>> task.validator.errors = set([])
>>> op = AggregationModelOperand(ns, elem, task.validator, False, 4)
>>> op.type
'value'
>>> op.value
array([ 1.,  2.,  3.,  4.])
>>> task.validator.errors
set([])
```

Extract variable vector operand with inequal number of variable and weight variables:

```
>>> xml = u'''<operand>
...     <variable>d h n sp</variable>
...     <weight>ba ba ba</weight>
```

```
...         <level>tree</level>
...     </operand>'''
>>> elem = etree.fromstring(xml)
>>> task.validator.errors = set([])
>>> op = AggregationModelOperand(ns, elem, task.validator, False, 4)
>>> task.validator.errors
set(['variable vector and weight vector must contain equal number of variables!'])
```

Extract aggregation model operand with non-numerical values in value vector:

```
>>> xml = u'''<operand>
...     <value>1 2 3 X</value>
... </operand>'''
>>> elem = etree.fromstring(xml)
>>> task.validator.errors = set([])
>>> op = AggregationModelOperand(ns, elem, task.validator, False, 4)
>>> task.validator.errors
set(['Invalid value vector '1 2 3 X'])
```



# CASHFLOWMODEL.PY

```
>>> from simo.builder.modelbase.modelbase import ModelbaseDef
>>> tdf = open('../..simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../..simulator/xml/schemas/cash_flow_modelbase.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = u'''<cash_flow_modelbase xmlns="http://www.simo-project.org/simo"
...     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
...     xsi:schemaLocation="http://www.simo-project.org/simo ../schemas/cash_flow_modelbase.xsd">
...     <cash_flow_model>
...         <name>random_prices</name>
...         <implemented_at>cashflowmodels.py</implemented_at>
...         <implemented_in>python</implemented_in>
...         <reference_values>
...             <cash_flow_table>timber_prices</cash_flow_table>
...         </reference_values>
...         <variables>
...             <variable>
...                 <name>Testname</name>
...                 <level>Testlevel</level>
...             </variable>
...         </variables>
...     </cash_flow_model>
... </cash_flow_modelbase>
... '''
>>> class Lexicon(object):
...     def __init__(self):
...         self.models = {}
...     def get_variable_ind(self, level, variable, active=False):
...         return (1, 1)
...     def get_level_ind(self, level):
...         return 1
...     def add_model(self, mtype, mname):
...         if mtype not in self.models:
...             self.models[mtype] = set()
...         self.models[mtype].add(mname)
>>> mb = ModelbaseDef(typedef)
>>> mb.schema = ('cash_flow', schema)
>>> mb.xml = ('testxml', xml, Lexicon(), '', 'cash_flow')
>>> mb.xml['cash_flow']['testxml'][:16]
u'<cash_flow_model'
```

## 9.1 class CashFlowModel(object):

cash flow model

Attributes:

-name -at: the model library name in which the model implementation is found -variables: a list of variable definitions

### 9.1.1 def \_\_init\_\_(self, ns, elem):

Parses the XML data into model instance attributes:

```
>>> cf = mb.obj['cash_flow']['testxml']['random_prices']
>>> cf.name
'random_prices'
>>> cf.language
'python'
>>> cf.variables
[{'name': 'Testname', 'level': 'Testlevel'}]
```

## 9.2 class CashFlowModelParam(object):

Variable values for cash flow model, defined in modelchain.

### 9.2.1 def \_\_init\_\_(self, ns, elem, lexicon, model):

# CASHFLOWTABLE.PY

```
>>> from simo.builder.modelbase.modelbase import ModelbaseDef
>>> tdf = open('../../simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../../simulator/xml/schemas/cash_flow_table.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = u'''<cash_flow_tables xmlns="http://www.simo-project.org/simo"
...     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...     <cash_flow_table name="timber_prices" logged="true">
...         <base>
...             <classifiers>
...                 <classifier>
...                     <variable>PRICE_REGION</variable>
...                     <level>comp_unit</level>
...                 </classifier>
...                 <classifier to_db="true">
...                     <variable>assortment</variable>
...                     <level>within-operation</level>
...                     <labels>
...                         <label>
...                             <value>1</value>
...                             <name>log</name>
...                         </label>
...                         <label>
...                             <value>2</value>
...                             <name>pulp</name>
...                         </label>
...                     </labels>
...                 </classifier>
...             </classifiers>
...             <tables>
...                 <table>
...                     <cash_flow>1 1 1 160 370 55</cash_flow>
...                     <cash_flow>2 1 1 160 400 56</cash_flow>
...                 </table>
...             </tables>
...         </base>
...     </cash_flow_table>
... </cash_flow_tables>'''
>>> mb = ModelbaseDef(typedef)
>>> mb.schema = ('cash_flow_table', schema)
>>> try:
...     mb.xml = ('testxml', xml, None, '', 'cash_flow_table')
... except ValueError, e:
...     e.message
'errors in xml to object conversion'
>>> mb.errors # doctest: +NORMALIZE_WHITESPACE
```

```
set(["No lexicon set when validating variable for cash flow table
collection 'testxml'",
    "No lexicon set when adding model for cash flow table
collection 'testxml'"])
>>> class Lexicon(object):
...     def __init__(self):
...         self.models = {}
...     def get_variable_ind(self, level, var, active=False):
...         return (1, 1)
...     def get_level_ind(self, level):
...         return 1
...     def add_model(self, mtype, mname):
...         if mtype not in self.models:
...             self.models[mtype] = set()
...             self.models[mtype].add(mname)
>>> mb.xml = ('testxml', xml, Lexicon(), '', 'cash_flow_table')
>>> mb.xml['cash_flow_table']['testxml'][:17]
u'<cash_flow_tables'
>>> mb.errors
set([])
>>> mb.all_ok
True
```

## 10.1 class CashFlowTable(object):

Class for cash flow tables used in operations:

```
>>> cf = mb.obj['cash_flow_table']['testxml']['timber_prices']
>>> cf.classifiers # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
{'PRICE_REGION': <simo.builder.modelbase.table.TableClassifier...>,
 'assortment': <simo.builder.modelbase.table.TableClassifier...>}
>>> cf.classifier_order
['PRICE_REGION', 'assortment']
>>> cf.keycount
1
>>> cf.persist
True
>>> cf.db_cfiers
['assortment']
```

### 10.1.1 def \_\_init\_\_(self, ns, elem, validator, dirs):

Initialize cash flow table

### 10.1.2 def \_\_parse\_base(self, ns, elem, validator):

Parse the classifiers used in the cash flow table as well as the table data:

### 10.1.3 def \_\_get\_trend(self, date):

Set currently active trend, None if no active trends

### 10.1.4 def \_\_adjust\_table(self, trend, date):

Adjust cash flow table values with the current trend



### 10.1.5 def set\_active\_array(self, date):

Set active array and adjust table values with trend

### 10.1.6 def get\_values(self, classifiers, labels):

Get values from the active array

### 10.1.7 def set\_values(self, classifiers, labels, values):

Set values of active array

## 10.2 class Trend(object):

Class for cash flow table time trends

Attributes:

- start: date object
- end: date object
- classifiers: dictionary of classifier objects
- keycount: number of classifiers as integer
- classifier\_order: classifier order as list of integers
- factor\_array: trend factors in a numpy array
- factor\_dict: trend factors in a dictionary

### 10.2.1 def \_\_parse\_trend(self, ns, elem):

Parse trend element into Trend object:

```

>>> from lxml import etree
>>> from simo.builder.modelbase.cashflowtable import Trend
>>> trendxml = u"""
...     <trend>
...         <time_period>
...             <start>2000-01-01</start>
...             <end>2001-01-01</end>
...         </time_period>
...         <classifiers>
...             <classifier>
...                 <variable>SP</variable>
...                 <level>stratum</level>
...             </classifier>
...         </classifiers>
...         <cumulative_factors>
...             <factor>1 1.1</factor>
...             <factor>2 1.2</factor>
...             <factor>3 1.3</factor>
...         </cumulative_factors>
...     </trend>"""
>>> elem = etree.fromstring(trendxml)
>>> ns = ""
>>> tr = Trend(ns, elem, mb)

```

```
>>> tr.start
datetime.date(2000, 1, 1)
>>> tr.end
datetime.date(2001, 1, 1)
>>> tr.factor_array
array([[ 1. ,  1.1],
       [ 2. ,  1.2],
       [ 3. ,  1.3]])
>>> tr.factor_dict
{'1': 1.1000000000000001, '3': 1.3, '2': 1.2}
```

### 10.2.2 def adjust(self, date, values, factor):

Calculate interpolation multiplier based on given date:

```
>>> import datetime, numpy
>>> dt = datetime.date(2005,1,1)
>>> vals = numpy.ones(1, dtype=float)
>>> factor = 0.05
>>> tr.adjust(dt, vals, factor)
array([ 1.25])
```

## 10.3 class CashFlowTable(object):

Class for cash flow tables

Attributes:

- name: cash flow table name
- **classifiers: dictionary:**
  - keys: classifier indices
  - values: Classifier objects
- keycount: classifier key count
- arrays: list of Array objects
- Narrays: number of tables (arrays)
- active\_array: currently active table
- trends: list of Trend objects

### 10.3.1 def \_\_init\_\_(self, ns, elem, validator, dirs):

Parses the XML data into class attributes:

```
>>> mb.obj['cash_flow_table']['testxml']['timber_prices']
... # doctest: +ELLIPSIS
<simo.builder.modelbase.cashflowtable.CashFlowTable object at 0x...>
>>> ct = mb.obj['cash_flow_table']['testxml']['timber_prices']
>>> for i, j in ct.classifiers.iteritems(): print i, j
... # doctest: +ELLIPSIS
PRICE_REGION <simo.builder.modelbase.table.TableClassifier object at ...>
assortment <simo.builder.modelbase.table.TableClassifier object at ...>
>>> ct.arrays[0].array
array([[ 1.,  1.,  1., 160., 370., 55.]
```

```
[ 2., 1., 1., 160., 400., 56.]]
>>> ct.arrays[0].dict
{'1:1:1:160:370': 55.0, '2:1:1:160:400': 56.0}
```

### 10.3.2 def \_\_parse\_base(self, ns, elem):

Parses the classifiers used in the cash flow table as well as the table data

### 10.3.3 def \_\_get\_trend(self, date):

Set currently active trend, None if no active trends

### 10.3.4 def \_\_check\_classifiers(self, cfiers):

docstring for \_\_check\_classifiers

### 10.3.5 def \_\_adjust\_table(self, trend, date):

Adjust cash flow table values with the current trend

### 10.3.6 def set\_active\_array(self, date):

Set active array and adjust table values with trend:

```
>>> import datetime
>>> dt = datetime.date(2005,1,1)
>>> ct.set_active_array(dt)
```

### 10.3.7 def get\_values(self, classifiers, labels):

Get values from the active array:

```
>>> ct.get_values([1], [1])
array([ 55., 56.])
>>> ct.get_values([0], [1])
array([ 55.])
>>> ct.get_values([0], [2])
array([ 56.])
```

### 10.3.8 def set\_values(self, classifiers, labels, values):

Set values of active array:

```
>>> ct.set_values([1], [1], numpy.array([65, 66]))
>>> ct.active_array.array
array([[ 1., 1., 1., 160., 370., 65.],
       [ 2., 1., 1., 160., 400., 66.]])
```



# GEOTABLE.PY

```
>>> from simo.builder.modelbase.modelbase import ModelbaseDef
>>> tdf = open('../../simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../../simulator/xml/schemas/geo_table.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = u'''<geo_tables xmlns="http://www.simo-project.org/simo"
...     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...     <geo_table name="dem" desc="Digital elevation model">
...         <filename>test.latlonarray</filename>
...         <ulx>21.0</ulx>
...         <uly>61.0</uly>
...         <nrow>3</nrow>
...         <ncol>3</ncol>
...         <xdim>0.5</xdim>
...         <ydim>0.5</ydim>
...         <nvar>1</nvar>
...         <coordinate_variables>
...             <longitude>
...                 <variable>GEO_X</variable>
...                 <level>comp_unit</level>
...             </longitude>
...             <latitude>
...                 <variable>GEO_Y</variable>
...                 <level>comp_unit</level>
...             </latitude>
...         </coordinate_variables>
...         <targets>
...             <target>
...                 <variable>ALT</variable>
...                 <level>comp_unit</level>
...             </target>
...         </targets>
...     </geo_table>
... </geo_tables>
... '''

>>> class Validator(object):
...     def add_error(msg):
...         print msg

>>> class Lexicon(object):
...     def __init__(self):
...         self.models = {}
...     def get_variable_ind(self, level, variable, active=False):
...         if variable == 'GEO_X':
...             return (1,1)
```

```
...         elif variable == 'GEO_Y':
...             return (1,2)
...         elif variable == 'ALT':
...             return (1,3)
...         elif variable == 'TS':
...             return (1,4)
...     def get_level_ind(self, level):
...         return 1
...     def add_model(self, mtype, mname):
...         if mtype not in self.models:
...             self.models[mtype] = set()
...             self.models[mtype].add(mname)
>>> mb = ModelbaseDef(typedef)
>>> mb.schema = ('geo_table', schema)
>>> mb.xml = ('testxml', xml, Lexicon(),
...          ['./builder/modelbase/test/data'], 'geo_table')
>>> mb.xml['geo_table']['testxml'][:11]
u'<geo_tables'
```

## 11.1 class GeoTable(object):

Values tied to a regular grid over the earth. Coordinates in decimal degrees.

Attributes:

- name
- filename: where the data for the table is
- ulx: upper left x-coordinate (longitude)
- uly: upper left y-coordinate (latitude)
- nrow: number of data rows (along y-coord)
- ncol: number of data columns (along x-coord)
- xdim: the grid step in x-direction
- ydim: the grid step in y-direction
- nvar: number of variables tied to each node in the grid
- longitude\_var: data variable containing the longitude coordinate; dictionary with keys 'variable' and 'level' containing the variable and level indices for the data matrix
- latitude\_var: data variable containing the latitude coordinate; dictionary with keys 'variable' and 'level' containing the variable and level indices for the data matrix
- targets: result variable definitions: a list of (levelindex, variable index) tuples
- arr: the actual data in a multidimensional array
- sim: the simulator instance using the table

### 11.1.1 def \_\_init\_\_(self, ns, root, validator):

Construct geo table from the parsed geo table xml:

```
>>> gt = mb.obj['geo_table']['testxml']['dem']
>>> gt.load_function(Validator())
>>> gt.dirs
['./builder/modelbase/test/data']
>>> gt.filename
```

```

'test.latlonarray'
>>> gt.type
'geo_table'
>>> gt.ulx
21.0
>>> gt.longitude_var
(1, 1)
>>> gt.latitude_var
(1, 2)
>>> # the array is 3*3, values from 100 to 108
>>> gt.arr[0,2,0]
108.0
>>> gt.arr[2,0,0]
100.0
>>> gt.targets
[(1, 3)]

```

### 11.1.2 def \_\_get\_coord\_var(self, ns, celem):

Coordinate variable definition to data matrix indices

### 11.1.3 def \_\_read\_geo\_array(self):

Reads the geo array data from disk into a numpy array

## 11.2 class GeoTableParam(object):

Extracts geo table model parameters defined in the model chain

Attributes:

- `get_all`: boolean for returning all the variables found in the table
- `result_variables`: only a subset of table variables are returned: a list of:
  - dictionary having keys 'variable' and 'level' having variable index and level index in the data matrix as values

### 11.2.1 def \_\_init\_\_(self, ns, elem, variable\_ind, model):

Constructs the geo table parameter definition object based on the XML element. If result variables are listed, validates the variables using passed the lexicon method instance.

Invalid result variables defined in model chain parameters:

```

>>> execfile('builder/modelbase/test/mocktask.py')
>>> from lxml import etree
>>> from simo.builder.modelbase.geotable import GeoTableParam
>>> pelem = etree.XML('''<geo_table>
...                 <only_vars>
...                 <var>
...                 <name>TS</name>
...                 <level>comp_unit</level>
...                 </var>
...                 </only_vars>
...                 </geo_table>''')
>>> param = GeoTableParam('', pelem, task, gt)

```

```
>>> task.validator.errors # doctest: +NORMALIZE_WHITESPACE
set(["variable 'TS' at level 'comp_unit' is not a target variable for
    geo table 'dem'"])
```



# MANAGEMENTMODEL.PY

```
>>> from simo.builder.modelbase.modelbase import ModelbaseDef
>>> tdf = open('../../simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../../simulator/xml/schemas/management_modelbase.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = u'''<management_base
...     xmlns="http://www.simo-project.org/simo"
...     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...     <model>
...         <name>group_objects_by_variable</name>
...         <description>desc</description>
...         <implemented_at>simomanagement.py</implemented_at>
...         <parameters>
...             <parameter>
...                 <type>level</type>
...                 <description>target level</description>
...             </parameter>
...             <parameter>
...                 <type>level</type>
...                 <description>child level</description>
...             </parameter>
...             <parameter>
...                 <type>variable</type>
...                 <description>grouping variable</description>
...             </parameter>
...             <parameter>
...                 <type>value</type>
...                 <description>0 or 1</description>
...             </parameter>
...         </parameters>
...     </model>
... </management_base>'''
>>> class Lexicon(object):
...     def __init__(self):
...         self.models = {}
...     def get_variable_ind(self, level, variable, active=False):
...         if variable == 'GEO_X':
...             return (1,1)
...         elif variable == 'GEO_Y':
...             return (1,2)
...         elif variable == 'ALT':
...             return (1,3)
...         elif variable == 'TS':
...             return (1,4)
...     def get_level_ind(self, level):
...         return 1
```

```
...     def add_model(self, mtype, mname):
...         if mtype not in self.models:
...             self.models[mtype] = set()
...             self.models[mtype].add(mname)
>>> mb = ModelbaseDef(typedef)
>>> mb.schema = ('management', schema)
>>> mb.xml = ('testxml', xml, Lexicon(), '', 'management')
>>> mb.xml['management']['testxml'][:16]
u'<management_base'
```

## 12.1 class ManagementModel(object):

Management model

Attributes:

-name -at: the model library name in which the model implementation is found -parameters: a list of parameter definitions, list items are dictionaries with keys 'type' and 'desc'. Type is either level, variable or value.

### 12.1.1 def \_\_init\_\_(self, ns, elem):

Parses the XML data into model instance attributes:

```
>>> mm = mb.obj['management']['testxml']['group_objects_by_variable']
>>> mm.name
'group_objects_by_variable'
>>> mm.language
'python'
>>> mm.parameters
[{'type': 'level', 'desc': 'target level'}, {'type': 'level', 'desc': 'child level'}, {'type': 'v
```

## 12.2 class ModelParamManagementModel(object):

Parameter values for management model, defined in modelchain.

### 12.2.1 def \_\_init\_\_(self, ns, elem, lexicon, model):

Too few parameter values defined in model chain:

```
>>> execfile('builder/modelbase/test/mocktask.py')
>>> from simo.builder.modelbase.managementmodel import ManagementModelParam
>>> from lxml import etree
>>> pelem = etree.XML('''<management>
...     <parameters>
...         <parameter>
...             <level>stratum</level>
...         </parameter>
...     </parameters>
... </management>''')
>>> param = ManagementModelParam('', pelem, task, mm)
>>> task.validator.errors # doctest: +NORMALIZE_WHITESPACE
set(["wrong number of parameters defined for management model
    'group_objects_by_variable' in task 'test task'"])
```

Wrong type of parameters defined in model chain:

```

>>> pelem = etree.XML(''<management>
...                     <parameters>
...                     <parameter>
...                         <level>stratum</level>
...                     </parameter>
...                     <parameter>
...                         <variable>
...                             <name>SP</name>
...                             <level>stratum</level>
...                         </variable>
...                     </parameter>
...                     <parameter>
...                         <level>stratum</level>
...                     </parameter>
...                     <parameter>
...                         <value>1</value>
...                     </parameter>
...                     </parameters>
...                 </management>'')
>>> param = ManagementModelParam('', pelem, task, mm)
>>> task.validator.errors # doctest: +NORMALIZE_WHITESPACE
set(["wrong number of parameters defined for management model
    'group_objects_by_variable' in task 'test task'",
    "parameter at position 1 should be of type 'level' instead of
    'variable' for management model 'group_objects_by_variable'
    in task 'test task'",
    "parameter at position 2 should be of type 'variable' instead of
    'level' for management model 'group_objects_by_variable' in
    task 'test task'"])

```



# MODEL.PY

```
>>> from simo.builder.modelbase.model import Model
>>> from simo.builder.modelbase.model import PModel
>>> from lxml import etree
>>> xml = u'''<model>
...     <name>Productive_value_of_land_pine_Pukkala</name>
...     <implemented_at>StaticstandModels.dll</implemented_at>
...     <implemented_in>C</implemented_in>
...     <author>
...         <name>Timo Pukkala</name>
...     </author>
...     <description>    ...</description>
...     <published_in>    ...</published_in>
...     <species_list>
...         <species>1</species>
...     </species_list>
...     <geographical_coverage>    ...</geographical_coverage>
...     <applies_for>    ...</applies_for>
...     <research_material>    ...</research_material>
...     <variables>
...         <variable>
...             <name>TS</name>
...             <level>comp_unit</level>
...             <limits>
...                 <lower_limit>700</lower_limit>
...                 <upper_limit>1300</upper_limit>
...             </limits>
...         </variable>
...     </variables>
...     <parameters>
...         <parameter>
...             <name>IR</name>
...             <limits>
...                 <lower_limit>0</lower_limit>
...                 <upper_limit>20</upper_limit>
...             </limits>
...         </parameter>
...     </parameters>
...     <result>
...         <object>self</object>
...         <variables>
...             <variable>
...                 <name>PVland_Sp</name>
...             </variable>
...         </variables>
...     </result>
... </model>'''
>>> class Validator:
...     def elem_name(self, text):
```

```
...         return text
...     def variable_ind(self, level, variable, active=False):
...         return (1,1)
...     def add_error(self, errorstr):
...         pass
>>> elem = etree.fromstring(xml)
```

## 13.1 class Model(object):

Super class for all modelbase model classes.

Properties:

- type: get model type as string
- name: get model name as string
- language: get model implementation language as string
- dirs: get model library directories as a list of strings
- function: get model function object
- library: get model library object
- wrapper: get model library wrapper object

### 13.1.1 def \_\_init\_\_(self):

::

```
>>> m = Model()
```

### 13.1.2 def load\_function(self, validator):

Load model function object from model library:

```
>>> m._name = 'dummymodel'
>>> m._lang = 'python'
>>> m._lib = 'dummylib.py'
>>> m._dirs = ['builder/modelbase/test']
>>> m.load_function(Validator())
True
>>> m.function # doctest: +ELLIPSIS
<function dummymodel at ...>
```

### 13.1.3 def \_\_load\_library(self, validator):

Load the model library and wrapper module from the given path.

### 13.1.4 def \_\_load\_function\_object(self, validator):

Load model function object from model wrapper.

## 13.2 class POModel(Model):

Super class for prediction and operation models.

Properties:

- type: get model type as string
- name: get model name as string
- language: get model implementation language as string
- dirs: get model library directories as a list of strings
- function: get model function object
- library: get model library object
- wrapper: get model library wrapper object

Attributes:

- n\_vars: number of input variables
- vars: input variables in a dictionary: level as the key and value is a dictionary with the structure { 'index': <variable indices in a list>, 'order': <variable order in a list>, 'limits': <variable limits in a list> }
- n\_params: number of input parameters
- params: number of input parameters in a list where each item is a (<parameter name>, <parameter limits>) -tuple

### 13.2.1 def \_\_init\_\_(self, ns, elem, validator, dir):

```
>>> m = POModel('', elem, Validator(), 'dummydir')
>>> m.n_vars
1
>>> m.vars[1]['index']
array([1])
>>> m.vars[1]['order']
[0]
```

### 13.2.2 def \_\_parse\_parameters(self, ns, elem):

Parse model parameters and possible limits set for the parameter values.

### 13.2.3 def \_\_parse\_variables(self, ns, elem):

Parse model variable definitions (data level, variable name) and limits set for the variable values.

## 13.3 class Limit(object):

Class for handling model variable limits.

### 13.3.1 def \_\_init\_\_(self, ns, elem):

Construct variable/parameter limits from XML element:

```
>>> from lxml import etree
>>> from simo.builder.modelbase.model import Limit
>>> xml = u'''<limits>
...     <lower_limit>700</lower_limit>
...     <upper_limit>1300</upper_limit>
... </limits>'''
>>> ns = ''
>>> elem = etree.fromstring(xml)
>>> limits = Limit(ns, elem)
>>> limits.lower
700.0
>>> limits.upper
1300.0
```

### 13.3.2 def check(self, values):

Check if a value (or an array of values) is inside limits:

```
>>> import numpy
>>> values = numpy.array([600, 700, 1000, 1300, 1400], dtype=float)
>>> limits.check(values)
array([False,  True,  True,  True, False], dtype=bool)
```

### 13.3.3 def limitstr(self, variable):

Return a string representation of the limits:

```
>>> limits.limitstr('VARNAME')
"700.0 <= 'VARNAME' <= 1300.0"
```



# MODELBASE.PY

## 14.1 class ModelbaseDef(XmlObject):

Central repository for model definitions and implementations.

Four central properties:

- `typedef_schema`: the basic XML type definitions common for all SIMO XML documents
- `schema`: a dictionary of text representations of schema documents. Keyed by the model type: 'aggregation', 'cash\_flow', 'geo\_table', 'management', 'operation', 'parameter\_table', 'prediction'
- `xml`: a dictionary of dictionaries; first one keyed by the model type, second one by the XML document name and containing the text representations of the XML documents.
- `ok`: a dictionary of booleans, keyed by the model type. Indicates whether both the schema and all the XML documents have been parsed successfully.

### 14.1.1 def \_\_init\_\_(self, typedef):

Initializes the modelbase by processing the type definition schema.

### 14.1.2 def xml\_to\_obj(self, root, lexicon):

Responsible for creating the model object instances after the schema property has been set. This is called when the `xml` property is and each model class provides its own implementation of this method.



# OPERATIONMODEL.PY

```
>>> from simo.builder.modelbase.operationmodel import OperationModel
>>> from minimock import Mock
>>> from lxml import etree
>>> xml = u'''<operation_group name='testgroup'>
...     <operation>
...     <name>thinning</name>
...     <type>
...         <model>
...             <implemented_at>harvest.py</implemented_at>
...             <implemented_in>python</implemented_in>
...             <cash_flow_handler>model</cash_flow_handler>
...             <interaction>on</interaction>
...         </model>
...     </type>
...     <description>...</description>
...     <data>
...         <group>
...             <level>comp_unit</level>
...             <variables>
...                 <variable>
...                     <name>D_gM</name>
...                     <limits>
...                         <lower_limit>0</lower_limit>
...                         <upper_limit>40</upper_limit>
...                     </limits>
...                 </variable>
...             </variables>
...         </group>
...     </data>
...     <variables>
...         <variable>
...             <name>STEM_DIVISOR</name>
...             <level>simulation</level>
...         </variable>
...     </variables>
...     <parameters>
...         <parameter>
...             <name>target_density</name>
...             <optional>false</optional>
...             <limits>
...                 <enum>10 20 30</enum>
...             </limits>
...         </parameter>
...         <parameter>
...             <name>required_param</name>
...             <optional>false</optional>
...         </parameter>
...     </parameters>
```

```
...         <name>hupi_param</name>
...         <optional>True</optional>
...     </parameter>
... </parameters>
... <parameter_tables>
...     <parameter_table>thinning_preference_order</parameter_table>
... </parameter_tables>
... <operation_result_weight>
...     <name>AREA</name>
...     <level>comp_unit</level>
... </operation_result_weight>
... <results>
...     <type>
...         <operation_result/>
...     </type>
...     <variables>
...         <variable>
...             <name>Volume</name>
...             <classifiers>
...                 <classifier>
...                     <variable>SP</variable>
...                     <level>stratum</level>
...                 </classifier>
...                 <classifier>
...                     <cashflow_classifier>
...                         <variable>assortment</variable>
...                     </cashflow_classifier>
...                 </classifier>
...             </classifiers>
...         </variable>
...     </variables>
... </results>
... </operation>
... </operation_group>'''
>>> class Validator:
...     def elem_name(self, text):
...         return text
...     def variable_ind(self, level, variable, active=False):
...         return (1,1)
...     def level_ind(self, level):
...         return 1
...     def add_model(self, mtype, mname):
...         return 15
>>> elem = etree.fromstring(xml)
```

## 15.1 class OperationModel(Model):

Class for operation model definitions

Properties:

- type: get model type as string
- name: get model name as string
- language: get model implementation language as string
- dirs: get model library directories as a list of strings
- function: get model function object
- library: get model library object

- wrapper: get model library wrapper object

Attributes:

- n\_vars: number of input variables
- vars: input variables in a dictionary: level as the key and value is a dictionary with the structure {'index': <variable indices in a list>, 'order': <variable order in a list>, 'limits': <variable limits in a list>}
- n\_params: number of input parameters
- params: dictionary of parameters keyed by parameter name, valued by tuples of (optional, parameter limits)
- optype: operation type as string
- operation\_result\_weight: operation result variable indice: (level, variable)
- data\_vars: input data as dictionary, levels as keys, lists of variable indices as values
- data\_limits: input data limits
- cash\_flow\_handler: string, either 'model' or 'simulator'
- interactive: True or False, if interactive, model can affect data values, non-interactive models can only produce operation results.
- parameter\_tables: list of parameter table names
- result\_type: 'data' or 'operation\_result'
- result\_level: operation result target level
- result\_target: 'existing' or 'new'
- result\_variables: list of result variable tuples: (variable, classifier key)
- result\_classifiers: dictionary, where key: classifier key, value: list of classifier definitions

### 15.1.1 def \_\_init\_\_(self, ns, elem, validator, dirs, group):

Construct operation model object from XML element:

```
>>> op = OperationModel('', elem[0], Validator(), ['dummydir'], \
...                      'testgroup')
>>> op.name
'thinning'
>>> op.group
'testgroup'
>>> op.language
'python'
>>> op._lib
'harvest.py'
>>> op.vars
{1: {'index': array([1]), 'order': [0], 'limits': [None]}}
>>> ind = op.param_names.index('target_density')
>>> op.param_defs[ind] # doctest: +ELLIPSIS
(False, <...Limit object...>)
>>> ind = op.param_names.index('required_param')
>>> op.param_defs[ind]
(False, None)
>>> ind = op.param_names.index('hupi_param')
>>> op.param_defs[ind]
(True, None)
>>> op.n_params
3
>>> ind = op.param_names.index('target_density')
>>> l = op.param_defs[ind][1]
```

```
>>> l.enum
[10.0, 20.0, 30.0]
>>> l.lower
>>> op.operation_result_weight
(1, 1)
>>> op.data_vars
{1: [1]}
>>> op.data_limits # doctest: +ELLIPSIS
{1: [<...Limit object...>]}
>>> l = op.data_limits[1][0]
>>> l.lower
0.0
>>> l.upper
40.0
>>> op.cash_flow_handler
'model'
>>> op.interactive
True
>>> op.parameter_tables
['thinning_preference_order']
>>> op.result_type
'operation_result'
>>> op.result_level
>>> op.result_target
>>> op.result_variables
[('Volume', ('SP', 'assortment'))]
>>> op.result_classifiers.keys()
[('SP', 'assortment')]
>>> op.index
15
```

### 15.1.2 def \_parse\_operation(self, ns, elem):

Parse operation model structures

### 15.1.3 def \_parse\_data(self, ns, elem):

Parse operation model input data structures

### 15.1.4 def \_parse\_param\_tables(self, ns, elem):

Parse operation parameter table names

### 15.1.5 def \_parse\_weights(self, ns, elem):

Parse operation model result weight variable

### 15.1.6 def \_parse\_results(self, ns, elem):

Parse operation model result structures

### 15.1.7 def \_parse\_classifiers(self, ns, elem):

Parse operation model result classifiers

## 15.2 class Classifier(object):

Class for operation model result classifiers

Attributes:

- type
- variable
- level

### 15.2.1 def \_\_init\_\_(self, type, variable, level):

## 15.3 class OperationModelParam(object):

Operation model parameters

Attributes:

- parameters: operation model parameters in a dictionary, where parameter name is key and parameter value(s) is value
- long\_term: list of effects, effect: {'variable': ... , 'level':, ... , 'effect: ...}
- cash\_flow\_table: cash flow table implementation
- cash\_flow\_model: cash flow model implementation
- parameter\_tables: parameter tables as a list implementations
- simulation\_effects: simulation effects in a dictionary
- memory\_level: memory level indice

### 15.3.1 def \_\_init\_\_(self, ns, elem, validator, model, cftables, cfmodels, ptables):

Construct operation model parameters:

```
>>> execfile('builder/modelbase/test/mocktask.py')
>>> xml = u'''<operation>
...   <simulation_effects>
...     <erase_memory>false</erase_memory>
...   </simulation_effects>
...   <long_term>
...     <variable_effect>
...       <level>stratum</level>
...       <variable>iBA</variable>
...       <effects>
...         <effect>
...           <time_period>
...             <start>0</start>
...             <end>5</end>
...           </time_period>
...           <factor>1.25</factor>
...         </effect>
...       </effects>
...     </variable_effect>
...   </long_term>
...   <memory_level>stratum</memory_level>
...   <parameters>
...     <parameter>
```

```
...     <name>target_density</name>
...     <value>0.5</value>
...   </parameter>
...   <parameter>
...     <name>track_width</name>
...     <value>4</value>
...   </parameter>
... </parameters>
... <parameter_tables>
...   <parameter_table>
...     <table_definition_name>thinning_preference_order</table_definition_name>
...     <value_table_name>thinning_preference_order</value_table_name>
...   </parameter_table>
...   <parameter_table>
...     <table_definition_name>dummy</table_definition_name>
...     <value_table_name>dummy_table</value_table_name>
...   </parameter_table>
... </parameter_tables>
... <cash_flow_table>timber_prices</cash_flow_table>
... </operation>'''
>>> elem = etree.fromstring(xml)
>>> from simo.builder.modelbase.operationmodel import OperationModelParam
>>> cftables = {'timber_prices': Mock('CashFlowTable')}
>>> cftables['timber_prices'].classifier_order = [1]
>>> cftables['timber_prices'].keycount=1
>>> clf = Mock('Classifier')
>>> clf.level = 1
>>> clf.ind = 2
>>> cftables['timber_prices'].classifiers = {1:clf}
>>> cfmodels = {'cf_model': Mock('CashFlowModel')}
>>> ptables = {'thinning_preference_order': Mock('ParameterTable')}
>>> omp = OperationModelParam('', elem, task, op, cftables, cfmodels,
...                             ptables)
>>> omp.simulation_effects
{'force_step': None, 'erase_memory': False}
>>> omp.parameters
[0.5, None, None]
>>> omp.long_term
[{'variable': 1, 'effects': [[0, 5], 1.25]], 'level': 2}]
>>> omp.cash_flow_table # doctest: +ELLIPSIS
<Mock ... CashFlowTable>
>>> omp.cash_flow_model
>>> omp.parameter_tables # doctest: +ELLIPSIS
[<Mock ... ParameterTable>]
>>> omp.memory_level
2
>>> errs = list(task.validator.errors)
>>> errs.sort()
>>> errs # doctest: +NORMALIZE_WHITESPACE
["Cash flow table key classifier '1' not found on level 'comp_unit'
  for operation model 'thinning', task 'test task'",
 "Non optional parameter 'required_param' is not defined for operation
  model 'thinning', task 'test task'",
 "Parameter 'track_width' is not a valid parameter for operation model
  'thinning', task 'test task'",
 "Parameter table 'dummy' is not valid for operation model 'thinning',
  task 'test task'",
 "Value 0.5 for parameter 'target_density' is not in the accepted value
  list [10.0, 20.0, 30.0] for operation model 'thinning',
  task 'test task'"]
```



### 15.3.2 def \_parse\_params(self, ns, elem):

Parse operation model parameters when there shouldn't be any

```

>>> ns = ''
>>> elem = Mock('Elem')
>>> elem.find.mock_returns = None
>>> omp.model = Mock('Model')
>>> omp.model.n_params = 0
>>> omp.model.name = 'test model'
>>> omp._validator = Mock('Validator')
>>> omp._parse_params(ns, elem)
Called Elem.find('parameters')

```

Parse parameters when there should be two, but there are none

```

>>> omp.model.n_params = 2
>>> omp._parse_params(ns, elem) # doctest: +NORMALIZE_WHITESPACE
Called Elem.find('parameters')
Called Validator.add_error(
    "No parameters defined for model 'test model', 2 expected, task 'test
    task'")

```

### 15.3.3 def \_parse\_long\_term(self, ns, elem):

Parse operation model long time effects

### 15.3.4 def \_get\_cash\_flow\_table(self, ns, elem):

Get operation model cash flow table

### 15.3.5 def \_get\_cash\_flow\_model(self, ns, elem):

Get operation model cash flow model

### 15.3.6 def \_get\_param\_tables(self, ns, elem):

Get operation model parameter tables

### 15.3.7 def \_get\_sim\_effects(self, ns, elem):

Get operation model simulation effects

### 15.3.8 def \_get\_mem\_level(self, ns, elem):

Get operation model memory level



# PARAMETERTABLE.PY

```
>>> from simo.builder.modelbase.modelbase import ModelbaseDef
>>> tdf = open('../../simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../../simulator/xml/schemas/parameter_table.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = u'''<parameter_tables xmlns="http://www.simo-project.org/simo"
...     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
... <parameter_table name="regeneration_limits"
...     type="parameter_table" desc="...">
...     <classifiers>
...         <classifier>
...             <variable>MAIN_SP</variable>
...             <level>comp_unit</level>
...         </classifier>
...     </classifiers>
...     <targets>
...         <target>
...             <data>
...                 <variable>upper_dgm</variable>
...                 <level>comp_unit</level>
...             </data>
...         </target>
...         <target>
...             <data>
...                 <variable>upper_age</variable>
...                 <level>comp_unit</level>
...             </data>
...         </target>
...     </targets>
...     <table>
...         <value>1 32 70</value>
...         <value>9 21 50</value>
...     </table>
... </parameter_table>
... </parameter_tables>'''
>>> mb = ModelbaseDef(typedef)
>>> mb.schema = ('parameter_table', schema)
>>> try:
...     mb.xml = ('testxml', xml, None, '', 'parameter_table')
... except ValueError, e:
...     e.message
'errors in xml to object conversion'
>>> mb.errors # doctest: +NORMALIZE_WHITESPACE
set(["No lexicon set when adding model for parameter table
    collection 'testxml'",
    "No lexicon set when validating variable for parameter table
```

```
collection 'testxml'"]])
>>> class Lexicon(object):
...     def __init__(self):
...         self.models = {}
...     def get_variable_ind(self, level, var, active=False):
...         return (1, 1)
...     def get_level_ind(self, level):
...         return 1
...     def add_model(self, mtype, mname):
...         if mtype not in self.models:
...             self.models[mtype] = set()
...             self.models[mtype].add(mname)
>>> mb.xml = ('testxml', xml, Lexicon(), '', 'parameter_table')
>>> mb.errors
set([])
>>> mb.ok
{'testxml': True}
>>> mb.xml['parameter_table']['testxml'][:17]
u'<parameter_tables'
```

## 16.1 class ParameterTable(object):

Parameter table definition

Attributes:

- name: parameter table name as string
- labels: parameter table labels
- classifiers: parameter table classifiers as a dictionary with TableClassifier objects as values
- Nclassifiers: number of classifiers as integer
- targets: parameter table targets as a list of ParamTableTarget objects
- Ntargets: number of targets as integer
- arrays: list of Array objects
- Narrays: number of arrays as integer
- active\_array: currently active array

### 16.1.1 def \_\_init\_\_(self, ns, elem, validator, dirs):

## 16.2 class ParamTableTarget(object):

Class for parameter table targets

Attributes:

- type: target type
- name: target variable name
- level: target variable level
- length: target length

### 16.2.1 def \_\_init\_\_(self, ns, elem, validator):

# PREDICTIONMODEL.PY

```
>>> from simo.builder.modelbase.predictionmodel import PredictionModel
>>> from lxml import etree
>>> xml = u'''<model_group name="Distribution models">
...     <model>
...         <name>Productive_value_of_land_pine_Pukkala</name>
...         <implemented_at>StaticstandModels.dll</implemented_at>
...         <implemented_in>C</implemented_in>
...         <author>
...             <name>Timo Pukkala</name>
...         </author>
...         <description>...</description>
...         <published_in>...</published_in>
...         <species_list>
...             <species>1</species>
...         </species_list>
...         <geographical_coverage>...</geographical_coverage>
...         <applies_for>...</applies_for>
...         <research_material>...</research_material>
...         <variables>
...             <variable>
...                 <name>TS</name>
...                 <level>comp_unit</level>
...             </variable>
...         </variables>
...         <parameters>
...             <parameter>
...                 <name>IR</name>
...             </parameter>
...         </parameters>
...         <result>
...             <object>self</object>
...             <variables>
...                 <variable>
...                     <name>PVland_Sp</name>
...                 </variable>
...             </variables>
...         </result>
...     </model>
... </model_group>'''
>>> class Validator:
...     def elem_name(self, text):
...         return text
...     def variable_ind(self, level, variable, active=False):
...         return (1,1)
...     def add_model(self, mname, mtype):
...         pass
>>> elem = etree.fromstring(xml)
```

## 17.1 class PredictionModel(POModel):

Class for prediction model definitions

Properties:

- type: get model type as string
- name: get model name as string
- language: get model implementation language as string
- dirs: get model library directories as a list of strings
- function: get model function object
- library: get model library object
- wrapper: get model library wrapper object

Attributes:

- n\_vars: number of input variables
- vars: input variables in a dictionary: level as the key and value is a dictionary with the structure {'index': <variable indices in a list>, 'order': <variable order in a list>, 'limits': <variable limits in a list>}
- n\_params: number of input parameters
- params: number of input parameters in a list where each item is a (<parameter name>, <parameter limits>)-tuple
- result\_level: result level indice as int
- result\_vars: result variables in a list of ResultVariable instances

### 17.1.1 def \_\_init\_\_(self, ns, elem, validator, dirs):

Construct prediction model object from XML element:

```
>>> pr = PredictionModel('', elem[0], Validator(), 'dummydir',
...                       elem.attrib['name'])
>>> pr.name
'Productive_value_of_land_pine_Pukkala'
>>> pr.group
'Distribution models'
>>> pr.language
'c'
>>> pr.n_vars
1
>>> pr.vars
{1: {'index': array([1]), 'order': [0], 'limits': [None]}}
>>> pr.n_params
1
>>> pr.param_names
['IR']
>>> pr.param_limits
[None]
>>> pr.result_level
1
```

## 17.2 class ResultVariable(object):

Class for prediction model result variables

Attributes:

- variable
- time\_span
- time\_unit
- cumulation

### 17.2.1 def \_\_init\_\_(self, variable, timespan=None, unit=None, cumul=None):

## 17.3 class PredictionModelParam(Persistent):

Class for prediction model parameters

### 17.3.1 def \_\_init\_\_(self, ns, elem, task, model):

Initialize prediction model parameter object:

```
>>> execfile('builder/modelbase/test/mocktask.py')
>>> xml_no_param = u'''<prediction>
...     <rect_factor>1.15</rect_factor>
...     <risk_level>2</risk_level>
... </prediction>'''
>>> xml = u'''<prediction>
...     <parameters>
...         <parameter>
...             <name>IR</name>
...             <value>10.5</value>
...         </parameter>
...         <parameter>
...             <name>PARAMETER2</name>
...             <value>VALUE</value>
...         </parameter>
...     </parameters>
...     <rect_factor>1.15</rect_factor>
...     <risk_level>2</risk_level>
... </prediction>'''
>>> from simo.builder.modelbase.predictionmodel import PredictionModelParam
>>> elem = etree.fromstring(xml_no_param)
>>> pmp = PredictionModelParam('', elem, task, pr)
>>> abs(pmp.rect_factor - 1.15) < 0.0001
True
>>> abs(pmp.risk_level - 2.0) < 0.0001
True
>>> task.validator.errors # doctest: +NORMALIZE_WHITESPACE
set(["No parameters defined in model chain for prediction model
'Productive_value_of_land_pine_Pukkala', 1 parameters expected"])
>>> task.validator.errors = set([])
>>> elem = etree.fromstring(xml)
>>> pmp = PredictionModelParam('', elem, task, pr)
>>> pmp.parameters
[10.5]
>>> task.validator.errors # doctest: +NORMALIZE_WHITESPACE
```

```
set(["Parameter 'PARAMETER2' is not a valid parameter for prediction  
    model 'Productive_value_of_land_pine_Pukkala'"])
```



# TABLE.PY

## 18.1 def table2array(table):

Create a single Numpy array from XML cash\_flow table:

```
>>> from lxml import etree
>>> from simo.builder.modelbase.table import table2array
>>> tablexml = u"""<table>
...     <cash_flow>1 1 1 160 370 55</cash_flow>
...     <cash_flow>2 1 1 160 400 56</cash_flow>
... </table>"""
>>> elem = etree.fromstring(tablexml)
>>> arr = table2array(elem)
>>> arr
array([[ 1.,    1.,    1., 160., 370., 55.],
       [ 2.,    1.,    1., 160., 400., 56.]])
```

## 18.2 def table2dict(table):

Create a dictionary with the classifying values joined together as the dictionary key and the cash flow as the value; i.e., a row '1 2 3 4 5 6 7 100' translates to `cdict['1:2:3:4:5:6:7']=100`:

```
>>> from lxml import etree
>>> from simo.builder.modelbase.table import table2dict
>>> tablexml = u"""<table>
...     <cash_flow>1 1 1 160 370 55</cash_flow>
...     <cash_flow>2 1 1 160 400 56</cash_flow>
... </table>"""
>>> elem = etree.fromstring(tablexml)
>>> dct = table2dict(elem)
>>> dct
{'1:1:1:160:370': 55.0, '2:1:1:160:400': 56.0}
```

## 18.3 def parse\_classifiers(ns, elem):

Parse cash flow table and/or trend classifiers:

```
>>> class Lexicon(object):
...     def variable_ind(self, level, var, active=False):
...         return (1, 1)
...     def level_ind(self, level):
...         return 1
```

```
>>> class Validator(object):
...     def __init__(self, lexicon):
...         self.errors = set([])
...         self.warnings = []
...         self.lexicon = lexicon
...     def variable_ind(self, level, variable):
...         return self.lexicon.variable_ind(level, variable)
...     def level_ind(self, level):
...         return self.lexicon.level_ind(level)
...     def add_error(self, msg):
...         self.errors.add(msg)
...     def add_warnings(self, msg):
...         self.warnings.append(msg)
...     model_type = 'cash_flow_table'
>>> validator = Validator(Lexicon())
>>> from lxml import etree
>>> from simo.builder.modelbase.table import parse_classifiers
>>> cfierxml = u"""<classifiers>
...     <classifier>
...         <variable>PRICE_REGION</variable>
...         <level>comp_unit</level>
...     </classifier>
...     <classifier to_db="true">
...         <variable>SP</variable>
...         <level>comp_unit</level>
...     </classifier>
...     <classifier>
...         <variable>assortment</variable>
...         <level>within-operation</level>
...         <labels>
...             <label>
...                 <value>1</value>
...                 <name>log</name>
...             </label>
...             <label>
...                 <value>2</value>
...                 <name>pulp</name>
...             </label>
...         </labels>
...     </classifier>
... </classifiers>"""
>>> elem = etree.fromstring(cfierxml)
>>> ns = ""
>>> cfiers, db_cfiers, db_cfiers_ind, corder, keycount = \
...     parse_classifiers(ns, elem, validator)
>>> 'PRICE_REGION' in cfiers
True
>>> c = cfiers['assortment']
>>> c.name
'assortment'
>>> c.level
'within-operation'
>>> c.ind
>>> c.order
2
>>> c.variable_type
>>> len(c.labels)
2
>>> l = c.labels[1]
>>> l.label
'pulp'
>>> l.value
```

```

2
>>> db_cfiers
['SP']
>>> keycount
2

```

## 18.4 def parse\_tables(ns, elem):

Parse <tables> element contents into ctypes array, value dictionary and activation time objects for each table:

```

>>> from lxml import etree
>>> from simo.builder.modelbase.table import parse_tables
>>> tablesxml = u"""<tables>
...     <table>
...         <cash_flow>1 1 1 160 370 55</cash_flow>
...         <cash_flow>2 1 1 160 400 56</cash_flow>
...     </table>
... </tables>"""
>>> elem = etree.fromstring(tablesxml)
>>> tables, n = parse_tables(elem)
>>> tables[0].array
array([[ 1.,    1.,    1., 160., 370., 55.],
       [ 2.,    1.,    1., 160., 400., 56.]])
>>> tables[0].dict
{'1:1:1:160:370': 55.0, '2:1:1:160:400': 56.0}
>>> n
1

```

### 18.4.1 class Label(object):

Class for cash flow table classifier labels

Attributes:

- label: label text
- value: label value

### 18.4.2 class Classifier(object):

Class for cash flow table classifiers

Attributes:

- level: classifier level as string
- name: classifier name as string
- ind: classifier level-variable indices a tuple: (level, variable,)
- labels: list of Label objects
- order: classifier order as list of integers

### 18.4.3 class Array(object):

Class for cash flow table arrays

Attributes:

- array: table values in a numpy array
- dict: table values in a dictionary
- start\_date: date object
- end\_date: date object

# CONDITIONPARSER.PY

## 19.1 class ConditionParser(object):

Class for parsing conditions in model chain into Reverse Polish Notation form.

### 19.1.1 def \_\_init\_\_(self, expr, validator, name=None):

Validator is responsible for providing the matrix indices for all the variable, level and operation definitions used in the condition expression:

```
>>> class Lexicon(object):
...     def __init__(self):
...         self.errors = []
...     def variable_ind(self, level, variable):
...         if variable == 'AGE':
...             return (1, 'AGE')
...         elif variable == 'SC':
...             return (1, 'SC')
...         elif variable == 'SP':
...             return (2, 'SP')
...         elif variable == 'd':
...             return (3, 'd')
...         else:
...             return (1, 'past_thinning')
...     def level_ind(self, level):
...         return 2
...     def operation_ind(self, operation):
...         if operation == 'thinning':
...             return 1
...         else:
...             return 2
...     def add_error(self, msg):
...         self.errors.append(msg)
>>> from simo.builder.modelchain.conditionparser import ConditionParser
>>> #from conditionparser import ConditionParser
>>> cp = ConditionParser(Lexicon(), 'comp_unit')
```

Note that normally the Lexicon methods return only indices, but here text is used to illustrate the structure of the parsed output.

### 19.1.2 def parse(self, expr):

Parses the SIMO condition expression into a nested list and further into a Reverse Polish notation / Postfix stack:

```
>>> s0 = 'comp_unit:AGE gt 1.0'
>>> c = cp.parse(s0)
>>> for i in c:
...     print i # doctest: +ELLIPSIS
('variable', (1, 'AGE', True))
('value', 1.0)
('eq', <function gte at ...>)

>>> s1 = 'comp_unit:AGE gt 1.0 and comp_unit:SC le 4.0'
>>> c = cp.parse(s1)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (1, 'AGE', True))
('value', 1.0)
('eq', <function gte at ...>)
('variable', (1, 'SC', True))
('value', 4.0)
('eq', <function lee at ...>)
('group', <function and_ at ...>)

>>> s2 = '''(comp_unit:AGE gt 1.0 and comp_unit:SC le 4.0)
...         or
...         stratum:SP in [1.0, 2.0, 3.0]'''
>>> c = cp.parse(s2)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (1, 'AGE', True))
('value', 1.0)
('eq', <function gte at ...>)
('variable', (1, 'SC', True))
('value', 4.0)
('eq', <function lee at ...>)
('group', <function and_ at ...>)
('variable', (2, 'SP', True))
('value', [1.0, 2.0, 3.0])
('eq', <function in_ at ...>)
('group', <function or_ at ...>)

>>> s3 = '''((comp_unit:AGE gt 1.0 and comp_unit:SC le 4.0)
...         or comp_unit:thinning since_gt 10)'''
>>> c = cp.parse(s3)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (1, 'AGE', True))
('value', 1.0)
('eq', <function gte at ...>)
('variable', (1, 'SC', True))
('value', 4.0)
('eq', <function lee at ...>)
('group', <function and_ at ...>)
('operation', 1)
('value', 10.0)
('since', <function sgt at ...>)
('group', <function or_ at ...>)

>>> s4 = '''((comp_unit:AGE gt 1.0 and comp_unit:SC le 4.0)
...         or stratum:SP in [1.0, 2.0, 3.0])
...         and
...         ((comp_unit:thinning since_gt 10
...         or comp_unit:thinning since_gt comp_unit:past_thinning)
...         or stratum not exists)'''
>>> c = cp.parse(s4)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (1, 'AGE', True))
('value', 1.0)
```

```

('eq', <function gte at ...>)
('variable', (1, 'SC', True))
('value', 4.0)
('eq', <function lee at ...>)
('group', <function and_ at ...>)
('variable', (2, 'SP', True))
('value', [1.0, 2.0, 3.0])
('eq', <function in_ at ...>)
('group', <function or_ at ...>)
('operation', 1)
('value', 10.0)
('since', <function sgt at ...>)
('operation', 1)
('variable', (1, 'past_thinning', True))
('since', <function sgt at ...>)
('group', <function or_ at ...>)
('level', 2)
None
('ex', <function n_ext at ...>)
('group', <function or_ at ...>)
('group', <function and_ at ...>)

>>> s5 = '''(comp_unit:AGE gt 1.0 and comp_unit:SC le 4.0
...         or stratum:SP in [1.0, 2.0, 3.0])
...         and
...         (comp_unit:thinning since_gt 10
...         or comp_unit:thinning since_gt comp_unit:past_thinning
...         or stratum not exists)'''
>>> c = cp.parse(s5)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (1, 'AGE', True))
('value', 1.0)
('eq', <function gte at ...>)
('variable', (1, 'SC', True))
('value', 4.0)
('eq', <function lee at ...>)
('group', <function and_ at ...>)
('variable', (2, 'SP', True))
('value', [1.0, 2.0, 3.0])
('eq', <function in_ at ...>)
('group', <function or_ at ...>)
('operation', 1)
('value', 10.0)
('since', <function sgt at ...>)
('operation', 1)
('variable', (1, 'past_thinning', True))
('since', <function sgt at ...>)
('group', <function or_ at ...>)
('level', 2)
None
('ex', <function n_ext at ...>)
('group', <function or_ at ...>)
('group', <function and_ at ...>)

>>> s = '''comp_unit:AGE gt 1.0 or comp_unit:SC le 4.0
...         or stratum:SP in [1.0, 2.0, 3.0]'''
>>> c = cp.parse(s)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (1, 'AGE', True))
('value', 1.0)
('eq', <function gte at ...>)
('variable', (1, 'SC', True))

```

```
('value', 4.0)
('eq', <function lee at ...>)
('group', <function or_ at ...>)
('variable', (2, 'SP', True))
('value', [1.0, 2.0, 3.0])
('eq', <function in_ at ...>)
('group', <function or_ at ...>)
>>> s = 'comp_unit:AGE exists'
>>> c = cp.parse(s)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (1, 'AGE', True))
None
('ex', <function ext at ...>)

>>> s = 'tree:d gt self:d'
>>> c = cp.parse(s)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (3, 'd', True))
('variable', (3, 'd', False))
('eq', <function gte at ...>)
>>> cp.validator.errors
[]
>>> cp.validator.errors = []

>>> s = 'tree:d exists'
>>> c = cp.parse(s)
>>> for i in c:
...     print i # doctest: +ELLIPSIS
('variable', (3, 'd', True))
None
('ex', <function ext at ...>)

>>> s = 'function:random_number_0_1 lt 2.0'
>>> c = cp.parse(s)
>>> for i in c:
...     print i # doctest: +ELLIPSIS
('function', <function random_number_0_1 at ...>)
('value', 2.0)
('eq', <function lte at ...>)
```

#### Failing conditions:

```
>>> f = 'comp_unit;AGE gt 1.0'
>>> c = cp.parse(f)
>>> cp.validator.errors
['Error parsing expression "comp_unit;AGE gt 1.0", Expected ":" at char 9, got ";AGE"']
>>> cp.validator.errors = []
>>> f = 'comp_unit:AGE gt 1,0'
>>> c = cp.parse(f)
>>> cp.validator.errors
['Error parsing expression "comp_unit:AGE gt 1,0", Expected end of text at char 18, got ",0"']
>>> cp.validator.errors = []
>>> f = 'comp_unit:AGE > 1.0'
>>> c = cp.parse(f)
>>> cp.validator.errors
['Error parsing expression "comp_unit:AGE > 1.0", Expected Re:(\'gt|ge|eq|ue|le|lt\') at char 14,']
>>> cp.validator.errors = []
>>> f = 'comp_unit:AGE gt 1.0 and comp_unit:SC <= 4.0'
>>> c = cp.parse(f)
>>> cp.validator.errors
['Error parsing expression "comp_unit:AGE gt 1.0 and comp_unit:SC <= 4.0", Expected Re:(\'gt|ge|e']
>>> cp.validator.errors = []
```



```
>>> f = '''((comp_unit:AGE gt 1.0 and comp_unit:SC le 4.0)
...         or stratum:SP in [1.0, 2.0, 3.0])
...         and
...         ((comp_unit:thinning since_gt 10
...           or comp_unit:thinning since_gt comp_unit:past_thinning)
...           or stratum not exist)'''
>>> c = cp.parse(f)
>>> cp.validator.errors
['Error parsing expression "((comp_unit:AGE gt 1.0 and comp_unit:SC le 4.0) or stratum:SP in [1.0, 2.0, 3.0]) and ((comp_unit:thinning since_gt 10 or comp_unit:thinning since_gt comp_unit:past_thinning) or stratum not exist)'"']
>>> cp.validator.errors = []
>>> f = 'comp_unit:AGE gt 1.0 and SC lt 4.0'
>>> c = cp.parse(f)
>>> cp.validator.errors
['Error parsing expression "comp_unit:AGE gt 1.0 and SC lt 4.0", Expected ":" at char 28, got "lt"']
```

### 19.1.3 def \_\_construct\_condition(self, cond):

Responsible for the actual processing of the parsed nested list into a RPN stack.

### 19.1.4 def \_\_process\_triplet(self, cond):

Processess the atomic condition of [variable, operator, value] or [variable, operator, variable] or [level, exists operator] or [operation, operator, value] or [operation, operator, variable].

Also recursively calls the `_construct_condition` to divide the [condition, boolean operator, condition] cases.

### 19.1.5 def \_\_set\_variable(self, dtype, vardef):

Retrieves the indices needed for 'variable', 'operation' and 'level' type operands.



# MODELCHAIN.PY

## 20.1 class ModelChainDef(XmlObject):

Model chain definitions.

### 20.1.1 def \_\_init\_\_(self, typedef):

Initializes the model chain collection by processing the type definition schema:

```
>>> from simo.builder.modelchain.modelchain import ModelChainDef
>>> from simo.builder import names
>>> from minimock import Mock
>>> tdf = open('../../simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../../simulator/xml/schemas/model_chain.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = u'''<model_chains
...   xsi:schemaLocation="http://www.simo-project.org/simo
...   ../../schemas/model_chain.xsd"
...   xmlns="http://www.simo-project.org/simo"
...   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...   <chain_group name="Forced thinnings">
...     <model_chain evaluate_at="comp_unit"
...       name="Calculate thinning limits">
...       <task name="Calculate thinning limits">
...         <model>
...           <name>Thinning_limits_JKK</name>
...           <prediction>
...             <parameters>
...               <parameter>
...                 <name>THINNING_LEVEL</name>
...                 <value>0.5</value>
...               </parameter>
...             </parameters>
...           </prediction>
...         </model>
...       </task>
...     </model_chain>
...     <model_chain evaluate_at="comp_unit" name="Forced first thinning">
...       <task name="Basic thinning for pine in southern Finland">
...         <task name="Invalid model">
...           <model>
...             <name>invalid</name>
...             <aggregation>
```

```
...         <target>JUST_THINNED</target>
...         <operands>
...             <operand>
...                 <value>0</value>
...             </operand>
...         </operands>
...     </aggregation>
... </model>
... </task>
... <task name="Assign just thinned dummy to zero to avoid">
...     <model>
...         <name>assign_value</name>
...         <aggregation>
...             <target>JUST_THINNED</target>
...             <operands>
...                 <operand>
...                     <value>0</value>
...                 </operand>
...             </operands>
...         </aggregation>
...     </model>
... </task>
... </task>
... </model_chain>
... </chain_group>
... <chain_group name="Forced clearcut and strip cut">
...     <model_chain evaluate_at="comp_unit" name="Forced clearcut">
...         <condition>comp_unit:MAIN_SP in [1, 8] and
...             comp_unit:NF eq 0</condition>
...         <task name="just making the structure deeper">
...             <task name="clearcut">
...                 <condition>comp_unit:MAIN_SP in [1, 8] and
...                     comp_unit:SOME eq 0</condition>
...                 <model>
...                     <name>clearcut</name>
...                     <operation>
...                         <simulation_effects>
...                             <erase_memory>false</erase_memory>
...                         </simulation_effects>
...                         <cash_flow_table>timber_prices</cash_flow_table>
...                     </operation>
...                 </model>
...             </task>
...         </task>
...     </model_chain>
...     <branching_group name="Final_harvest">
...         <condition>comp_unit:open_area eq 0</condition>
...         <branch_task
...             branching_operations="seedtree_position;clearcut"
...             name="Clearcut or seedtree cut">
...             <branch_conditions>
...                 <condition name="Normal clearcut">
...                     comp_unit:passed_regen_limit gt 0
...                 </condition>
...                 <condition name="Delayed clearcut, 5 years">
...                     comp_unit:passed_regen_limit gt 5
...                 </condition>
...                 <condition name="Delayed clearcut, 10 years">
...                     comp_unit:passed_regen_limit gt 10
...                 </condition>
...             </branch_conditions>
...             <condition>comp_unit:MAIN_GROUP eq 1 and
...                 comp_unit:REGENERABLE eq 1</condition>
```

```

...         <task name="Clearcut">
...             <condition>comp_unit:PEAT eq 0</condition>
...             <task name="clearcut">
...                 <model>
...                     <name>clearcut2</name>
...                     <operation>
...                         <simulation_effects>
...                             <erase_memory>false</erase_memory>
...                         </simulation_effects>
...                         <cash_flow_table>timber_prices</cash_flow_table>
...                     </operation>
...                 </model>
...             </task>
...             <task name="zero REGENERABLE comp_unit">
...                 <model>
...                     <name>zero_value</name>
...                     <aggregation>
...                         <target>REGENERABLE</target>
...                         <operands>
...                             <operand>
...                                 <variable>REGENERABLE</variable>
...                                 <level>comp_unit</level>
...                             </operand>
...                         </operands>
...                     </aggregation>
...                 </model>
...             </task>
...         </branch_task>
...     </branching_group>
... </model_chain>
... </chain_group>
... </model_chains>'''
>>> class Lexicon(object):
...     def get_variable_ind(self, level, variable, active=False):
...         if variable == 'MAIN_SP':
...             return (1,1)
...         elif variable == 'NF':
...             return (1,2)
...         if variable == 'REGENERABLE':
...             return (1,3)
...         elif variable == 'MAIN_GROUP':
...             return (1,4)
...         elif variable == 'passed_regen_limit':
...             return (1,5)
...         elif variable == 'PEAT':
...             return (1,6)
...         else:
...             return (None, None)
...     def get_level_ind(self, level):
...         return 1
...     def get_level_name(self, level):
...         return 'mock level'
...     def get_operation_ind(self, operation):
...         return 3
...     models = {names.PREDICTION: ['Thinning_limits_JKK'],
...               names.OPERATION: ['clearcut', 'clearcut2'],
...               names.AGGREGATION: ['assign_value']}
>>> m = Mock('Model')
>>> pm = Mock('PredictionModel')
>>> pm.name = 'mock model'
>>> pm.param_names = ['test']

```

```
>>> pm.param_limits = [None]
>>> pm.n_params = 1
>>> cf = Mock('Classifier')
>>> om1 = Mock('OperationModel')
>>> om1.result_classifiers = {('SP', 'assortment'): [cf, cf]}
>>> om1.result_type = 'operation_result'
>>> om1.result_variables = [('Volume', None), ('Income', None)]
>>> om1.n_params = 0
>>> om1.parameter_tables = None
>>> om2 = Mock('OperationModel')
>>> om2.result_classifiers = {('SP', 'something else'): [cf, cf]}
>>> om2.result_type = 'operation_result'
>>> om2.result_variables = [('Volume', None), ('Income', None)]
>>> om2.n_params = 0
>>> om2.parameter_tables = None
>>> om3 = Mock('OperationModel')
>>> om3.result_classifiers = {}
>>> om3.result_type = 'data'
>>> om3.result_variables = [('dummy_1', 1), ('dummy_2', 2)]
>>> om3.n_params = 0
>>> om3.parameter_tables = None
>>> cft = Mock('CashFlowTable')
>>> cft.classifier_order = []
>>> cft.keycount=0
>>> clf = Mock('Classifier')
>>> clf.persist = True
>>> clf.level = 2
>>> clf.ind = 2
>>> cft.classifiers = {'SP':clf}
>>> cft.db_cfiers = set(['SP'])
>>> mi = {names.PREDICTION: {'Thinning_limits_JKK': pm},
...       names.OPERATION: {'clearcut': om1, 'clearcut2': om2,
...                           'planting': om3},
...       names.AGGREGATION: {'assign_value': m},
...       names.CASH_FLOW_TABLE: {'timber_prices': cft},
...       names.CASH_FLOW: {},
...       names.PARAMETER_TABLE: {}}
>>> mcd = ModelChainDef(typedef)
>>> mcd.schema = schema
```

### 20.1.2 def xml\_to\_obj(self, xmlname, root, lexicon, chaintype, models):

Converts the model chain xml definition into Python objects::

```
>>> try:
...     mcd.xml = ('testxml', xml, Lexicon(), 'simulation', mi)
... except ValueError, e:
...     print e.message
Called PredictionModel.vars.keys()
Called OperationModel.vars.keys()
Called OperationModel.vars.keys()
errors in xml to object conversion
>>> mcd.xml['simulation']['testxml'][:13]
u'<model_chains'
>>> mc = mcd.obj['simulation']['testxml']
>>> mc.chain_groups['Forced clearcut and strip cut']
['Forced clearcut']
>>> mc.chain_groups['Forced thinnings']
['Calculate thinning limits', 'Forced first thinning']
>>> mc.branching_groups # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
{'Final_harvest': (('variable', (None, None, True)),
```

```

        ('value', 0.0), ('eq', <function eee at ...>)],
        {'Clearcut or seedtree cut': ['Normal clearcut',
        'Delayed clearcut, 5 years',
        'Delayed clearcut, 10 years']})})
>>> mc.chain_type
'simulation'
>>> len(mc.chains)
3
>>> mc.depth
3
>>> 'SP' in mc.opres_cfiers
True
>>> 'assortment' in mc.opres_cfiers
True
>>> 'something else' in mc.opres_cfiers
True
>>> mc.opres_vars
set(['Volume', 'Income'])
>>> mc.cf_cfiers
set(['SP'])
>>> c = mc.chains[0]
>>> c.condition
>>> c.evaluate_at
'comp_unit'
>>> c.name
'Calculate thinning limits'
>>> len(c.tasks)
1
>>> t = c.tasks[0]
>>> t.name
'Calculate thinning limits'
>>> t.value_fixer
False
>>> t.model # doctest: +ELLIPSIS
<Mock ... PredictionModel>
>>> t.condition_parser
>>> t.validator
>>> p = t.model_param
>>> p.parameters
[None]
>>> p.rect_factor
1.0
>>> p.risk_level
1
>>> p._validator
>>> c = mc.chains[2]
>>> c.condition[0]
('variable', (1, 1, True))
>>> t = c.tasks[0]
>>> 'assortment' in t.opres_cfiers
True
>>> st = t.subtasks[0]
>>> st.condition[3]
('variable', (None, None, True))
>>> st.model # doctest: +ELLIPSIS
<Mock ... OperationModel>
>>> 'assortment' in st.opres_cfiers
True
>>> p = st.model_param
>>> p.cash_flow_table # doctest: +ELLIPSIS
<Mock ... CashFlowTable>
>>> p.simulation_effects

```

```
{'force_step': None, 'erase_memory': False}
>>> t = c.tasks[1]
>>> t.branching_group
'Final_harvest'
>>> t.branching_ops
[3, 3]
>>> for op in t.branch_conditions: print op
...     # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
('Normal clearcut', [('variable', (1, 5, True)),
                    ('value', 0.0),
                    ('eq', <function gte at ...>)])
('Delayed clearcut, 5 years', [('variable', (1, 5, True)),
                              ('value', 5.0),
                              ('eq', <function gte at ...>)])
('Delayed clearcut, 10 years', [('variable', (1, 5, True)),
                               ('value', 10.0),
                               ('eq', <function gte at ...>)])
>>> t.condition # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
[('variable', (1, 4, True)), ('value', 1.0), ('eq', <function eee at ...>),
 ('variable', (1, 3, True)), ('value', 1.0), ('eq', <function eee at ...>),
 ('group', <function and_ at ...>)]
```

Check that the subtask structure is correctly generated:

```
>>> c = mc.chains[1]
>>> t = c.tasks[0]
>>> len(t.subtasks)==2
True
>>> t.subtasks[0].name
'Invalid model'
>>> t.subtasks[1].name
'Assign just thinned dummy to zero to avoid'
>>> mc.memory_models # doctest:
set(['mock model'])
```

Errors generated during conversion:

```
>>> err = list(mcd.errors)
>>> err.sort()
>>> for e in err: print e # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
Model implementation for model 'invalid' of type 'aggregation' .../Invalid model
Model implementation for model 'zero_value' of type 'aggregation' ...
Parameter 'THINNING_LEVEL' is not a valid parameter for prediction model 'mock model' ...
Parameter 'test' is not defined for prediction model 'mock model' ...
Variable 'JUST_THINNED' not found at level 'comp_unit' ...
Variable 'SOME' not found at level 'comp_unit' ...
Variable 'open_area' not found at level 'comp_unit' ...
>>> mcd.ok
{'testxml': False}
```

## 20.2 class ModelChainCollection(Persistent):

A bunch of model chain groups originally in an XML file



## 20.3 class ModelChain(Persistent):

```
def __init__(self, ns, root, validator, chaintype, models, bgroups, opres_cfiers, memory_models):  
=====
```

Construct model chain from the parsed model chain xml.



# TASK.PY

## 21.1 class Task(object):

def \_\_init\_\_(self, ns, elem, validator, cp, taskpath, chainlevel, depth, memory\_models, models=None, bgroup=None): =====

Initialize Task object

### 21.1.1 def \_construct\_chain\_task(self, ns, elem):

Constructs a nested task-subtasks structure

### 21.1.2 def \_get\_model\_impl(self, mtype, model=None, all=False):

Checks that the implementation really exists for the task's model. Optionally returns all implementations of the given type

mtype – model type model – model name all – boolean for returning all implementations

### 21.1.3 def \_set\_model\_and\_param(self, ns, elem):

Extract the model parameter definitions from an XML element if the model exists Stores the model object and creates the model parameter object instance according to the model type



# EXPRPARSER.PY

SIMO optimization expression parser:

```
>>> epsilon = 0.00001
>>> from lxml import etree
>>> import datetime
```

## 22.1 class ExpressionParser(object):

Optimization task expression parser

Parameters:

- validator: SIMO lexicon validator

### 22.1.1 def \_\_init\_\_(self):

```
>>> from simo.builder.optimization.exprparser import ExpressionParser
>>> parser = ExpressionParser()
```

### 22.1.2 def parse(self, expr):

Parse SIMO optimization subobjective or constraint expression into a nested list and further into a Reverse Polish notation / Postfix stack.

Expression is stored as a list of tuples. Each tuple is one of: operation or data variable definition, value definition, or function object. The first item of the tuple defines the type of the expression item. The expression is stored in postfix notation (reverse polish notation).

Variable and operation tuple structure:

- 0: 'variable' or 'operation'
- 1: (level, variable) tuple if 'variable', operation result variable if 'operation'
- 2: (start date, end date) tuple, None if date is -1 (stands for last possible date in the data)
- 3: condition expression, given also in postfix notation, None if no condition
- 4: discount, True if variable values should be discounted, False if not

Value tuple structure: - 0: 'value' - 1: value as float

Function tuple structure: - 0: 'eq', 'ari', or 'aggr' for equality, arithmetic, and aggregation functions respectively  
- 1: function object:

```
>>> initdate = datetime.date(2000, 1, 1)
>>> step = 'year'
```

Parse: sum[period](X):

```
>>> expr = u'sum[1:10] (comp_unit:PV)'
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
('variable', (u'comp_unit', u'PV'), (datetime.date(2000, 1, 1),
datetime.date(2009, 12, 31)), None, False)
('aggr', <function nansum at ...>)
```

Parse: sum[period](X:discount):

```
>>> expr = u'sum[1:10] (comp_unit:PV:discount)'
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
('variable', (u'comp_unit', u'PV'), (datetime.date(2000, 1, 1),
datetime.date(2009, 12, 31)), None, True)
('aggr', <function nansum at ...>)
```

Parse: sum(X) \* 15:

```
>>> expr = u'sum[-1:-1] (comp_unit:PV) * 15.0'
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (u'comp_unit', u'PV'), (None, None), None, False)
('aggr', <function nansum at ...>)
('value', 15.0)
('ari', <function multi at ...>, '*')
```

Parse: sum(X \* Y):

```
>>> expr = u'sum[-1:-1] (comp_unit:PV:discount * comp_unit:AREA)'
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (u'comp_unit', u'PV'), (None, None), None, True)
('variable', (u'comp_unit', u'AREA'), (None, None), None, False)
('ari', <function multi at ...>, '*')
('aggr', <function nansum at ...>)
```

Parse: sum(X / Y):

```
>>> expr = u'sum[-1:-1] (comp_unit:X:discount / comp_unit:Y)'
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (u'comp_unit', u'X'), (None, None), None, True)
('variable', (u'comp_unit', u'Y'), (None, None), None, False)
('ari', <function divide at ...>, '/')
('aggr', <function nansum at ...>)
```

Parse: sum(X \* Y) + sum(Z):

```
>>> expr = u'''sum[-1:-1] (comp_unit:PV:discount * comp_unit:AREA) +
...         sum[1:-1] (operation:cash_flow:discount)'''
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (u'comp_unit', u'PV'), (None, None), None, True)
('variable', (u'comp_unit', u'AREA'), (None, None), None, False)
('ari', <function multi at ...>, '*')
```

```
('aggr', <function nansum at ...>)
('operation', u'cash_flow', (datetime.date(2000, 1, 1), None), None, True)
('aggr', <function nansum at ...>)
('ari', <function plus at ...>, '+')
```

Parse: `mean(X * Y)`:

```
>>> expr = u'mean[-1:-1](operation:Income:discount * comp_unit:AREA)'
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS
('operation', u'Income', (None, None), None, True)
('variable', (u'comp_unit', u'AREA'), (None, None), None, False)
('ari', <function multi at ...>, '*')
('aggr', <function nanmean at ...>)
```

Parse: `sum(X) >= sum(Y)`:

```
>>> expr = u'''sum[-1:-1](comp_unit:PV:discount) ge
...         sum[-1:-1](comp_unit:PV:discount)'''
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (u'comp_unit', u'PV'), (None, None), None, True)
('aggr', <function nansum at ...>)
('variable', (u'comp_unit', u'PV'), (None, None), None, True)
('aggr', <function nansum at ...>)
('eq', <function gee at ...>, 'ge')
```

Parse: `(sum(X) >= sum(Y)) and (min(Z) != 0.0)`:

```
>>> expr = u'''sum[-1:-1](comp_unit:PV:discount) ge
...         sum[-1:-1](comp_unit:PV:discount)
...         and
...         min[-1:-1](operation:Volume) ue 0.0'''
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (u'comp_unit', u'PV'), (None, None), None, True)
('aggr', <function nansum at ...>)
('variable', (u'comp_unit', u'PV'), (None, None), None, True)
('aggr', <function nansum at ...>)
('eq', <function gee at ...>, 'ge')
('operation', u'Volume', (None, None), None, False)
('aggr', <function nanmin at ...>)
('value', 0.0)
('eq', <function nee at ...>, 'ue')
('eq', <function and_ at ...>, 'and')
```

Parse: `sum(X * Y + X * Z)`:

```
>>> expr = u'''sum[-1:-1](comp_unit:AREA * comp_unit:PV + comp_unit:AREA *
...         comp_unit:PV_land)'''
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (u'comp_unit', u'AREA'), (None, None), None, False)
('variable', (u'comp_unit', u'PV'), (None, None), None, False)
('ari', <function multi at ...>, '*')
('variable', (u'comp_unit', u'AREA'), (None, None), None, False)
('variable', (u'comp_unit', u'PV_land'), (None, None), None, False)
('ari', <function multi at ...>, '*')
('ari', <function plus at ...>, '+')
('aggr', <function nansum at ...>)
```

Parse: `sum(Z * (X + Y))`:

```
>>> expr = u'''sum[-1:-1](comp_unit:AREA *
...                        (comp_unit:PV + comp_unit:PV_land))'''
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (u'comp_unit', u'AREA'), (None, None), None, False)
('variable', (u'comp_unit', u'PV'), (None, None), None, False)
('variable', (u'comp_unit', u'PV_land'), (None, None), None, False)
('ari', <function plus at ...>, '+')
('ari', <function multi at ...>, '*')
('aggr', <function nansum at ...>)
```

Parse:  $\text{sum}((X + Y) * (Z + K))$ :

```
>>> expr = u'''sum[-1:-1]((comp_unit:PV + comp_unit:PV_land) *
...                        (comp_unit:AREA + operation:cash_flow))'''
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (u'comp_unit', u'PV'), (None, None), None, False)
('variable', (u'comp_unit', u'PV_land'), (None, None), None, False)
('ari', <function plus at ...>, '+')
('variable', (u'comp_unit', u'AREA'), (None, None), None, False)
('operation', u'cash_flow', (None, None), None, False)
('ari', <function plus at ...>, '+')
('ari', <function multi at ...>, '*')
('aggr', <function nansum at ...>)
```

Parse:  $\text{sum}(X * Y) > \text{sum}(Z * K)$ :

```
>>> expr = u'''sum[20:20](comp_unit:V * comp_unit:AREA) gt
...                    sum[10:10](comp_unit:V * comp_unit:AREA)'''
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
('variable', (u'comp_unit', u'V'), (datetime.date(2019, 1, 1),
datetime.date(2019, 12, 31)), None, False)
('variable', (u'comp_unit', u'AREA'), (datetime.date(2019, 1, 1),
datetime.date(2019, 12, 31)), None, False)
('ari', <function multi at ...>, '*')
('aggr', <function nansum at ...>)
('variable', (u'comp_unit', u'V'), (datetime.date(2009, 1, 1),
datetime.date(2009, 12, 31)), None, False)
('variable', (u'comp_unit', u'AREA'), (datetime.date(2009, 1, 1),
datetime.date(2009, 12, 31)), None, False)
('ari', <function multi at ...>, '*')
('aggr', <function nansum at ...>)
('eq', <function gte at ...>, 'gt')
```

Parse:  $\text{sum}(X) - 0.05 * 1000 * 25 > 5.0$ :

```
>>> expr = u'''sum[1:-1](operation:Volume / comp_unit:AREA) - 0.05 * 1000
...                    gt 5.0'''
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS
('operation', u'Volume', (datetime.date(2000, 1, 1), None), None, False)
('variable', (u'comp_unit', u'AREA'), (datetime.date(2000, 1, 1), None), None, False)
('ari', <function divide at ...>, '/')
('aggr', <function nansum at ...>)
('value', 0.050000000000000003)
('value', 1000.0)
('ari', <function multi at ...>, '*')
('ari', <function minus at ...>, '-')
```



```
('value', 5.0)
('eq', <function gte at ...>, 'gt')
```

Parse: `min((X + Y) * Z) > 1000.0`:

```
>>> expr = u'''min[-1:-1]((comp_unit:PV + comp_unit:PV_land) *
...      comp_unit:AREA) gt 1000.0'''
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS
('variable', (u'comp_unit', u'PV'), (None, None), None, False)
('variable', (u'comp_unit', u'PV_land'), (None, None), None, False)
('ari', <function plus at ...>, '+')
('variable', (u'comp_unit', u'AREA'), (None, None), None, False)
('ari', <function multi at ...>, '*')
('aggr', <function nanmin at ...>)
('value', 1000.0)
('eq', <function gte at ...>, 'gt')
```

Parse: `(sum(X) / sum(Y) > 0.05) and (sum(Z) / sum(K) < 0.15)`:

```
>>> expr = u'''(sum[-1:-1](operation:Volume[SP eq 1]) /
...      sum[-1:-1](operation:Volume) gt 0.05)
...      and
...      (sum[-1:-1](operation:Volume[SP eq 1]) /
...      sum[-1:-1](operation:Volume) lt 0.15)'''
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
('operation', u'Volume', (None, None), [('operation', u'SP'),
('value', 1), ('eq', <function ee at ...>, 'eq')], False)
('aggr', <function nansum at ...>)
('operation', u'Volume', (None, None), None, False)
('aggr', <function nansum at ...>)
('ari', <function divide at ...>, '/')
('value', 0.050000000000000003)
('eq', <function gte at ...>, 'gt')
('operation', u'Volume', (None, None), [('operation', u'SP'),
('value', 1), ('eq', <function ee at ...>, 'eq')], False)
('aggr', <function nansum at ...>)
('operation', u'Volume', (None, None), None, False)
('aggr', <function nansum at ...>)
('ari', <function divide at ...>, '/')
('value', 0.14999999999999999)
('eq', <function lte at ...>, 'lt')
('eq', <function and_ at ...>, 'and')
```

Parse: `sum(X)` with simple conditions for variable X:

```
>>> expr = u'''sum[-1:-1](operation:Income[SP eq 1]:discount)'''
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
('operation', u'Income', (None, None), [('operation', u'SP'),
('value', 1), ('eq', <function ee at ...>, 'eq')], True)
('aggr', <function nansum at ...>)

>>> expr = u'''sum[-1:-1](operation:Income[operation_name eq clearcut])'''
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
('operation', (None, None), [('operation', u'operation_name'),
('value', u'clearcut'), ('eq', <function ee at ...>, 'eq')], False)
('aggr', <function nansum at ...>)

>>> expr = u'''sum[-1:-1](operation:Income[operation_group eq final_harvest])'''
```

```
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
('operation', u'Income', (None, None), [('operation', u'operation_group'),
  ('value', u'final_harvest'), ('eq', <function ee at ...>, 'eq')], False)
('aggr', <function nansum at ...>)
```

Parse: sum(X) with a condition with multiple conditions for variable X:

```
>>> expr = u'''sum[-1:-1](operation:Income[operation_name eq thinning and
...                               SP eq 1 and assortment eq 1]:discount)'''
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
('operation', u'Income', (None, None),
  [('operation', u'operation_name'), ('value', u'thinning'),
  ('eq', <function ee at ...>, 'eq'),
  ('operation', u'SP'), ('value', 1), ('eq', <function ee at ...>, 'eq'),
  ('eq', <function and_ at ...>, 'and'),
  ('operation', u'assortment'), ('value', 1),
  ('eq', <function ee at ...>, 'eq'),
  ('eq', <function and_ at ...>, 'and')],
  True)
('aggr', <function nansum at ...>)
```

Parse:  $(\text{sum}(X * Y) / \text{sum}(X * Y) - 0.1) \geq 0$ , with some conditions:

```
>>> expr = u'''sum[1:1](comp_unit:V[MAIN_SP in (3, 4, 5, 6, 7, 9)] *
...                               comp_unit:AREA) /
...                               sum[1:1](comp_unit:V * comp_unit:AREA) - 0.1 ge 0'''
>>> c,e = parser.parse(expr, initdate, step)
>>> for i in c: print i # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
('variable', (u'comp_unit', u'V'), (datetime.date(2000, 1, 1),
  datetime.date(2000, 12, 31)),
  [('variable', u'MAIN_SP'), ('value', [3, 4, 5, 6, 7, 9]),
  ('eq', <function in_ at ...>, 'in')], False)
('variable', (u'comp_unit', u'AREA'), (datetime.date(2000, 1, 1),
  datetime.date(2000, 12, 31)), None, False)
('ari', <function multi at ...>, '*')
('aggr', <function nansum at ...>)
('variable', (u'comp_unit', u'V'), (datetime.date(2000, 1, 1),
  datetime.date(2000, 12, 31)), None, False)
('variable', (u'comp_unit', u'AREA'), (datetime.date(2000, 1, 1),
  datetime.date(2000, 12, 31)), None, False)
('ari', <function multi at ...>, '*')
('aggr', <function nansum at ...>)
('ari', <function divide at ...>, '/')
('value', 0.10000000000000001)
('ari', <function minus at ...>, '-')
('value', 0.0)
('eq', <function gee at ...>, 'ge')

('variable', (u'comp_unit', u'V'), (datetime.date(2000, 1, 1),
  datetime.date(2000, 12, 31)),
  [('variable', u'MAIN_SP'), ('value', [3, 4, 5, 6, 7, 9]),
  ('eq', <function in_ at 0x1ff4ef0>)], False)
('aggr', <function nansum at 0x1e87a30>)
('variable', (u'comp_unit', u'V'), (datetime.date(2000, 1, 1), datetime.date(2000, 12, 31)), None)
('aggr', <function nansum at 0x1e87a30>)
('ari', <function divide at 0x1ff4e30>)
('value', 0.10000000000000001)
('ari', <function minus at 0x1ff4db0>)
('value', 0.0)
('eq', <function gee at 0x1ff4c30>)
```

Parse expressions with invalid syntax:

```
>>> expr = u'''comp_unit:PV[-1:-1]'''
>>> c,e = parser.parse(expr, initdate, step)
>>> print e
(0, "Expected one of 'sum', 'mean', 'min', 'max', 'avg'")

>>> expr = u'''sum[-1:-1](comp_unit:PV * 10.0)'''
>>> c,e = parser.parse(expr, initdate, step)
>>> print e
(28, 'Expected ":"')

>>> expr = u'''min[-1:-1](comp_unit:PV * comp_unit:AREA > 1000.0)'''
>>> c,e = parser.parse(expr, initdate, step)
>>> print e
(41, 'Expected ")"')
```



# OBJFUNC.PY

SIMO optimization objective, subobjective and utility function definitions.

## 23.1 class ObjectiveFunction(object):

Optimization task objective function definition.

### 23.1.1 def \_\_init\_\_(self, ftype, target, scope):

```
>>> from simo.builder.optimization.objfunc import ObjectiveFunction
>>> obf = ObjectiveFunction("additive", "max", "global")
>>> obf.type
'additive'
>>> obf.target
'max'
>>> obf.scope
'global'
```

## 23.2 class SubObjective(object):

Optimization task subobjective definition.

### 23.2.1 def \_\_init\_\_(self, weight, expr, ufunc, optlevel=None):

```
>>> from simo.builder.optimization.objfunc import SubObjective
>>> sob = SubObjective(1.0, "EXPR", "UFUNC", 0.0)
>>> sob.weight
1.0
>>> sob.expr
'EXPR'
>>> sob.ufunc
'UFUNC'
>>> sob.optlevel
0.0
```

## 23.3 class UFunc(object):

Subobjective utility function.

### 23.3.1 `def __init__(self, ftype, x, y):`

```
>>> from simo.builder.optimization.objfunc import UFunc
>>> ufu = UFunc("linear", 1, 2)
>>> ufu.type
'linear'
>>> ufu.midpoint
True
>>> ufu.coords
(1, 2)
```

# OPTTASK.PY

SIMO Optimization task definition:

```
>>> epsilon = 0.00001
>>> from lxml import etree
>>> import datetime
>>> from simo.builder.optimization.opttask import OptimizationTaskDef
>>> tdf = open('../simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../simulator/xml/schemas/optimization_task.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = '''<optimization_task
...     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
...     xsi:schemaLocation="http://www.simo-project.org/simo
...     ../schemas/optimization_task.xsd"
...     xmlns="http://www.simo-project.org/simo">
...     <initial_date>2008-01-01</initial_date>
...     <period_step>year</period_step>
...     <discount_rate>3.0</discount_rate>
...     <objective_function>
...         <level>comp_unit</level>
...         <type>
...             <function>additive</function>
...             <target>max</target>
...             <scope>global</scope>
...         </type>
...     <sub_objectives>
...         <sub_objective>
...             <weight>1.0</weight>
...             <expression>
...                 sum[1:10] (operation:cash_flow:discount)
...             </expression>
...             <utility_function>
...                 <type>linear</type>
...             </utility_function>
...         </sub_objective>
...     </sub_objectives>
...     <constraints>
...         <constraint>
...             sum[6:10] (operation:cash_flow) /
...             sum[1:5] (operation:cash_flow) lt 1.3
...         </constraint>
...         <constraint>
...             sum[6:10] (operation:cash_flow) /
...             sum[1:5] (operation:cash_flow) gt 0.7
...         </constraint>
...     </constraints>
... </optimization_task>'''
```

```
...         </objective_function>
...     </optimization_task>'''
```

## 24.1 class OptimizationTaskDef(XmlObject):

### 24.1.1 def \_\_init\_\_(self, typedef):

```
>>> class Lexicon(object):
...     def get_level_ind(self, level):
...         return 1
...     def get_variable_ind(self, level, var, active=False):
...         return (1, 1)
>>> otd = OptimizationTaskDef(typedef)
>>> otd.schema = schema
>>> try:
...     otd.xml = xml
... except ValueError, e:
...     print e
name, xml content and lexicon instance must be passed
>>> otd.xml = ('optimization-task', xml, Lexicon())
>>> otd.xml['optimization-task'][0:18]
'<optimization_task'
```

## 24.2 class OptimizationTask(object):

Optimization task definition

### 24.2.1 def \_\_init\_\_(self):

```
>>> opt = otd.obj['optimization-task']
```

### 24.2.2 def \_parse(self, root):

Parse optimization task definition from xml document.

```
>>> opt.init_date
datetime.date(2008, 1, 1)
>>> opt.period_step
'year'
>>> opt.discount_rate
3.0
>>> opt.main_level
'comp_unit'
```

### 24.2.3 def \_parse\_date(self, elem):

Parse initial date definition into datetime object:



```
>>> xml = u'''<initial_date>2008-01-01</initial_date>'''
>>> elem = etree.fromstring(xml)
>>> opt._parse_date(elem)
datetime.date(2008, 1, 1)
>>> xml = u'''<initial_date>today</initial_date>'''
>>> elem = etree.fromstring(xml)
>>> opt._parse_date(elem) == datetime.date.today()
True
```

#### 24.2.4 def \_parse\_objective\_func(self, elem):

Parse objective function definition:

```
>>> opt._objective_func.type
'additive'
>>> opt._objective_func.target
'max'
>>> opt._objective_func.scope
'global'
```

#### 24.2.5 def \_parse\_subobjectives(self, elem):

Parse objective function subobjectives:

```
>>> for i in opt._subobjectives[0].expr: print i # doctest: +ELLIPSIS
('operation', u'cash_flow', (datetime.date(2008, 1, 1), datetime.date(2017, 12, 31)), None, True)
('aggr', <function nansum at ...>)
```

#### 24.2.6 def \_parse\_constraints(self, elem):

Parse optimization task constraint expressions:

```
>>> for i in opt._constraints[0]: print i # doctest: +ELLIPSIS
('operation', u'cash_flow', (datetime.date(2013, 1, 1), datetime.date(2017, 12, 31)), None, False)
('aggr', <function nansum at ...>)
('operation', u'cash_flow', (datetime.date(2008, 1, 1), datetime.date(2012, 12, 31)), None, False)
('aggr', <function nansum at ...>)
('ari', <function divide at ...>, '/')
('value', 1.3)
('eq', <function lte at ...>, 'lt')
>>> for i in opt._constraints[1]: print i # doctest: +ELLIPSIS
('operation', u'cash_flow', (datetime.date(2013, 1, 1), datetime.date(2017, 12, 31)), None, False)
('aggr', <function nansum at ...>)
('operation', u'cash_flow', (datetime.date(2008, 1, 1), datetime.date(2012, 12, 31)), None, False)
('aggr', <function nansum at ...>)
('ari', <function divide at ...>, '/')
('value', 0.69999999999999996)
('eq', <function gte at ...>, 'gt')
>>> print opt.constraint_sizes
[2, 2]
```

#### 24.2.7 def \_get\_expr\_size(self, expr):

Get the dimensions needed for storing expression data:

```
>>> expr = [('variable', ('lev', 'var')),
...         ('variable', ('lev', 'var')),
...         ('eq', lambda x: x != 0)]
>>> opt._get_expr_size(expr)
2
>>> expr = [('variable', ('lev', 'var')),
...         ('variable', ('lev', 'var')),
...         ('eq', lambda x: x != 0),
...         ('operation', ('name',)),
...         ('eq', lambda x: x != 0)]
>>> opt._get_expr_size(expr)
3
```

### 24.2.8 def \_parse\_utility\_function(self, elem):

Parse subobjective utility function definition:

```
>>> xml = u'''<utility_function>
...     <type>linear</type>
... </utility_function>'''
>>> elem = etree.fromstring(xml)
>>> ufunc = opt._parse_utility_function(elem)
>>> ufunc.type
'linear'
```

# AGGRDEF.PY

aggrdef test definitions:

```
>>> from lxml import etree
>>> import datetime
>>> from pprint import pprint
>>> from simo.builder.output.aggrdef import AggregationOutputDef
>>> tdf = open('../simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../simulator/xml/schemas/report_definition.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = '''<?xml version="1.0" encoding="UTF-8"?>
... <report_definition
...   xmlns="http://www.simo-project.org/simo"
...   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
...   xsi:schemaLocation="http://www.simo-project.org/simo
...     ../schemas/report_definition.xsd">
...   <initial_date>2009-01-01</initial_date>
...   <discount_rate>3.0</discount_rate>
...   <reporting_periods>
...     <period>
...       <length>5</length>
...       <unit>year</unit>
...     </period>
...   </reporting_periods>
...   <reports>
...     <report>
...       <expression>sum[0:99] (stratum:SP*stratum:HgM) </expression>
...       <proportion>true</proportion>
...       <group_by>
...         <grouping>
...           <level>stratum</level>
...           <variable>HgM</variable>
...         </grouping>
...         <grouping>
...           <level>stratum</level>
...           <variable>DgM</variable>
...         </grouping>
...       </group_by>
...     </report>
...     <report>
...       <expression>sum[0:99] (operation:Volume) </expression>
...       <proportion>false</proportion>
...     </report>
...   </reports>
... </report_definition>'''
```

## 25.1 class AggregationOutputDef(XmlObject):

Aggregation output definitions.

### 25.1.1 def \_\_init\_\_(self, typedef):

```
>>> aodef = AggregationOutputDef(typedef)
>>> aodef.schema = schema
>>> aodef.xml = ('test data', xml, None)
```

## 25.2 class AggregationOutput(object):

Aggregationoutput info.

### 25.2.1 def \_\_init\_\_(self, ns, root, validator):

```
>>> ao = aodef.obj['test data']
>>> pprint(ao.data) # doctest: +NORMALIZE_WHITESPACE
{'discount_rate': 3.0,
 'initial_date': datetime.date(2009, 1, 1),
 'reporting_periods': [{'length': '5', 'unit': 'year'}],
 'reporting_variables': [{'expression': 'sum[0:99] (stratum:SP*stratum:HgM)',
                          'grouping': {'stratum': ['HgM', 'DgM']},
                          'proportion': True},
                        {'expression': 'sum[0:99] (operation:Volume)',
                          'grouping': {},
                          'proportion': False}]}
```

### 25.2.2 def \_parse(self, root):

Parses the aggregation output instructions into self.data

## EXPRDEF.PY

exprdef test definitions:

```
>>> from lxml import etree
>>> import datetime
>>> from pprint import pprint
>>> from simo.builder.output.exprdef import ExpressionOutputDef
>>> tdf = open('../simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../simulator/xml/schemas/expression_definition.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = '''<?xml version="1.0" encoding="UTF-8"?>
... <expression_definition
...   xmlns="http://www.simo-project.org/simo"
...   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
...   xsi:schemaLocation="http://www.simo-project.org/simo
...     ../schemas/item_definition.xsd">
...   <initial_date>2009-01-01</initial_date>
...   <discount_rate>3.0</discount_rate>
...   <expressions>
...     <item>
...       <expression>sum[-1:-1] (stratum:SP*stratum:HgM) </expression>
...       <label>1st item</label>
...     </item>
...     <item>
...       <expression>sum[-1:-1] (operation:Volume) </expression>
...       <label>2nd item</label>
...     </item>
...   </expressions>
... </expression_definition>'''
```

### 26.1 class ExpressionOutputDef(XmlObject):

Expression output definitions.

#### 26.1.1 def \_\_init\_\_(self, typedef):

```
>>> aodef = ExpressionOutputDef(typedef)
>>> aodef.schema = schema
>>> aodef.xml = ('test data', xml, None)
```

## 26.2 class ExpressionOutput(object):

Expressionoutput info.

### 26.2.1 def \_\_init\_\_(self, ns, root, validator):

```
>>> ao = aodef.obj['test data']
>>> pprint(ao.data) # doctest: +NORMALIZE_WHITESPACE
{'discount_rate': 3.0,
 'expression_variables': [{'expression': 'sum[-1:-1](stratum:SP*stratum:HgM)',
                           'label': '1st item'},
                          {'expression': 'sum[-1:-1](operation:Volume)',
                           'label': '2nd item'}],
 'initial_date': datetime.date(2009, 1, 1)}
```

### 26.2.2 def \_parse(self, root):

Parses the agregation output instructions into self.data

# OUTPUTCONSTRAINT.PY

## 27.1 class OutputConstraintDef(XmlObject):

### 27.1.1 def \_\_init\_\_(self, typedef):

Initializes the output constraint by processing the schema and xml documents:

```
>>> from simo.builder.output.outputconstraint import \
...     OutputConstraintDef
>>> tdf = open('../..simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../..simulator/xml/schemas/output_constraint.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = u'''<output_constraints xmlns="http://www.simo-project.org/simo"
...     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
...     xsi:schemaLocation="http://www.simo-project.org/simo
...     ../schemas/Output_constraint.xsd">
...     <data_level>
...         <name>comp_unit</name>
...         <var_list>AREA SC</var_list>
...     </data_level>
...     <data_level>
...         <name>stratum</name>
...         <var_list>SP Age BA some_exotic_variable</var_list>
...     </data_level>
... </output_constraints>'''
>>> passingxml = u'''<output_constraints
...     xmlns="http://www.simo-project.org/simo"
...     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
...     xsi:schemaLocation="http://www.simo-project.org/simo
...     ../schemas/Output_constraint.xsd">
...     <data_level>
...         <name>comp_unit</name>
...         <var_list>AREA SC</var_list>
...     </data_level>
...     <data_level>
...         <name>stratum</name>
...         <var_list>SP Age BA</var_list>
...     </data_level>
...     <data_level>
...         <name>tree</name>
...         <var_list>*</var_list>
...     </data_level>
... </output_constraints>'''
>>> class Lexicon(object):
```

```
...     def get_level_ind(self, level):
...         levels = {'comp_unit':1, 'stratum':2, 'tree':3}
...         return levels[level]
...     def get_variable_ind(self, level, var, check_active=False):
...         #NB! removed while outputconstrains are inactive
...         #if check_active:
...         if True:
...             if var=='some_exotic_variable':
...                 return (None, None)
...             else:
...                 return (1, 1)
>>> ocd = OutputConstraintDef(typedef)
>>> ocd.schema = schema
>>> try:
...     ocd.xml = ('testxml', xml, Lexicon())
... except ValueError, e:
...     print e
errors in xml to object conversion
>>> ocd.errors # doctest: +NORMALIZE_WHITESPACE
set(["Variable 'some_exotic_variable' not found at level 'stratum' in
lexicon for output constraint 'testxml'"])
>>> # NB! This is the error which we should get if the checking for
>>> # active variables is used
>>> #set(["Variable 'some_exotic_variable' not found at level 'stratum' in
>>> # lexicon or it's not set to be computed during simulation for output
>>> # constraint 'testxml'"])
>>> ocd.xml = ('passingxml', passingxml, Lexicon())
```

### 27.1.2 def xml\_to\_obj(self, root, lexicon):

```
>>> oc = ocd.obj['passingxml']
>>> oc.variables[('comp_unit', 1)]
[('AREA', 1), ('SC', 1)]
>>> oc.variables[('stratum', 2)]
[('SP', 1), ('Age', 1), ('BA', 1)]
>>> oc.variables[('tree', 3)] is None
True
>>> ocd.errors
set([])
>>> ocd.warnings
[]
```



# RANDOMVALUES.PY

SIMO random value definition:

```
>>> epsilon = 0.00001
>>> from lxml import etree
>>> import datetime
>>> from simo.builder.simulation.randomvalues import RandomValueDef
>>> tdf = open('../simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../simulator/xml/schemas/random_variables.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = '''<random_variable_schemes
...     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
...     xsi:noNamespaceSchemaLocation="../schemas/random_variables.xsd">
...         <scheme name="simple error">
...             <variable>
...                 <name>BA</name>
...                 <level>stratum</level>
...                 <effect>
...                     <random_error>
...                         <allow_negative>false</allow_negative>
...                         <unit>absolute</unit>
...                     </random_error>
...                 </effect>
...                 <random_value_generation>
...                     <target_level>stratums</target_level>
...                     <distribution>
...                         <normal_dist>
...                             <mean>0.0</mean>
...                             <stdev>1.0</stdev>
...                         </normal_dist>
...                     </distribution>
...                 </random_value_generation>
...             </variable>
...         </scheme>
...         <scheme name="complex error">
...             <model>
...                 <name>my_error_model</name>
...                 <implemented_at>error_models.py</implemented_at>
...                 <level>stratum</level>
...                 <variables>
...                     <variable>D_gM</variable>
...                     <variable>H_gM</variable>
...                 </variables>
...             </model>
...         </scheme>
...     </random_variable_schemes>'''
```

## 28.1 class RandomValueDef(XmlObject):

Random variable definitions definitions.

### 28.1.1 def \_\_init\_\_(self, typedef):

```
>>> class Lexicon(object):
...     def get_level_ind(self, level):
...         return 1
...     def get_variable_ind(self, level, var, active=False):
...         return (1, 1)
>>> rdef = RandomValueDef(typedef)
>>> rdef.schema = schema
>>> rdef.xml = ('Test schemes', xml, Lexicon())
```

## 28.2 class RandomValue(object):

Random variable definitions definitions.

### Properties:

- schemes: return the schemes dictionary

### 28.2.1 def \_\_init\_\_(self, ns, root, validator):

```
>>> rv = rdef.obj['Test schemes']
>>> for x, y in rv.schemes.items():
...     print x, y # doctest:+ELLIPSIS
complex error [<simo.builder.simulation.randomvalues.ErrorModel object at ...>]
simple error [<simo.builder.simulation.randomvalues.RandomVariable object at ...>]
```

### 28.2.2 def \_parse(self, root):

Parse all random value generation schemes. A scheme consists of a single or multiple random variables and error models.

## 28.3 class ErrorModel(object):

Class for defining a random error model for SIMO Monte Carlo simulations.

### 28.3.1 def \_\_init\_\_(self, elem, ns, validator):

## 28.4 class RandomVariable(object):

Class for defining a random variable in SIMO Monte Carlo simulations.

**28.4.1 def \_\_init\_\_(self, elem, ns, validator):**

Initialize random variable definition.

**28.4.2 def \_parse\_effect(self, elem):**

Parse random value effect definition. Random values can have one of the following effects: random error, classification error, removal of object.

**28.4.3 def \_parse\_removalprob(self, elem):**

Parse removal probability definition for object.

**28.4.4 def \_parse\_error\_matrix(self, elem):**

Parse error matrix for simulating classification errors.

**28.4.5 def \_parse\_rand\_generation(self, elem):**

Parse random value generation definition.

**28.5 class RandomError(object):**

Class for defining initial random error for a variable.

**28.5.1 def \_\_init\_\_(self, unit, allowneg):**

Initialize random error definition

**28.6 class ClassificationError(object):**

Class for defining classification error for a categorical variable.

**28.6.1 def \_\_init\_\_(self, labels, errmatrix):**

Initialize classification error definition.

**28.7 class RemovalProbability(object):**

Class for defining a removal probability for an object.

**28.7.1 def \_\_init\_\_(self, type, threshold):**

Initialize removal probability definition.

## **28.8 class Sigmoid(object):**

Sigmoid function shape for removal probability.

**28.8.1 def \_\_init\_\_(self, alpha, beta):**

## **28.9 class UniformDist(object):**

**28.9.1 def \_\_init\_\_(self, min, max):**

## **28.10 class NormalDist(object):**

**28.10.1 def \_\_init\_\_(self, mean, stdev):**

## **28.11 class LognormalDist(object):**

**28.11.1 def \_\_init\_\_(self, mean, stdev):**

## **28.12 class WeibullDist(object):**

**28.12.1 def \_\_init\_\_(self, scale, shape):**

## **28.13 class BetaDist(object):**

**28.13.1 def \_\_init\_\_(self, alpha, beta):**

## **28.14 class Systematic(object):**

**28.14.1 def \_\_init\_\_(self, fromval, toval, step):**

# SIMCONTROL.PY

## 29.1 class SimControlDef(XmlObject):

### 29.1.1 def \_\_init\_\_(self, typedef):

Initializes the simulation control by processing the schema and xml documents:

```
>>> from simo.builder.simulation.simcontrol import SimControlDef
>>> from simo.builder import names
>>> from minimock import Mock
>>> tdf = open('../../simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../../simulator/xml/schemas/simulation.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = u'''<simulation
...     xmlns="http://www.simo-project.org/simo"
...     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
...     xsi:schemaLocation="http://www.simo-project.org/simo
...     ../schemas/simulation.xsd" desc="first test in SIMO">
...     <main_level>comp_unit</main_level>
...     <built_in_attributes>
...         <time_step>time_step</time_step>
...         <year>year</year>
...     </built_in_attributes>
...     <control>
...     <growth_season_start_date>--03-15</growth_season_start_date>
...     <growth_season_end_date>--07-15</growth_season_end_date>
...         <span>
...             <time>
...                 <time_step>1</time_step>
...                 <operation_step>1</operation_step>
...                 <time_unit>year</time_unit>
...                 <steps>10</steps>
...             </time>
...             <init_chains>
...                 <chain>Chain1</chain>
...                 <chain>Chain2</chain>
...             </init_chains>
...             <simulation_chains>
...                 <chain>Chain3</chain>
...             </simulation_chains>
...             <operation_chains>
...                 <chain>Chain4</chain>
...                 <chain>Chain5</chain>
...             </operation_chains>
...     </control>
... </simulation>'''
```

```
...             <forced_operation_chains>
...                 <chain>Chain6</chain>
...             </forced_operation_chains>
...         </span>
...     </control>
... <stop_logic>comp_unit:DEV_CLASS eq 5.0 or comp_unit:V gt 300
... </stop_logic>
...     <init_variables>
...         <variable>
...             <name>DIAM_CLASS_WIDTH</name>
...             <level>simulation</level>
...             <value>1</value>
...         </variable>
...         <variable>
...             <name>passed_thinning_limit</name>
...             <level>comp_unit</level>
...             <value>0</value>
...         </variable>
...         <variable>
...             <name>passed_regen_limit</name>
...             <level>comp_unit</level>
...             <value>0</value>
...         </variable>
...     </init_variables>
...     <output_constraints active="on">
...         <level>
...             <name>comp_unit</name>
...             <variables>
...                 <variable>AREA</variable>
...                 <variable>DEVEL_CLASS</variable>
...                 <variable>MAIN_GROUP</variable>
...             </variables>
...         </level>
...         <level>
...             <name>stratum</name>
...             <variables>
...                 <variable>SP</variable>
...             </variables>
...         </level>
...     </output_constraints>
... </simulation>'''
>>> class Lexicon(object):
...     def get_level_ind(self, level):
...         if level=='comp_unit':
...             return 1
...         elif level=='simulation':
...             return 0
...         elif level=='stratum':
...             return 2
...         else:
...             return 3
...     def get_variable_ind(self, level, var, check_active=False):
...         if var=='some_exotic_variable':
...             return (None, None)
...         else:
...             return (1, 1)
```

```

>>> mcc1 = Mock('ModelChainCollection')
>>> mcc1.depth = 10
>>> mcc1.opres_cfiers = set([])
>>> mcc1.opres_vars = set([])
>>> mcc1.cf_cfiers = set([])
>>> mcc1.memory_models = set([])
>>> mcc1.name = 'test-chain-1'
>>> mcc2 = Mock('ModelChainCollection')
>>> mcc2.depth = 15
>>> mcc2.branching_groups = {'bg1':{'bt1':['cond1', 'cond2']},
...                           'bg2':{'bt2':['cname', 'cother']}}
>>> mcc2.opres_cfiers = set(['SP'])
>>> mcc2.opres_vars = set(['Volume'])
>>> mcc2.cf_cfiers = set(['SP'])
>>> mcc2.memory_models = set(['mock model 1', 'mock model 5'])
>>> mcc2.name = 'test-chain-2'
>>> mcc3 = Mock('ModelChainCollection')
>>> mcc3.depth = 5
>>> mcc3.branching_groups = {'bg2':{'bt1':['cond1', 'cond2']},
...                           'bg4':{'bt2':['cname', 'cother']}}
>>> mcc3.opres_cfiers = set(['SP'])
>>> mcc3.opres_vars = set(['Volume', 'Income'])
>>> mcc3.cf_cfiers = set(['SP'])
>>> mcc3.memory_models = set(['mock model 2'])
>>> mcc3.name = 'test-chain-3'
>>> mcc4 = Mock('ModelChainCollection')
>>> mcc4.depth = 10
>>> mcc4.opres_cfiers = set(['SP', 'assortment'])
>>> mcc4.opres_vars = set(['Volume', 'Income'])
>>> mcc4.cf_cfiers = set(['SP', 'assortment'])
>>> mcc4.memory_models = set(['mock model 2', 'mock model 3'])
>>> mcc4.name = 'test-chain-4'
>>> mcs = {names.INIT: {'Chain1': mcc1, 'Chain2': mcc1},
...        names.SIMULATION: {'Chain3': mcc1},
...        names.OPERATION: {'Chain4': mcc2, 'Chain5': mcc3},
...        names.FORCED_OPERATION: {'Chain6': mcc4}}
>>> scd = SimControlDef(typedef)
>>> scd.schema = schema
>>> try:
...     scd.xml = ('testxml', xml, Lexicon(), mcs)
... except ValueError, e:
...     pass

```

### 29.1.2 def xml\_to\_obj(self, root, lexicon, simodb):

NB! output constraints have no effect in the simulation in the current implementation; i.e., all data variables are written to the result database

TODO: remove output constraints from simcontrol, if the current implementation stays:

```

>>> sc = scd.obj['testxml']
>>> sc.main_level
1
>>> sc.built_ins
{'time_step': (1, 1), 'year': (1, 1)}
>>> sc.growth_season_end_month
7
>>> sc.growth_season_end_day
15
>>> sc.stop_logic # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
[('variable', (1, 1, True)), ('value', 5.0), ('eq', <function eee...),

```

```
('variable', (1, 1, True)), ('value', 300.0), ('eq', <function gte...>),
('group', <function or_...>)]
>>> span = sc.time_spans[0]
>>> span.branching_groups
{'bg1': {'bt1': ['cond1', 'cond2']}, 'bg2': {'bt2': ['cname', 'cother']}, 'bg4': {'bt2': ['cname',
>>> span.chain_collections['forced_operation'] # doctest: +ELLIPSIS
[<Mock ... ModelChainCollection>]
>>> span.chain_collections['init'] # doctest: +ELLIPSIS
[<Mock ... ModelChainCollection>, <Mock ... ModelChainCollection>]
>>> names.SIMULATION in span.chain_collections
True
>>> names.OPERATION in span.chain_collections
True
>>> span.ending.target
10
>>> span.ending.type
'steps'
>>> span.forced_operations
True
>>> span.operation_step
1
>>> span.time_step
1
>>> span.unit
'year'
>>> sc.forced_operations
True
>>> sc.max_model_chain_depth
15
>>> sc.all_branching_groups # doctest: +NORMALIZE_WHITESPACE
{'bg1': {'bt1': ['cond1', 'cond2']},
'bg2': {'bt1': ['cond1', 'cond2']},
'bg4': {'bt2': ['cname', 'cother']}}
>>> mm = list(sc.memory_models)
>>> mm.sort()
>>> mm
['mock model 1', 'mock model 2', 'mock model 3', 'mock model 5']
>>> 'assortment' in sc.opres_cfiers
True
>>> 'SP' in sc.opres_cfiers
True
>>> 'SP' in sc.cf_cfiers
True
>>> sc.opres_vars
set(['Volume', 'Income'])
>>> sc.init_variables
[((1, 1), 1.0), ((1, 1), 0.0), ((1, 1), 0.0)]
>>> sc.output_constraints[1]
[1, 1, 1]
>>> sc.output_constraints[2]
[1]
>>> sc.output_constraints[3] is None
True
>>> scd.errors # doctest: +NORMALIZE_WHITESPACE
set(["Duplicate branching group name 'bg2' in operation model chains
'test-chain-2' and 'test-chain-3' (1. timespan) for Simulation control
'testxml'"])
>>> scd.warnings
[]
```



# LEXICONTRANSLATIONTABLE.PY

## 30.1 class LexiconTransTableDef(XmlObject):

Provides lexicon data level and variable name translations for log messages.

### 30.1.1 def \_\_init\_\_(self, typedef, schema, xmldata):

Initializes the translation by processing the schema and xml documents for the lexicon translation definitions:

```
>>> from simo.builder.translation.lexicontranslationtable import LexiconTransTableDef
>>> tdf = open('../..simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../..simulator/xml/schemas/lexicon_translation_table.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = u'''<lexicon_translation_table
...     xmlns="http://www.simo-project.org/simo"
...     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...     <level>
...         <name>
...             <from>comp_unit</from>
...             <to>
...                 <lang name="fi">kuvio</lang>
...                 <lang name="se">n nting</lang>
...             </to>
...         </name>
...         <variables>
...             <variable>
...                 <name>
...                     <from>DEVEL_CLASS</from>
...                     <to>
...                         <lang name="fi">kehitysluokka</lang>
...                     </to>
...                 </name>
...                 <values>
...                     <value>
...                         <from>Open regeneration site</from>
...                         <to>
...                             <lang name="fi">aukea</lang>
...                         </to>
...                     </value>
...                     <value>
...                         <from>Young seedling stand (less than 1,3 m)</from>
...                         <to>
...                             <lang name="fi">pieni taimikko</lang>
```

```
...         </to>
...         </value>
...     </values>
... </variable>
... <variable>
...     <name>
...         <from>INVENTORY_METHOD</from>
...         <to>
...             <lang name="fi">inventointitapa</lang>
...             <lang name="se">så ska det vara</lang>
...         </to>
...     </name>
...     <values>
...         <value>
...             <from>some method</from>
...             <to>
...                 <lang name="fi">joku inventointi</lang>
...                 <lang name="se">ju inventering ja</lang>
...             </to>
...         </value>
...     </values>
... </variable>
... </variables>
... </level>
... </lexicon_translation_table>'''
>>> ltd = LexiconTransTableDef(typedef)
>>> ltd.schema = schema
>>> ltd.xml = ('testxml', xml, None)
>>> ltd.xml['testxml'][:26]
u'<lexicon_translation_table'
```

### 30.1.2 def xml\_to\_obj(self, root, lexicon):

Construct lexicon data level name and variable name translation tables from the parsed translation table xml.

Only one translation table, keyed as 'default' is supported.

The lexicon data level name table is a dictionary (by data level name in single quotes) of language codes yielding unicode translation strings:

```
>>> lt = ltd.obj['testxml']
>>> lt.level_table["'comp_unit'"]['fi']
u''kuvio''
```

The data level variable table is similar dictionary yielding unicode translation strings:

```
>>> lt.var_table["'DEVEL_CLASS'"]['fi']
u''kehitysluokka''
```

## 30.2 class LexiconTransTable(object):

Lexicon translation definitions.

Attributes:

- level\_table: dictionary, level name as key, dictionary of language code: level name translation as data
- var\_table: dictionary, variable as key, dictionary of language code: variable name translation as data

**30.2.1 def \_\_init\_\_(self, ns, elem):**

Parses the XML data into a class instance



# MESSAGETRANSLATIONTABLE.PY

## 31.1 class MsgTransTableDef(XmlObject):

Provides log message translations.

### 31.1.1 def \_\_init\_\_(self, typedef, schema, xmldata):

Initializes the translation by processing the schema and xml documents for the message translation definitions.

```
>>> from simo.builder.translation.message translationtable import MsgTransTableDef
>>> tdf = open('../..simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../..simulator/xml/schemas/message_translation_table.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = u'''<message_translation_table
...     xmlns="http://www.simo-project.org/simo"
...     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...         <log_level name="test">
...             <logger name="simulation">
...                 <message>
...                     <from>Variable (.+)</from>
...                     <to>
...                         <lang name="fi">Muuttuja $1</lang>
...                         <lang name="se">Och det samma på svenska</lang>
...                     </to>
...                 </message>
...             </logger>
...             <logger name="optimization">
...                 <message>
...                     <from>something</from>
...                     <to>
...                         <lang name="fi">jotakin</lang>
...                         <lang name="se">någonting</lang>
...                         <lang name="de">etwas</lang>
...                     </to>
...                 </message>
...             </logger>
...         </log_level>
...     </message_translation_table>'''
>>> mtd = MsgTransTableDef(typedef)
>>> mtd.schema = schema
>>> mtd.xml = ('testxml', xml, None)
>>> mtd.xml['testxml'][:26]
u'<message_translation_table'
```

### 31.1.2 def xml\_to\_obj(self, root, lexicon):

Construct message translation table from the parsed translation table xml. The translation table is a dictionary keyed by log message level, which in turn is a dictionary keyed by the logger name. The items of this dictionary are two item tuples: (i) a list of tuples of regular expression pattern and corresponding original message string, (ii) a dictionary with original message strings as keys, and dictionaries with translated messages as data items and language identifiers as keys.

```
>>> mt = mtd.obj['testxml']
>>> mt.translation['test']['optimization'][0] # doctest: +ELLIPSIS
[(<_sre.SRE_Pattern object at 0x...>, 'something')]
>>> mt.translation['test']['optimization'][1]
{'something': {'fi': u'jotakin', 'de': u'etwas', 'se': u'n\xc3\xa5gonting'}}
```

## 31.2 class MsgTransTable(object):

Log message translation definitions

Attributes:

- translation: dictionary keyed by log message level, has
- dictionary keyed by log message origin, has
- a tuple of:
  - list of tuples of: regular expression patters (index 0) matching the original message (index)
  - dictionary keyed by the original message has
  - dictionary keyed by the language code has the translated message as data

### 31.2.1 def \_\_init\_\_(self, ns, elem):

Parses the XML data into a class instance

Database modules:

Relational database for storing the input and result data for simulation and optimization and an object database for storing the xml and Python descriptions of the simulator.

# DB.PY

## 32.1 class SQLiteDB(object):

This object is mainly to be used as a base class. It will connect to the given database file (and wipe it if need be). It also houses a close method for when you're done (and an open if you decide to continue later, in which case a new connection to the original file is formed)

### 32.1.1 def \_\_init\_\_(self, db\_path='default.db', wipe=False, memory=False):

Connects to the given database file. Destroys the old one first if wipe = True. Stores to db in memory only if memory is set to True:

```
>>> from simo.db.datadb import db
>>> from pysqlite2 import dbapi2 as sqlite3
>>> test = db.SQLiteDB('test_mock.db', wipe=True)
>>> sql = 'CREATE TABLE "test" (id INTEGER PRIMARY KEY, message TEXT)'
>>> c = test.conn.execute(sql)
>>> sql = 'INSERT INTO "test" (message) VALUES ("Test text")'
>>> c = test.conn.execute(sql)
>>> iter = test.conn.execute('SELECT "test".* FROM "test"')
>>> for row in iter:
...     print row
(1, u'Test text')
>>> test.close()
>>> test = db.SQLiteDB(memory=True)
>>> sql = "CREATE TABLE test (id INTEGER PRIMARY KEY, message TEXT)"
>>> c = test.conn.execute(sql)
>>> c = test.conn.execute('INSERT INTO test (message) VALUES ("Test text")')
>>> iter = test.conn.execute('SELECT * FROM test')
>>> for row in iter:
...     print row
(1, u'Test text')
>>> test.close()
>>> test.open()
>>> try:
...     iter = test.conn.execute('SELECT * FROM test')
... except sqlite3.OperationalError, e:
...     print e
no such table: test
>>> test.close()
```

### 32.1.2 def open(self):

Re-opens a connection to the database file given in init.

```
>>> test = db.SQLiteDB(db_path='test_mock.db', wipe=False)
>>> test.close()
>>> try:
...     cursor = test.conn.cursor()
... except sqlite3.ProgrammingError, e:
...     print 'Database closed;', e.args[0]
Database closed; Cannot operate on a closed database.
>>> test.open()
>>> iter = test.conn.execute('SELECT * FROM test')
>>> for row in iter:
...     print row
(1, u'Test text')
```

### 32.1.3 def close(self):

Commits any lingering changes and closes the database connection.

```
>>> test = db.SQLiteDB(db_path='test_mock.db', wipe=False)
>>> c = test.conn.execute('CREATE TABLE fun (id INTEGER PRIMARY KEY)')
>>> c = test.conn.execute('INSERT INTO fun VALUES (1)')
>>> test.close()
>>> test.open()
>>> iter = test.conn.execute('SELECT * FROM fun')
>>> for row in iter:
...     print row
(1,)
>>> test.close()
```

## 32.2 class DataDB(SQLiteDB):

The object will connect to a given database file on init or create one if needed, using the provided Lexicon for the table structure. It will also create a log table on initialization. If the `make_new_db` parameter (default True) is not set to False, an existing database, if any, will be destroyed.

```
def __init__(self, db_type, content_def, hierarchy, level_meta, opres_def, cf_cfiers, logger,
db_path='default.db', make_new_db=False, memory=False, spatial=False, constraints=None):
=====
```

Connects to the given database file. Destroys the old file and sets up the new one with the given lexicon, if `make_new_db==True` and sets up initial values (`self.content_def`, `self.hierarchy`), will complain of changed schema if `db_type` is 'write' and `make_new_db` is False and the schema (`content_def`) has changed:

```
>>> from simo.db.datadb.test import mock_db
>>> from pprint import pprint
>>> hierarchy = mock_db.hierarchy
>>> content_def = mock_db.base_cd
>>> levels = mock_db.base_levels
>>> opres_def = mock_db.opres_def
>>> cf_cfiers = mock_db.cf_cfiers
>>> test = db.DataDB('write', content_def, hierarchy, levels, opres_def,
...                 cf_cfiers, mock_db.MockLogger(), 'test_mock.db',
...                 make_new_db=True, memory=False, spatial=True)
```

### 32.2.1 def \_create\_meta\_tables(self):

Sets up tables for storing the numerical and categorical attribute definitions



### 32.2.2 def \_set\_meta\_tables(self, level\_meta):

Writes numerical and categorical variable definitions to the meta tables:

```
>>> res = test.conn.execute('SELECT * from META_NUMERICAL').fetchall()
>>> pprint(res)
[(1, u'tree', u'd', None, None, None, u'Diameter...'),
 (2, u'sample_esplot', u'BA', None, None, None, u'Basal area...'),
 (3,
  u'simulation',
  u'DIAM_CLASS_WIDTH',
  None,
  None,
  None,
  u'Width of a single class...'),
 (4, u'sample_tree', u'h', None, None, None, u'Height...'),
 (5, u'sample_plot', u'BA', None, None, None, u'Basal area...'),
 (6, u'stratum', u'N', None, None, None, u'Number of stems...')]
>>> res = test.conn.execute('SELECT * from META_CATEGORICAL').fetchall()
>>> pprint(res)
[(1, u'stratum', u'SP', None, None, u'Tree species...'),
 (2, u'comp_unit', u'SC', None, None, u'Site class...')]
>>> sql = 'SELECT * from META_CATEGORICAL_VALUE'
>>> res = test.conn.execute(sql).fetchall()
>>> pprint(res)
[(1, 1, 1.0, u'Pine'), (2, 1, 2.0, u'Spruce'), (3, 2, 1.0, u'Very good')]
```

### 32.2.3 def \_create\_tabledef(self, defkey, content\_def=None, levels=None):

Creates table definitions for each data level in content definition. Table definition is stored keyed with the level name and stores the level index, parent level name, attribute names and corresponding matrix indices, optional geometry type of the level, and a set of attributes names that are of type TEXT:

```
>>> metadata = test.conn.execute('PRAGMA table_info(comp_unit)').fetchall()
>>> set(metadata) == set([(0, u'id', u'TEXT', 1, None, 1),
...                      (1, u'oid', u'TEXT', 1, None, 0),
...                      (2, u'iteration', u'INTEGER', 1, None, 1),
...                      (3, u'branch', u'INTEGER', 1, None, 1),
...                      (4, u'pid', u'TEXT', 1, None, 0),
...                      (5, u'date', u'DATE', 1, None, 1),
...                      (6, u'SC', u'REAL', 0, None, 0),
...                      (7, u'StandLabel', u'TEXT', 0, None, 0),
...                      (8, u'geom', u'MULTIPOLYGON', 0, None, 0)])
True
>>> cu = test.table_def['comp_unit']
>>> assert cu['attr ind'] == [0, 1]
>>> assert cu['attr name'] == [u'SC', u'StandLabel']
>>> assert cu['columns'] == set([u'iteration', u'oid', u'pid',
...                             u'StandLabel', u'SC',
...                             u'geom', u'branch', u'date', u'id'])
>>> assert cu['create attr'] == u' "SC" REAL,\n "StandLabel" TEXT,\n'
>>> assert cu['float attr ind'] == [0]
>>> assert cu['float attr name'] == [u'SC']
>>> assert cu['float insert sql'] == u'INSERT INTO comp_unit '\
...                                '(id, oid, iteration, branch, '\
...                                'pid, date, SC) '\
...                                'VALUES (?, ?, ?, ?, ?, ?, ?)'\
>>> assert cu['geom type'] == 'MULTIPOLYGON'
>>> assert cu['geom update sql'] == 'UPDATE comp_unit '\
...                                'SET geom=GeomFromText(?, ?) WHERE '\
```

```
...                                     'id=? AND iteration=? AND branch=? and date=?'
>>> assert cu['insert sql'] == u'INSERT INTO comp_unit (id, oid, '\
...                                     'iteration, branch, pid, date, '\
...                                     'SC, StandLabel) '\
...                                     'VALUES (?, ?, ?, ?, ?, ?, ?)'
>>> assert cu['level'] == 2
>>> assert cu['other attr name'] == [u'StandLabel']
>>> assert cu['parent name'] == 'estate'
>>> assert cu['text vars'] == set(['StandLabel'])
>>> assert cu['to remove'] == [1]
```

### 32.2.4 def \_get\_parent\_level(self, my\_hierarchy):

Will get a parent\_level value from a given hierarchy item (from self.lexicon.hierarchy) or None, if the item is root (level 0):

```
>>> test._get_parent_level(test.hierarchy[2])
1
>>> test._get_parent_level(test.hierarchy[0])
```

### 32.2.5 def \_create\_table(self, lname):

Based on the table definitions, creates new tables in the db. For existing tables adds new columns if the content definition has changed since the table was created:

```
>>> sql = "select sql from sqlite_master where type='table' and name='tree'"
>>> iter = test.conn.execute(sql)
>>> for item in iter:
...     print item[0]
CREATE TABLE "tree"
  (id TEXT NOT NULL,
  oid TEXT NOT NULL,
  iteration INTEGER NOT NULL,
  branch INTEGER NOT NULL,
  pid TEXT NOT NULL,
  date DATE NOT NULL,
  "d" REAL, "geom" POINT,
  CONSTRAINT pk_id_branch PRIMARY KEY
    (id, iteration, branch, date),
  CONSTRAINT fk_pid FOREIGN KEY (pid)
    REFERENCES "stratum" (id))
```

If the schema, i.e. the lexicon on which the database is based, is changed between runs and make\_new\_db is set to False, an error is triggered:

```
>>> test.close()
>>> content_def = mock_db.aug_cd
>>> levels = mock_db.aug_levels
>>> try:
...     test = db.DataDB('write', content_def, hierarchy, levels,
...                       opres_def, cf_cfiers, mock_db.MockLogger(), 'test_mock.db',
...                       make_new_db=False, memory=False, spatial=True)
... except ValueError, e:
...     print e[0] # doctest: +NORMALIZE_WHITESPACE
Lexicon has changed since table was created.
Run schema migration tool for the database
>>> test.close()
>>> test = db.DataDB('write', content_def, hierarchy, levels, opres_def,
```

```
...         cf_cfiers, mock_db.MockLogger(), 'test_mock.db',
...         make_new_db=True, memory=False, spatial=True)
```

### 32.2.6 def add\_data\_from\_matrix(self, ddate, matrix, ind2id, links, blocked, base\_level=1, update=False, start\_iter=0):

From the base level down, writes the data from the matrix using the matrix index mapping stored in the table def. If base level is not 1, and the type is update, copies the level data between 1 and the base level to new dates as well, if no data at the base level remains for the old date(s), removes the data from levels between 1 and base level for the old date(s). When not updating the data from matrix is written for the parent level objects. Starting iteration is added to the original iteration index.

NB! in update mode the geometry and text attribute content is lost below base level. When not updating, they are lost from all levels:

```
>>> test.add_data_from_matrix(mock_db.matrixdates, mock_db.datamatrix,
...                           mock_db.Ind2Id(), mock_db.links, [], mock_db.base_level)
>>> iter = test.conn.execute("SELECT * FROM simulation")
>>> for item in iter:
...     print item # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
(u'siml', u'siml', 0, 1, None, datetime.date(1, 1, 1), 1.0)
(u'siml', u'siml', 0, 1, None, datetime.date(1, 2, 2), 1.0)
(u'siml', u'siml', 1, 0, None, datetime.date(1, 1, 1), 1.0)
(u'siml', u'siml', 1, 0, None, datetime.date(1, 2, 2), 1.0)
(u'siml', u'siml', 0, 0, None, datetime.date(1, 1, 1), 1.0)
(u'siml', u'siml', 0, 0, None, datetime.date(1, 2, 2), 1.0)

>>> sql = 'SELECT * FROM estate ORDER BY iteration, branch, id'
>>> iter = test.conn.execute(sql)
>>> for item in iter:
...     print item # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
(u'estatel', u'estatel', 0, 0, u'siml', datetime.date(1, 1, 1), None)
(u'estatel', u'estatel', 0, 0, u'siml', datetime.date(1, 2, 2), None)
(u'estatel', u'estatel', 0, 1, u'siml', datetime.date(1, 1, 1), None)
(u'estatel', u'estatel', 0, 1, u'siml', datetime.date(1, 2, 2), None)
(u'estatel', u'estatel', 1, 0, u'siml', datetime.date(1, 1, 1), None)
(u'estatel', u'estatel', 1, 0, u'siml', datetime.date(1, 2, 2), None)

>>> sql = 'SELECT * FROM comp_unit ORDER BY iteration, branch, id'
>>> iter = test.conn.execute(sql)
>>> for item in iter:
...     print item # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
(u'standl', u'standl', 0, 0, u'estatel', datetime.date(1, 1, 1),
10.0, 1.0, None, None)
(u'stand2', u'stand2', 0, 0, u'estatel', datetime.date(1, 2, 2),
11.0, 1.0, None, None)
(u'standl', u'standl', 0, 1, u'estatel', datetime.date(1, 1, 1),
20.0, 2.0, None, None)
(u'stand2', u'stand2', 0, 1, u'estatel', datetime.date(1, 2, 2),
21.0, 2.0, None, None)
(u'standl', u'standl', 1, 0, u'estatel', datetime.date(1, 1, 1),
90.0, 9.0, None, None)
(u'stand2', u'stand2', 1, 0, u'estatel', datetime.date(1, 2, 2),
91.0, 9.0, None, None)

>>> sql = 'SELECT * FROM stratum ORDER BY iteration, branch, id'
>>> iter = test.conn.execute(sql)
>>> for item in iter:
...     print item # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
```

```
(u'stratum1-1', u'stratum1-1', 0, 0, u'stand1', datetime.date(1, 1, 1),
100.0, 1.0)
(u'stratum1-2', u'stratum1-2', 0, 0, u'stand1', datetime.date(1, 1, 1),
110.0, 1.0)
(u'stratum2-1', u'stratum2-1', 0, 0, u'stand2', datetime.date(1, 2, 2),
120.0, 1.0)
(u'stratum2-2', u'stratum2-2', 0, 0, u'stand2', datetime.date(1, 2, 2),
130.0, 1.0)
(u'stratum1-1', u'stratum1-1', 0, 1, u'stand1', datetime.date(1, 1, 1),
200.0, 2.0)
(u'stratum1-2', u'stratum1-2', 0, 1, u'stand1', datetime.date(1, 1, 1),
210.0, 2.0)
(u'stratum1-1', u'stratum1-1', 1, 0, u'stand1', datetime.date(1, 1, 1),
900.0, 9.0)
(u'stratum1-2', u'stratum1-2', 1, 0, u'stand1', datetime.date(1, 1, 1),
910.0, 9.0)
(u'stratum2-1', u'stratum2-1', 1, 0, u'stand2', datetime.date(1, 2, 2),
920.0, 9.0)
(u'stratum2-2', u'stratum2-2', 1, 0, u'stand2', datetime.date(1, 2, 2),
930.0, 9.0)

>>> sql = 'SELECT * FROM tree ORDER BY iteration, branch, id'
>>> iter = test.conn.execute(sql)
>>> for item in iter:
...     print item # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
(u'tree1-1-1', u'tree1-1-1', 0, 0, u'stratum1-1', datetime.date(1, 1, 1),
11.0, None)
(u'tree1-2-1', u'tree1-2-1', 0, 0, u'stratum1-2', datetime.date(1, 1, 1),
12.0, None)
(u'tree2-1-1', u'tree2-1-1', 0, 0, u'stratum2-1', datetime.date(1, 2, 2),
13.0, None)
(u'tree2-2-1', u'tree2-2-1', 0, 0, u'stratum2-2', datetime.date(1, 2, 2),
14.0, None)
(u'tree2-2-2', u'tree2-2-2', 0, 0, u'stratum2-2', datetime.date(1, 2, 2),
15.0, None)
(u'tree1-1-1', u'tree1-1-1', 0, 1, u'stratum1-1', datetime.date(1, 1, 1),
21.0, None)
(u'tree1-2-1', u'tree1-2-1', 0, 1, u'stratum1-2', datetime.date(1, 1, 1),
22.0, None)
(u'tree1-1-1', u'tree1-1-1', 1, 0, u'stratum1-1', datetime.date(1, 1, 1),
91.0, None)
(u'tree1-2-1', u'tree1-2-1', 1, 0, u'stratum1-2', datetime.date(1, 1, 1),
92.0, None)
(u'tree2-1-1', u'tree2-1-1', 1, 0, u'stratum2-1', datetime.date(1, 2, 2),
93.0, None)
(u'tree2-2-1', u'tree2-2-1', 1, 0, u'stratum2-2', datetime.date(1, 2, 2),
94.0, None)
(u'tree2-2-2', u'tree2-2-2', 1, 0, u'stratum2-2', datetime.date(1, 2, 2),
95.0, None)

>>> sql = 'SELECT * FROM sample_plot ORDER BY iteration, branch, id'
>>> iter = test.conn.execute(sql)
>>> for item in iter:
...     print item # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
(u'plot1', u'plot1', 0, 0, u'stand1', datetime.date(1, 1, 1),
21.0, None)
(u'plot1', u'plot1', 0, 1, u'stand1', datetime.date(1, 1, 1),
31.0, None)
(u'plot1', u'plot1', 1, 0, u'stand1', datetime.date(1, 1, 1),
91.0, None)

>>> sql = 'SELECT * FROM sample_tree ORDER BY iteration, branch, id'
```

```

>>> iter = test.conn.execute(sql)
>>> for item in iter:
...     print item # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
(u'sample_tree1-1', u'sample_tree1-1', 0, 0, u'plot1',
datetime.date(1, 1, 1), 31.0)
(u'sample_tree1-2', u'sample_tree1-2', 0, 0, u'plot1',
datetime.date(1, 1, 1), 32.0)
(u'sample_tree1-1', u'sample_tree1-1', 0, 1, u'plot1',
datetime.date(1, 1, 1), 41.0)
(u'sample_tree1-2', u'sample_tree1-2', 0, 1, u'plot1',
datetime.date(1, 1, 1), 42.0)
(u'sample_tree1-1', u'sample_tree1-1', 1, 0, u'plot1',
datetime.date(1, 1, 1), 91.0)
(u'sample_tree1-2', u'sample_tree1-2', 1, 0, u'plot1',
datetime.date(1, 1, 1), 92.0)

>>> sql = 'SELECT * FROM sample_estate ORDER BY iteration, branch, id'
>>> iter = test.conn.execute(sql)
>>> for item in iter:
...     print item # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
(u'sample_estate1', u'sample_estate1', 0, 0, u'simulation', datetime.date(1, 1, 1), None)
(u'sample_estate1', u'sample_estate1', 0, 0, u'simulation', datetime.date(1, 2, 2), None)
(u'sample_estate1', u'sample_estate1', 0, 1, u'simulation', datetime.date(1, 1, 1), None)
(u'sample_estate1', u'sample_estate1', 0, 1, u'simulation', datetime.date(1, 2, 2), None)
(u'sample_estate1', u'sample_estate1', 1, 0, u'simulation', datetime.date(1, 1, 1), None)
(u'sample_estate1', u'sample_estate1', 1, 0, u'simulation', datetime.date(1, 2, 2), None)

>>> sql = 'SELECT * FROM sample_esplot ORDER BY iteration, branch, id'
>>> iter = test.conn.execute(sql)
>>> for item in iter:
...     print item # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
(u'sample_esplot1-1', u'sample_esplot1-1', 0, 0, u'sample_estate', datetime.date(1, 1, 1), 27.0, 1)
(u'sample_esplot1-1', u'sample_esplot1-1', 0, 0, u'sample_estate', datetime.date(1, 2, 2), 27.0, 1)
(u'sample_esplot1-1', u'sample_esplot1-1', 0, 1, u'sample_estate', datetime.date(1, 1, 1), 37.0, 1)
(u'sample_esplot1-1', u'sample_esplot1-1', 0, 1, u'sample_estate', datetime.date(1, 2, 2), 37.0, 1)
(u'sample_esplot1-1', u'sample_esplot1-1', 1, 0, u'sample_estate', datetime.date(1, 1, 1), 97.0, 1)
(u'sample_esplot1-1', u'sample_esplot1-1', 1, 0, u'sample_estate', datetime.date(1, 2, 2), 97.0, 1)

```

Updating causes the levels between the root simulation level and base level to be updated to new dates as well, text and geometry attributes survive the update between root and base levels, below base level not:

```

>>> from datetime import date
>>> sql = 'UPDATE comp_unit SET StandLabel="my text" WHERE id="stand1"'
>>> c = test.conn.execute(sql)
>>> #TODO: Why doesn't nosetests work with these queries?
>>> #sql = 'UPDATE comp_unit SET geom=GeomFromText(?, ?) WHERE id=?'
>>> #data = ('POLYGON((1,1),(2,2),(3,1),(1.1))', 3067, 'stand1')
>>> #c = test.conn.execute(sql, data)
>>> #sql = 'UPDATE tree SET geom=GeomFromText(?, ?) WHERE id=?'
>>> #data = ('POINT(1,1)', 3067, 'tree1-1-1')
>>> #c = test.conn.execute(sql, data)
>>> test.add_data_from_matrix(mock_db.up_dates, mock_db.datamatrix,
...                           mock_db.Ind2Id(), mock_db.links, [], mock_db.base_level,
...                           update=True)

>>> sql = 'SELECT * FROM simulation ORDER BY iteration, branch, id'
>>> iter = test.conn.execute(sql)
>>> for item in iter:
...     print item # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
(u'sim1', u'sim1', 0, 0, None, datetime.date(1, 1, 1), 1.0)
(u'sim1', u'sim1', 0, 0, None, datetime.date(1, 2, 2), 1.0)
(u'sim1', u'sim1', 0, 1, None, datetime.date(1, 1, 1), 1.0)

```

```
(u'sim1', u'sim1', 0, 1, None, datetime.date(1, 2, 2), 1.0)
(u'sim1', u'sim1', 1, 0, None, datetime.date(1, 1, 1), 1.0)
(u'sim1', u'sim1', 1, 0, None, datetime.date(1, 2, 2), 1.0)

>>> sql = 'SELECT * FROM estate ORDER BY iteration, branch, id'
>>> iter = test.conn.execute(sql)
>>> for item in iter:
...     print item # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
(u'estatel', u'estatel', 0, 0, u'sim1', datetime.date(2009, 1, 1), None)
(u'estatel', u'estatel', 0, 0, u'sim1', datetime.date(2009, 2, 2), None)
(u'estatel', u'estatel', 0, 1, u'sim1', datetime.date(2009, 1, 1), None)
(u'estatel', u'estatel', 0, 1, u'sim1', datetime.date(2009, 2, 2), None)
(u'estatel', u'estatel', 1, 0, u'sim1', datetime.date(2009, 1, 1), None)
(u'estatel', u'estatel', 1, 0, u'sim1', datetime.date(2009, 2, 2), None)

>>> sql = 'SELECT * FROM comp_unit ORDER BY iteration, branch, id'
>>> iter = test.conn.execute(sql)
>>> for item in iter:
...     print item # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
(u'stand1', u'stand1', 0, 0, u'estatel', datetime.date(2009, 1, 1),
10.0, 1.0, u'my text', None)
(u'stand2', u'stand2', 0, 0, u'estatel', datetime.date(2009, 2, 2),
11.0, 1.0, None, None)
(u'stand1', u'stand1', 0, 1, u'estatel', datetime.date(2009, 1, 1),
20.0, 2.0, u'my text', None)
(u'stand2', u'stand2', 0, 1, u'estatel', datetime.date(2009, 2, 2),
21.0, 2.0, None, None)
(u'stand1', u'stand1', 1, 0, u'estatel', datetime.date(2009, 1, 1),
90.0, 9.0, u'my text', None)
(u'stand2', u'stand2', 1, 0, u'estatel', datetime.date(2009, 2, 2),
91.0, 9.0, None, None)

>>> sql = 'SELECT id, iteration, branch, date, geom FROM tree '\
...       'ORDER BY iteration, branch, id'
>>> iter = test.conn.execute(sql)
>>> for item in iter:
...     print item # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
(u'tree1-1-1', 0, 0, datetime.date(2009, 1, 1), None)
(u'tree1-2-1', 0, 0, datetime.date(2009, 1, 1), None)
(u'tree2-1-1', 0, 0, datetime.date(2009, 2, 2), None)
(u'tree2-2-1', 0, 0, datetime.date(2009, 2, 2), None)
(u'tree2-2-2', 0, 0, datetime.date(2009, 2, 2), None)
(u'tree1-1-1', 0, 1, datetime.date(2009, 1, 1), None)
(u'tree1-2-1', 0, 1, datetime.date(2009, 1, 1), None)
(u'tree1-1-1', 1, 0, datetime.date(2009, 1, 1), None)
(u'tree1-2-1', 1, 0, datetime.date(2009, 1, 1), None)
(u'tree2-1-1', 1, 0, datetime.date(2009, 2, 2), None)
(u'tree2-2-1', 1, 0, datetime.date(2009, 2, 2), None)
(u'tree2-2-2', 1, 0, datetime.date(2009, 2, 2), None)

>>> sql = 'SELECT id, iteration, branch, date FROM sample_tree '\
...       'ORDER BY iteration, branch, id'
>>> iter = test.conn.execute(sql)
>>> for item in iter:
...     print item # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
(u'sample_tree1-1', 0, 0, datetime.date(2009, 1, 1))
(u'sample_tree1-2', 0, 0, datetime.date(2009, 1, 1))
(u'sample_tree1-1', 0, 1, datetime.date(2009, 1, 1))
(u'sample_tree1-2', 0, 1, datetime.date(2009, 1, 1))
(u'sample_tree1-1', 1, 0, datetime.date(2009, 1, 1))
(u'sample_tree1-2', 1, 0, datetime.date(2009, 1, 1))
```

```

>>> sql = 'SELECT * FROM sample_estate ORDER BY iteration, branch, id'
>>> iter = test.conn.execute(sql)
>>> for item in iter:
...     print item # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
(u'sample_estate1', u'sample_estate1', 0, 0, u'simulation', datetime.date(2009, 1, 1), None)
(u'sample_estate1', u'sample_estate1', 0, 0, u'simulation', datetime.date(2009, 2, 2), None)
(u'sample_estate1', u'sample_estate1', 0, 1, u'simulation', datetime.date(2009, 1, 1), None)
(u'sample_estate1', u'sample_estate1', 0, 1, u'simulation', datetime.date(2009, 2, 2), None)
(u'sample_estate1', u'sample_estate1', 1, 0, u'simulation', datetime.date(2009, 1, 1), None)
(u'sample_estate1', u'sample_estate1', 1, 0, u'simulation', datetime.date(2009, 2, 2), None)

>>> sql = 'SELECT * FROM sample_esplot ORDER BY iteration, branch, id'
>>> iter = test.conn.execute(sql)
>>> for item in iter:
...     print item # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
(u'sample_esplot1-1', u'sample_esplot1-1', 0, 0, u'sample_estate', datetime.date(2009, 1, 1), 27.0)
(u'sample_esplot1-1', u'sample_esplot1-1', 0, 0, u'sample_estate', datetime.date(2009, 2, 2), 27.0)
(u'sample_esplot1-1', u'sample_esplot1-1', 0, 1, u'sample_estate', datetime.date(2009, 1, 1), 37.0)
(u'sample_esplot1-1', u'sample_esplot1-1', 0, 1, u'sample_estate', datetime.date(2009, 2, 2), 37.0)
(u'sample_esplot1-1', u'sample_esplot1-1', 1, 0, u'sample_estate', datetime.date(2009, 1, 1), 97.0)
(u'sample_esplot1-1', u'sample_esplot1-1', 1, 0, u'sample_estate', datetime.date(2009, 2, 2), 97.0)

```

### 32.2.7 def add\_branching\_info(self, level, binfo, ind2id, start\_iter=0):

Adds a new item to the BRANCHING\_INFO table. binfo is a dictionary consisting of key and data tuples as (iteration index, child branch index, object index):(parent branch index, branching operation group, operation, branch name, date)

```

>>> bdate = date(2009, 7, 8)
>>> test.add_branching_info(2, {(0,1,0):(0,'tgrp','top', 'name', bdate)},
...                          mock_db.ind2id)
>>> try:
...     test.add_branching_info(2,
...                             {(0,1,0):(0,'tgrp','top', 'name', bdate)}, mock_db.ind2id)
... except RuntimeError, e:
...     print e # doctest: +NORMALIZE_WHITESPACE
simo.db.datadb.db, error, An error occurred with branching: columns object,
iteration, from_branch, to_branch, date are not unique
>>> iter = test.conn.execute('SELECT * FROM BRANCHING_INFO')
>>> for row in iter: print row
(u'stand1', 0, 0, 1, datetime.date(2009, 7, 8), u'tgrp', u'top', u'name')

```

### 32.2.8 def add\_opres(self, levelname, iteration, data):

This adds new rows to the database. data is a list of dictionaries, each containing date, id, branch, values, op\_name and op\_group. values is a list of dictionaries containing var-val pairs

```

>>> ddate = date(2009, 2, 2)
>>> data = [{'date': ddate,
...         'op_id': 1,
...         'obj_id': 'stand1',
...         'branch': 0,
...         'values': [{'cash_flow': 3.0},
...                    {'Volume': 2.0, 'SP': 1., 'assortment': 2.},
...                    {'Volume': 6.0, 'SP': 1., 'assortment': 1.}],
...         'op_name': 'thinning',
...         'op_group': 'cutting'}]
>>> test.add_opres('comp_unit', 0, data)

```

```
>>> iter = test.conn.execute('SELECT * FROM op_res').fetchall()
>>> for row in iter:
...     print row # doctest: +NORMALIZE_WHITESPACE
(1, 1, u'stand1', u'comp_unit', 0, 0, 3.0, None, None, None,
 u'thinning', u'cutting', datetime.date(2009, 2, 2))
(2, 1, u'stand1', u'comp_unit', 0, 0, None, 1.0, 2.0, 2.0,
 u'thinning', u'cutting', datetime.date(2009, 2, 2))
(3, 1, u'stand1', u'comp_unit', 0, 0, None, 1.0, 1.0, 6.0,
 u'thinning', u'cutting', datetime.date(2009, 2, 2))
```

### 32.2.9 def fill\_dictionary(self, level, oid):

Returns a dictionary in the format that `add_data_from_dictionary` uses, containing all the iterations, branches and dates of the given level and oid:

```
>>> res = test.fill_dictionary('comp_unit', 'stand2')
>>> for item in res['comp_unit']:
...     ddict = item[1]
...     print item[0], ddict['iteration'], ddict['branch'], ddict['values']
...     #doctest: +NORMALIZE_WHITESPACE
2009-02-02 0 1 [(u'BA', 21.0), (u'SC', 2.0)]
2009-02-02 1 0 [(u'BA', 91.0), (u'SC', 9.0)]
2009-02-02 0 0 [(u'BA', 11.0), (u'SC', 1.0)]
```

### 32.2.10 def fill\_matrix(self, handler, lind, ids, iter=0, branch=0):

Fills a data matrix with data for the objects identified by the level and a set of object ids. The matrix is filled for all data levels and objects linked to from the given level and ids in top down order; i.e., simulation level first. All lineages are filled as well:

```
>>> test.fill_matrix(mock_db.handler, 2, ('stand1', 'stand2'))
...     # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
>>> pprint(mock_db.handler.added_ids)
[[u'sim1'],
 [u'estate1'],
 ['stand1'],
 [u'stratum1-1', u'stratum1-2'],
 [u'tree1-1-1'],
 [u'tree1-2-1'],
 [u'plot1'],
 [u'sample_tree1-1', u'sample_tree1-2'],
 ['stand2'],
 [u'stratum2-1', u'stratum2-2'],
 [u'tree2-1-1'],
 [u'tree2-2-1', u'tree2-2-2']]
>>> pprint(mock_db.handler.dmx)
[(array([[0, 0, 0]]), datetime.date(2009, 1, 1)),
 (array([[0, 0, 0]]), datetime.date(2009, 2, 2))]
>>> pprint(mock_db.handler.mx) # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
[(0, array([0]), [0], array([[ 1.]])),
 (2, array([0]), [0, 1], array([[ 10.], [ 1.]])),
 (3, array([0, 1]), [0, 1], array([[ 100., 110.], [ 1., 1.]])),
 (4, array([0]), [0], array([[ 11.]])),
 (4, array([0]), [0], array([[ 12.]])),
 (5, array([0]), [0], array([[ 21.]])),
 (6, array([0, 1]), [0], array([[ 31., 32.]])),
 (2, array([0]), [0, 1], array([[ 11.], [ 1.]])),
 (3, array([0, 1]), [0, 1], array([[ 120., 130.], [ 1., 1.]])),
```



```
(4, array([0]), [0], array([[ 13.]])),
(4, array([0, 1]), [0], array([[ 14., 15.]])])
```

### 32.2.11 def get\_level\_var\_headers(self, level):

Will return the variable headers of the table of the given level

```
>>> test.get_level_var_headers(0)
['DIAM_CLASS_WIDTH']
>>> test.get_level_var_headers(1)
['EstateName']
>>> test.get_level_var_headers(3)
['N', 'SP']
```

```
def get_data_from_level(self, from_data, headers, level, constraints={}, dates=None, rsort=None, required=None): =====
```

Will return an iterator for the data (as requested by headers) from the given level with the given constraints based on the given dictionary ({col: [op, val(, val)]}, op one of ('gt', 'ge', 'eq', 'ue', 'le', 'lt', 'in') and multiple vals only for 'in') dates is a tuple with the start and end date from between which the data is retrieved. rsort is the column by which the returned data should be sorted and required is a list of columns that must be non-null for the rows to be included in the results.

```
>>> headers = test.get_level_var_headers(4)
>>> headers = ['id', 'iteration', 'branch'] + headers
>>> constraints = [('id', 'in', ('tree1-1-1', 'tree1-2-1'), 'and'),
...               ('branch', 'in', (0, 1))]
>>> iter = test.get_data_from_level('data', headers, 4, constraints)
>>> for item in iter:
...     print item
(u'tree1-1-1', 1, 0, 91.0)
(u'tree1-2-1', 1, 0, 92.0)
(u'tree1-1-1', 0, 0, 11.0)
(u'tree1-2-1', 0, 0, 12.0)
(u'tree1-1-1', 0, 1, 21.0)
(u'tree1-2-1', 0, 1, 22.0)
>>> headers = ['object_id', 'branch', 'op_name', 'cash_flow', 'Volume',
...           'assortment', 'date']
>>> iter = test.get_data_from_level('op_res', headers, 0)
>>> for item in iter:
...     print item # doctest: +NORMALIZE_WHITESPACE
(u'stand1', 0, u'thinning', 3.0, None, None, datetime.date(2009, 2, 2))
(u'stand1', 0, u'thinning', None, 2.0, 2.0, datetime.date(2009, 2, 2))
(u'stand1', 0, u'thinning', None, 6.0, 1.0, datetime.date(2009, 2, 2))
>>> iter = test.get_data_from_level('op_res', headers, 0,
...                               required=['Volume', 'object_id'])
>>> for item in iter:
...     print item # doctest: +NORMALIZE_WHITESPACE
(u'stand1', 0, u'thinning', None, 2.0, 2.0, datetime.date(2009, 2, 2))
(u'stand1', 0, u'thinning', None, 6.0, 1.0, datetime.date(2009, 2, 2))
```

### 32.2.12 def get\_ids(self, level):

Will return all ids from the level

```
>>> idlist = test.get_ids('data', 2)
>>> set(idlist)==set([u'stand1', u'stand2'])
True
```

```
>>> idset = set(test.get_ids('data', 3))
>>> resset = set([u'stratum1-1', u'stratum1-2', u'stratum2-1',
...              u'stratum2-2'])
>>> resset==idset
True
```

### 32.2.13 def get\_max\_iter(self, from\_data, level):

Will return the max iteration from the level

```
>>> test.get_max_iter('data', 1)
1
>>> test.get_max_iter('data', 2)
1
```

### 32.2.14 def get\_max\_branch(self, from\_data, level):

Will return the max branch from the level

```
>>> test.get_max_branch('data', 1)
1
>>> test.get_max_branch('data', 2)
1
```

### 32.2.15 def get\_dates(self, from\_data, level, dates):

Will return the unique dates on a given level, between the given tuple of dates (lower limit, higher limit) ::

```
>>> date_lim = (date(2000, 1, 1), date(2010, 1, 1))
>>> dset = set(test.get_dates('data', 2, date_lim))
>>> dset==set([date(2009, 1, 1), date(2009, 2, 2)])
True
>>> test.get_dates('data', 3, (date(2009, 2, 2), date(2010, 1, 1)))
[datetime.date(2009, 2, 2)]
```

### 32.2.16 def get\_dates\_by\_id\_and\_iter(self, level, dates):

Will return a dictionary with (id, iter) pairs as keys and unique dates for a given id as values

```
>>> rd = test.get_dates_by_id_and_iter(2, (date(2000, 1, 1),
...                                       date(2010, 1, 1)))
>>> rd[(u'stand1', 0)]
[datetime.date(2009, 1, 1)]
>>> rd[(u'stand1', 1)]
[datetime.date(2009, 1, 1)]
>>> rd[(u'stand2', 1)]
[datetime.date(2009, 2, 2)]
>>> rd[(u'stand2', 0)]
[datetime.date(2009, 2, 2)]
>>> rd = test.get_dates_by_id_and_iter(3, (date(2009, 2, 2),
...                                       date(2010, 1, 1)))
>>> rd[(u'stratum2-1', 1)]
[datetime.date(2009, 2, 2)]
>>> rd[(u'stratum2-1', 0)]
```

```
[datetime.date(2009, 2, 2)]
>>> rd[(u'stratum2-2', 0)]
[datetime.date(2009, 2, 2)]
>>> rd[(u'stratum2-2', 1)]
[datetime.date(2009, 2, 2)]
```

### 32.2.17 def get\_all\_relative\_ids(self, ids, main\_level):

Will return all relative ids to the list of ids from the given level

```
>>> ids = ('stratum1-2', 'stratum1-1')
>>> res = test.get_all_relative_ids(ids, 'stratum')
>>> res.sort()
>>> res # doctest: +NORMALIZE_WHITESPACE
[u'estatel1', u'plot1', u'sample_treel-1', u'sample_treel-2', u'sim1',
 u'stand1', u'treel-1-1', u'treel-2-1']
```

### 32.2.18 def copy\_to\_db(self, key\_list, lname, db):

For the given keys of (iteration, object id, branch) copies the data to a new db from the given level. Level is taken as base level, below which all the children from different levels are copied as well:

```
>>> keys = [(0, 'stand1', 0), # iter, id, branch
...         (0, 'stand2', 1),
...         (1, 'stand1', 0),
...         (1, 'stand2', 0)]
>>> relatives1 = test.get_all_relative_ids(('stand1',), 'comp_unit')
>>> copydb = db.DataDB('write', content_def, hierarchy, levels, opres_def,
...                   cf_cfiers, mock_db.MockLogger(), 'test_copy.db',
...                   True, False, True)
>>> test.copy_to_db(keys, 'comp_unit', copydb)
>>> relatives2 = copydb.get_all_relative_ids(('stand1',), 'comp_unit')
>>> set(relatives1)==set(relatives2)
True
>>> headers = ['object_id', 'iteration', 'branch', 'op_name', 'op_group',
...            'cash_flow', 'Volume', 'assortment', 'date']
>>> opres1 = test.get_data_from_level('op_res', headers, 0)
>>> opres2 = copydb.get_data_from_level('op_res', headers, 0)
>>> opres1 = [i for i in opres1]
>>> opres1.sort()
>>> opres2 = [i for i in opres2]
>>> opres2.sort()
>>> opres1==opres2
True
```

### 32.2.19 def \_create\_constraints(self, constraints):

Will return a SQLite string of constraints based on the given dictionary ({col: [val, val, val]})

```
>>> const = [('id', 'in', ('treel-1-1', 'treel-2-1'), 'and'),
...          ('branch', 'in', (0, 1), 'or'), ('iteration', 'eq', 0)]
>>> test._create_constraints(const) #doctest: +NORMALIZE_WHITESPACE
('"id" IN (?,?) AND "branch" IN (?,?) OR "iteration" == ? ',
 ['treel-1-1', 'treel-2-1', 0, 1, 0])
>>> const = [('id', 'fail', 'this', 'and'),
...          ('test', 'eq', ('fail', 'fail')),
...          ('should not come here',)]
```

```
>>> test._create_constraints(const) #doctest: +NORMALIZE_WHITESPACE
db, error, Bad operation (fail) not in (le, lt, ge, gt, in, ue, eq)
db, error, missing concatenation operand from [('id', 'fail', 'this', 'and'),
('test', 'eq', ('fail', 'fail')), ('should not come here',)] index 1
('', [])
>>> const = [('arr', 'eq', ('fail', 'fail'), 'and'), ('rar', 'fail', 1)]
>>> test._create_constraints(const) #doctest: +NORMALIZE_WHITESPACE
db, error, Too many values for operation type (eq)
db, error, Bad operation (fail) not in (le, lt, ge, gt, in, ue, eq)
('', [])
```

### 32.2.20 def add\_data\_from\_dictionary(self, data, iter, branch):

Adds new data from a directory {levelname: [(ddate, {'id':id, 'oid': original\_id, 'parent id': pid, 'values': [(('key', value), ('key', value))], ...], ...}]:

```
>>> ddate = date(2009, 2, 7)
>>> data = {'comp_unit': [(ddate, {'id':'stand1', 'oid':'oid',
...                               'parent id': 'estate1',
...                               'values': [(('SC', 1.), ('BA', 10.))]}),
...         (ddate, {'id':'stand2', 'oid':'oid', 'parent id': 'sim1',
...                               'values': [(('SC', 2.), ('BA', 20.))]})]}
>>> test.add_data_from_dictionary(data, 0, 0)
>>> sql = "SELECT id, oid, iteration, branch, pid, BA, SC FROM\"
...       " comp_unit WHERE date=?"
>>> res = test.conn.execute(sql, (ddate,)).fetchall()
>>> for row in res:
...     print row
(u'stand1', u'oid', 0, 0, u'estate1', 10.0, 1.0)
(u'stand2', u'oid', 0, 0, u'sim1', 20.0, 2.0)
```

### 32.2.21 def get\_child\_ids(self, lind, oid, iter=0, branch=0, ddate=None):

Recursively gets child ids for the given object and optionally for the given date:

```
>>> test.get_child_ids(2, 'stand1') # doctest: +NORMALIZE_WHITESPACE
defaultdict(<type 'list'>, {3: [u'stratum1-1', u'stratum1-2'], 4:
[u'tree1-1-1', u'tree1-2-1'], 5: [u'plot1'], 6: [u'sample_tree1-1',
u'sample_tree1-2']})
```

### 32.2.22 def clear\_data():

Clears data from level tables:

```
>>> test.clear_data()
>>> keys = test.content_def.keys()
>>> keys.sort()
>>> for cdef_key in keys:
...     level_name = cdef_key[0]
...     sql = 'SELECT * FROM %s' % level_name
...     res = test.conn.execute(sql).fetchall()
...     print level_name, len(res)
comp_unit 0
estate 0
sample_esplot 0
sample_estate 0
sample_plot 0
```

```

sample_tree 0
simulation 0
stratum 0
tree 0
>>> test.close()

```

## 32.3 class OperationDB(SQLiteDB):

### 32.3.1 def \_\_init\_\_(self, content\_def, db\_path='default.db', wipe=False):

```

>>> opdb = db.OperationDB('test_mock.db', True)
>>> opdb.close()

```

The init will create/open a connection to the given database file (wiping the old one if necessary) and will then create the tables it needs.

### 32.3.2 def \_create\_tables(self):

Creates the tables that OperationDB needs.

```

>>> opdb.open()
>>> sql = "select sql from sqlite_master where type = 'table'"
>>> iter = opdb.conn.execute(sql)
>>> for item in iter:
...     print item[0] # doctest: +NORMALIZE_WHITESPACE
CREATE TABLE forced_operation
    (op_id INTEGER NOT NULL,
    unit_id TEXT NOT NULL,
    level TEXT NOT NULL,
    iteration INTEGER NOT NULL,
    branch INTEGER NOT NULL,
    name TEXT NOT NULL,
    timingtype TEXT NOT NULL,
    timestep INTEGER,
    date DATE,
    stepunit TEXT,
    CONSTRAINT pk_forced_operation PRIMARY KEY (op_id, unit_id))
CREATE TABLE operation_chain
    (id INTEGER PRIMARY KEY,
    op_id INTEGER NOT NULL,
    pid TEXT NOT NULL,
    chain_ordinal INTEGER NOT NULL,
    chain TEXT NOT NULL,
    CONSTRAINT fk_pid FOREIGN KEY (op_id, pid) REFERENCES
    forced_operation (op_id, unit_id))

```

```
def add_operation(unit_id, level, iteration, branch, name, timingtype, timestep, date, stepunit, chains):
```

```
=====
```

This adds new rows to the database.

```

>>> odate = date(2009, 3, 9)
>>> opdb.add_operation('stand1', 'comp_unit', 0, 0, 'mounding', 'step',
...                   2, None, 'year', ['Forced mounding'])
>>> opdb.add_operation('stand2', 'comp_unit', 0, 0, 'clearcut', 'step', 2,
...                   None, 'year', ['Forced clearcut',
...                   'Update comp_unit after forced clearcut or strip cut',

```

```
...         'Calculate productive value'])
>>> opdb.add_operation('stand3', 'comp_unit', 0,0,'clearcut', 'date', None,
...     odate, 'year', ['Forced clearcut',
...         'Update comp_unit after forced clearcut or strip cut',
...         'Calculate productive value'])
>>> iter = opdb.conn.execute('select * from forced_operation')
>>> expected_results = [
...     [1, u'stand1', u'comp_unit', 0, 0, u'mounding', u'step', 2, None,
...         u'year'],
...     [1, u'stand2', u'comp_unit', 0, 0, u'clearcut', u'step', 2, None,
...         u'year'],
...     [1, u'stand3', u'comp_unit', 0, 0, u'clearcut', u'date', None,
...         odate, u'year']]
>>> i = 0
>>> for row in iter:
...     res = expected_results[i]
...     i+=1
...     for val, exp in zip(row, res):
...         if type(val) != type(exp):
...             print 'Type mismatch:', type(val), type(exp)
...         if val != exp:
...             print 'Value mismatch: got', val, 'expected', exp
...
>>> iter = opdb.conn.execute('select * from operation_chain')
>>> expected_results = [
...     [1, 1, u'stand1', 0, u'Forced mounding'],
...     [2, 1, u'stand2', 0, u'Forced clearcut'],
...     [3, 1, u'stand2', 1, u'Update comp_unit after forced clearcut or strip cut'],
...     [4, 1, u'stand2', 2, u'Calculate productive value'],
...     [5, 1, u'stand3', 0, u'Forced clearcut'],
...     [6, 1, u'stand3', 1, u'Update comp_unit after forced clearcut or strip cut'],
...     [7, 1, u'stand3', 2, u'Calculate productive value']]
>>> i = 0
>>> for row in iter:
...     res = expected_results[i]
...     i+=1
...     for val, exp in zip(row, res):
...         if type(val) != type(exp):
...             print 'Type mismatch:', type(val), type(exp)
...         if val != exp:
...             print 'Value mismatch: got', val, 'expected', exp
```

### 32.3.3 def fill\_dictionary(self, levelname, ids):

Fills a dictionary with object ids as keys (ids) and ForcedOperation instances as data

```
>>> fcd_op_dict = opdb.fill_dictionary('comp_unit', ['stand1', 'stand2'])
>>> for key, vals in fcd_op_dict.items(): # doctest: +NORMALIZE_WHITESPACE
...     for val in vals:
...         print key, '-', val.id, val.level, val.timing_type, \
...             val.time_step,
...         print val.date, val.step_unit,
...         print val.chain_names,
...         print val.chain_indices
(0, 0, u'stand1') -
stand1 comp_unit step 2 None year
[u'Forced mounding']
None
(0, 0, u'stand2') -
stand2 comp_unit step 2 None year
```

```
[u'Forced clearcut',
 u'Update comp_unit after forced clearcut or strip cut',
 u'Calculate productive value']
None
```

### 32.3.4 def clear\_data():

Clears data from data tables:

```
>>> opdb.clear_data()
>>> sql = 'SELECT * FROM forced_operation'
>>> res = opdb.conn.execute(sql).fetchall()
>>> res
[]
>>> sql = 'SELECT * FROM operation_chain'
>>> res = opdb.conn.execute(sql).fetchall()
>>> res
[]
>>> opdb.close()
>>> opdb = None
```

## 32.4 class LoggerDB(SQLiteDB):

### 32.4.1 def \_\_init\_\_(self, db\_path='default.db', wipe=False):

Connects to the given database file (wiping the existing one if requested) and runs `_create_log_table()`:

```
>>> test = db.LoggerDB('test_mock.db', True)
```

### 32.4.2 def \_create\_log\_table(self):

Creates the log table into the database:

```
>>> sql = "select sql from sqlite_master where type='table' and "\
...       "name='log'"
>>> iter = test.conn.execute(sql)
>>> for item in iter:
...     print item[0]
CREATE TABLE log(
    run_id TEXT,
    level TEXT,
    module TEXT,
    time TIMESTAMP,
    message TEXT)
```

### 32.4.3 def add\_log\_message(self, level, module, timestamp, message):

This will add a new log message into the database:

```
>>> from datetime import datetime
>>> date = datetime(2009, 2, 2, 15, 17, 56, 626329)
>>> date2 = datetime(2010, 4, 9, 10, 25)
>>> test.add_log_message('test', 'one', 'two', date, 'Ni!')
```

```
>>> test.add_log_message('test2', 'three', 'four', date2, 'Shrubbery!')
>>> iter = test.conn.execute('select * from log')
>>> for item in iter:
...     print item # doctest: +NORMALIZE_WHITESPACE
(u'test', u'one', u'two',
 datetime.datetime(2009, 2, 2, 15, 17, 56, 626329), u'Ni!')
(u'test2', u'three', u'four',
 datetime.datetime(2010, 4, 9, 10, 25), u'Shrubbery!')
```

#### 32.4.4 def get\_log\_messages(self, level=None, module=None, start\_time=None, end\_time=None):

This method retrieves log data with the given constraints. level and module are tuples containing wanted levels and modules respectively. start\_time and end\_time are timestamps (datetime instances) to constrain the output.

```
>>> pprint(test.get_log_messages())
[(datetime.datetime(2009, 2, 2, 15, 17, 56, 626329),
  u'two',
  u'one',
  u'test',
  u'Ni!'),
 (datetime.datetime(2010, 4, 9, 10, 25),
  u'four',
  u'three',
  u'test2',
  u'Shrubbery!')]

>>> pprint(test.get_log_messages(level=('one', 'six', 'ten')))
[(datetime.datetime(2009, 2, 2, 15, 17, 56, 626329),
  u'two',
  u'one',
  u'test',
  u'Ni!')]

>>> pprint(test.get_log_messages(module=('four', 'three', 'seven')))
[(datetime.datetime(2010, 4, 9, 10, 25),
  u'four',
  u'three',
  u'test2',
  u'Shrubbery!')]

>>> pprint(test.get_log_messages(run_id=('test2', 'tset3', 'nonexistent')))
[(datetime.datetime(2010, 4, 9, 10, 25),
  u'four',
  u'three',
  u'test2',
  u'Shrubbery!')]

>>> pprint(test.get_log_messages(start_time=date2))
[(datetime.datetime(2010, 4, 9, 10, 25),
  u'four',
  u'three',
  u'test2',
  u'Shrubbery!')]

>>> pprint(test.get_log_messages(end_time=date))
[(datetime.datetime(2009, 2, 2, 15, 17, 56, 626329),
  u'two',
  u'one',
  u'test',
```



```

    u'Ni!'])

>>> pprint(test.get_log_messages(start_time=date, end_time=date2))
[(datetime.datetime(2009, 2, 2, 15, 17, 56, 626329),
  u'two',
  u'one',
  u'test',
  u'Ni!'),
 (datetime.datetime(2010, 4, 9, 10, 25),
  u'four',
  u'three',
  u'test2',
  u'Shrubbery!')]

>>> pprint(test.get_log_messages(run_id=('test', 'test2'),
...                               level=('one', 'three'),
...                               module=('two', 'four'),
...                               start_time=date, end_time=date2))
[(datetime.datetime(2009, 2, 2, 15, 17, 56, 626329),
  u'two',
  u'one',
  u'test',
  u'Ni!'),
 (datetime.datetime(2010, 4, 9, 10, 25),
  u'four',
  u'three',
  u'test2',
  u'Shrubbery!')]

>>> test.get_log_messages(level=('fail'))
[]
>>> test.get_log_messages(module=('fail'))
[]
>>> test.get_log_messages(start_time=datetime(1800, 1, 1),
...                       end_time=datetime(1805, 1, 1))
[]

>>> test.get_log_messages(run_id=('test',), empty_log=True)
>>> pprint(test.get_log_messages())
[(datetime.datetime(2010, 4, 9, 10, 25),
  u'four',
  u'three',
  u'test2',
  u'Shrubbery!')]

>>> test.close()
>>> test = None

```



## DB.PY

```
>>> from simo.db.simodb.test.test_classes import *
>>> from simo.builder import names
>>> from minimock import Mock
>>> tdf = open('../../simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../../simulator/xml/schemas/operation2modelchains.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = u'''<SIMO_operation_mapping xmlns="http://www.simo-project.org/simo"
... xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
... xsi:schemaLocation="http://www.simo-project.org/simo
...                      ./schemas/operation_mapping.xsd">
... <operation>
...   <from>
...     <name>thinning</name>
...   </from>
...   <to>
...     <model_chain>Calculate thinning limits</model_chain>
...     <model_chain>Forced low thinning</model_chain>
...   </to>
... </operation>
... <operation>
...   <from>
...     <name>first_thinning</name>
...   </from>
...   <to>
...     <model_chain>Calculate thinning limits</model_chain>
...     <model_chain>Forced first thinning</model_chain>
...   </to>
... </operation>
... <operation>
...   <from>
...     <name>clearcut</name>
...   </from>
...   <to>
...     <model_chain>Forced clearcut</model_chain>
...   </to>
... </operation>
... <operation>
...   <from>
...     <name>strip_cutting</name>
...   </from>
...   <to>
...     <model_chain>Forced stripcut</model_chain>
...     <model_chain>Calculate productive value</model_chain>
...   </to>
... </operation>
```

```
... </SIMO_operation_mapping>'''
>>> sf = open('../..simulator/xml/schemas/aggregation_modelbase.xsd')
>>> mbschema = sf.read()
>>> sf.close()
>>> mbxml = u'''<aggregation_base xmlns="http://www.simo-project.org/simo"
...           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...           <model>
...               <name>sum</name>
...               <description>desc...</description>
...               <implemented_at>simoaggr.py</implemented_at>
...               <vector_implementation>true</vector_implementation>
...           </model>
...       </aggregation_base>'''
>>> mcsf = open('../..simulator/xml/schemas/model_chain.xsd')
>>> mcschema = mcsf.read()
>>> mcsf.close()
>>> mcxml = u'''<model_chains
...     xsi:schemaLocation="http://www.simo-project.org/simo
...     ../..schemas/model_chain.xsd"
...     xmlns="http://www.simo-project.org/simo"
...     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...     <chain_group name="Forced thinnings">
...         <model_chain evaluate_at="comp_unit" name="Calculate thinning limits">
...             <task name="Calculate thinning limits">
...                 <model>
...                     <name>Thinning_limits_JKK</name>
...                     <prediction>
...                         <parameters>
...                             <parameter>
...                                 <name>THINNING_LEVEL</name>
...                                 <value>0.5</value>
...                             </parameter>
...                         </parameters>
...                     </prediction>
...                 </model>
...             </task>
...         </model_chain>
...     </chain_group>
... </model_chains>'''
>>> from simo.builder.modelbase.predictionmodel import PredictionModel
>>> from lxml import etree
>>> xml2 = u'''<model_group name="Distribution models">
...     <model>
...         <name>Productive_value_of_land_pine_Pukkala</name>
...         <implemented_at>StaticstandModels.dll</implemented_at>
...         <implemented_in>C</implemented_in>
...         <author>
...             <name>Timo Pukkala</name>
...         </author>
...         <description>...</description>
...         <published_in>...</published_in>
...         <species_list>
...             <species>1</species>
...         </species_list>
...         <geographical_coverage>...</geographical_coverage>
...         <applies_for>...</applies_for>
...         <research_material>...</research_material>
...         <variables>
...             <variable>
...                 <name>TS</name>
...                 <level>comp_unit</level>
...             </variable>
```

```

...         </variables>
...         <parameters>
...             <parameter>
...                 <name>THINNING_LEVEL</name>
...             </parameter>
...         </parameters>
...         <result>
...             <object>self</object>
...             <variables>
...                 <variable>
...                     <name>PVland_Sp</name>
...                 </variable>
...             </variables>
...         </result>
...     </model>
... </model_group>'''
>>> class Validator:
...     def elem_name(self, text):
...         return text
...     def variable_ind(self, level, variable, active=False):
...         return (1,1)
...     def add_model(self, mname, mtype):
...         pass
...     def add_error(self, text):
...         print 'error:', text
...     errors = []
...     warnings = []
>>> elem = etree.fromstring(xml2)
>>> pr = PredictionModel('', elem[0], Validator(),
...                       '../simulator/models/prediction',
...                       elem.attrib['name'])
>>> pr.param_names
['THINNING_LEVEL']
>>> pr.param_limits
[None]
>>> pr.n_params
1
>>> mimpl = {names.PREDICTION: {'Thinning_limits_JKK': pr}}
```

## 33.1 class SimoDB(object):

### 33.1.1 def \_\_init\_\_(self, file, create\_new=False, read\_only=False):

Initializes the database, creating or opening the necessary files

```

>>> from simo.db.simodb.db import SimoDB
>>> db = SimoDB('db/simodb/test/test.fs', True)
>>> db.dbroot['test'] = 'test'
>>> db.commit()
>>> print db.dbroot['test']
test
>>> db.close()
>>> db = SimoDB('db/simodb/test/test.fs', False)
>>> print db.dbroot['test']
test
>>> try:
...     db2 = SimoDB('db/simodb/test/test.fs', False)
... except Exception, e:
...     print e #Two read-write connections not allowed.
```

```
Couldn't lock 'db/simodb/test/test.fs.lock'
>>> db2 = SimoDB('db/simodb/test/test.fs', False, read_only=True)
>>> db.close()
>>> db2.close()
>>> db2 = None
>>> db = SimoDB('db/simodb/test/test2.fs', True)
```

### 33.1.2 def set\_schema(self, type, schema):

Sets a new type to the db if not present and sets a schema for that type. Returns True if successful, False if not.:

```
>>> try:
...     db.set_schema('operation2modelchains', schema)
... except ValueError, e:
...     print e
Typedef not set
>>> db.typedef = typedef
>>> db.set_schema('operation2modelchains', schema)
>>> db.set_schema('modelbase', mbschema, 'aggregation')
>>> db.set_schema('modelchain', mcschema)
>>> try:
...     db.set_schema('test', mcschema)
... except ValueError, e:
...     print e
unsupported SIMO class type 'test'
```

### 33.1.3 def get\_schema(self, simoclass, atype):

Returns the schema of the given type or None if the type is not available.:

```
>>> db.get_schema('operation2modelchains')==schema
True
>>> db.get_schema('modelbase', 'aggregation')==mbschema
True
>>> db.get_schema('modelchain')==mcschema
True
>>> db.get_schema('Something else')
```

### 33.1.4 def set\_xml(self, simoclass, xml, mtype):

Stores the xml to the database. This triggers object instance creation as well. The xml parameter is in general a tuple of (name, xml text, lexicon object instance). In case of model chain XML the tuple consists of (name, xml text, lexicon object instance, chain type, model implementation). For modelbase XML the tuple consists of: (name, xml text, lexicon object instance, directory path, model type). The directory path refers to where the implementations of models are located.

The method returns a tuple of (top level error, ok, warnings, errors). Raises an ValueError if the schema has not been set.

```
>>> db.set_xml('operation2modelchains', ('testxml', xml, XMLLexicon()))
(None, True, [], set([]))
>>> db.set_xml('modelbase', ('testxml', mbxml, ModelLexicon(), '',
...                           'aggregation'))
(None, True, [], set([]))
>>> db.set_xml('modelchain', ('testxml', mcxml, ChainLexicon(), 'simulation',
...                           mimpl))
(None, True, [], set([]))
```

```
>>> db.set_xml('modelchain', ('testxml2', mcxml, ChainLexicon(), 'simulation',
...                             mimpl))
(None, True, [], set([]))
```

### 33.1.5 def get\_xml(self, simoclass, name):

Returns the xmltext set to the given name in the given type in the database. Returns (None, None) if either the type or the name is not available

```
>>> data = db.get_xml('operation2modelchains', 'testxml')
>>> data[0][:23]
u'<SIMO_operation_mapping'
>>> data[1]
True
>>> data = db.get_xml('modelbase', 'testxml', 'aggregation')
>>> data[0][:17]
u'<aggregation_base'
>>> data[1]
True
>>> data = db.get_xml('modelchain', 'testxml', 'simulation')
>>> data[0][:13]
u'<model_chains'
>>> data[1]
True
>>> db.get_xml('something else', 'testxml')
(None, None)
```

### 33.1.6 def get\_names(self, simoclass, atype):

Returns a list of the names of the xmls that have been saved to the database. Modelchain and modelbase need chain/model type definition as well:

```
>>> db.get_names('operation2modelchains')
['testxml']
>>> db.get_names('modelbase', 'aggregation')
['testxml']
>>> db.get_names('modelchain', 'simulation')
['testxml', 'testxml2']
>>> db.get_names('something else')
```

### 33.1.7 def get\_obj(self, simoclass, name):

Returns an instance of the given type and name:

```
>>> ob = db.get_obj('operation2modelchains', 'testxml')
>>> ob.operation_mapping['first_thinning'] # doctest: +NORMALIZE_WHITESPACE
['Calculate thinning limits', 'Forced first thinning']
>>> ob.forced_op_chains # doctest: +NORMALIZE_WHITESPACE
set(['Calculate thinning limits', 'Forced stripcut', 'Forced low thinning',
    'Forced clearcut', 'Calculate productive value', 'Forced first thinning'])
>>> ob = db.get_obj('modelbase', 'testxml', 'aggregation')
>>> print ob.keys()
['sum']
>>> mbo = ob['sum']
>>> mbo.type
'aggregation'
>>> mbo.name
```

```
'sum'
>>> mbo._lib
'simoaggr.py'
>>> mbo.language
'python'
>>> mbo.dirs
''

>>> mc = db.get_obj('modelchain', 'testxml', 'simulation')# doctest: +ELLIPSIS
>>> mc.chain_groups['Forced thinnings']
['Calculate thinning limits']
>>> mc.chain_type
'simulation'
>>> len(mc.chains)
1
>>> c = mc.chains[0]
>>> c.condition
>>> c.evaluate_at
'comp_unit'
>>> c.name
'Calculate thinning limits'
>>> len(c.tasks)
1
>>> t = c.tasks[0]
>>> t.name
'Calculate thinning limits'
>>> t.value_fixer
False
>>> t.model # doctest: +ELLIPSIS
<simo.builder.modelbase.predictionmodel.PredictionModel object at ...>
>>> t.model._v_func #doctest: +ELLIPSIS
<function Productive_value_of_land_pine_Pukkala at ...>
>>> t.condition_parser
>>> t.validator
>>> p = t.model_param
>>> p.parameters
[0.5]
>>> p.rect_factor
1.0
>>> p.risk_level
1
>>> p._validator
```

### 33.1.8 def get\_valid(self, simoclass=None):

Returns if the given simoclass is valid or if simoclass is None, information on all the simoclasses.

```
>>> db.get_valid('modelchain') == \
... {'valid': True, 'types': {'simulation': {'testxml': True,
...                                         'testxml2': True}}}
True
>>> db.get_valid() == \
... {'modelchain': {'valid': True,
...                 'types': {'simulation': {'testxml': True,
...                                         'testxml2': True}}},
...  'problem_definition': {'valid': False, 'names': {}},
...  'lexicon': {'valid': False, 'names': {}},
...  'text2data': {'valid': False, 'names': {}},
...  'operation2modelchains': {'valid': True, 'names': {'testxml': True}},
...  'aggregation_definition': {'valid': False, 'names': {}},
...  'expression_definition': {'valid': False, 'names': {}},
...  'message_translation': {'valid': False, 'names': {}},
```



```
... 'lexicon_translation': {'valid': False, 'names': {}},
... 'random_values': {'valid': False, 'names': {}},
... 'modelbase': {'valid': True, 'types': {'aggregation': {'testxml': True}}},
... 'text2operation': {'valid': False, 'names': {}},
... 'output_constraint': {'valid': False, 'names': {}},
... 'simulation_control': {'valid': False, 'names': {}}}
True
```

Data input modules:



# IMPORTDATA.PY

## 34.1 class DataImporter(object):

Importing data from 'inlined' format; i.e. the data for each level in one file:

```
>>> from simo.input.importdata import DataImporter
>>> execfile('input/test/mock4importdata.py')
>>> #from simo.input.test.mock4importdata import *
>>> imp = DataImporter(inputdb, mapping, importdate,
...                    logger, logname, lexicon, 100)
>>> imp.import_data('inlined', [inline], 'simulation')
...     # doctest: +ELLIPSIS
...     # doctest: +NORMALIZE_WHITESPACE
Called Logger.log_message('testlog', 'info', 'Importing data...')
Called DataDB.add_data_from_dictionary(
    {'simulation': [(datetime.date(...), {'oid': 'simulation',
'values': [], 'id': 'simulation', 'parent id': None})]},
    0,
    0)
Called DataDB.add_data_from_dictionary(
    {'comp_unit': [(datetime.date(2009, 1, 6), {'oid': 'stand1',
'values': [(('DEV_CLASS', 1), ('MAIN_GROUP', 1), ('SOMETHING_ELSE', 99),
('Inventory_date', 733413), ('USE_RESTRICTION_SILVIC', '0'),
('USE_RESTRICTION_HARVEST', '0'))], 'id': 'stand1', 'parent id': 'simulation'})]},
    0,
    0)
Called DataDB.add_data_from_dictionary(
    {'stratum': [(datetime.date(2009, 1, 6), {'oid': 'stratum1_1',
'values': [(('BA', 200.0), ('BT', 'test'))], 'id': 'stand1-stratum1_1',
'parent id': 'stand1'})]},
    0,
    0)
Called DataDB.add_data_from_dictionary(
    {'stratum': [(datetime.date(2009, 1, 6), {'oid': 'stratum1_2',
'values': [(('BA', 22.0), ('BT', 'piece'))], 'id': 'stand1-stratum1_2',
'parent id': 'stand1'})]},
    0,
    0)
Called Logger.log_message(
    'testlog',
    'info',
    'In total 2 simulation units processed')
Called Logger.log_message(
    'testlog',
    'info',
    'In total 1 simulation units imported')
False
```

Importing data in 'by\_level' format; i.e., each data level has its' own file:

```
>>> imp.import_data('by_level', by_level, 'simulation')
...     # doctest: +NORMALIZE_WHITESPACE
...     # doctest: +ELLIPSIS
Called Logger.log_message('testlog', 'info', 'Importing data...')
Called DataDB.add_data_from_dictionary(
    {'simulation': [(datetime.date(...), {'oid': 'simulation',
'values': [], 'id': 'simulation', 'parent id': None})]},
    0,
    0)
Called DataDB.add_data_from_dictionary(
    {'comp_unit': [(datetime.date(2009, 1, 1), {'oid': 'stand1',
'values': [('MAIN_GROUP', 1), ('SOMETHING_ELSE', 99),
('Inventory_date', 733408), ('USE_RESTRICTION_SILVIC', '0'),
('USE_RESTRICTION_HARVEST', '0')], 'id': 'stand1', 'parent id': 'simulation'})]},
    0,
    0)
Called DataDB.add_data_from_dictionary(
    {'comp_unit': [(datetime.date(2009, 12, 31), {'oid': 'stand2',
'values': [('MAIN_GROUP', 1), ('SOMETHING_ELSE', 99),
('Inventory_date', 733772), ('USE_RESTRICTION_SILVIC', '0'),
('USE_RESTRICTION_HARVEST', '0')], 'id': 'stand2', 'parent id': 'simulation'})]},
    0,
    0)
Called DataDB.add_data_from_dictionary(
    {'stratum': [(datetime.date(2009, 1, 1), {'oid': 'stratum1_1',
'values': [('BA', 200.0), ('BT', 'oh')], 'id': 'stand1-stratum1_1',
'parent id': 'stand1'})]},
    0,
    0)
Called DataDB.add_data_from_dictionary(
    {'stratum': [(datetime.date(2009, 1, 1), {'oid': 'stratum1_2',
'values': [('BA', 22.0), ('BT', 'which')], 'id': 'stand1-stratum1_2',
'parent id': 'stand1'})]},
    0,
    0)
Called DataDB.add_data_from_dictionary(
    {'stratum': [(datetime.date(2009, 12, 31), {'oid': 'stratum2_1',
'values': [('BA', 31.0), ('BT', 'is')], 'id': 'stand2-stratum2_1',
'parent id': 'stand2'})]},
    0,
    0)
Called DataDB.add_data_from_dictionary(
    {'stratum': [(datetime.date(2009, 12, 31), {'oid': 'stratum2_2',
'values': [('BA', 1.0), ('BT', 'infact')], 'id': 'stand2-stratum2_2',
'parent id': 'stand2'})]},
    0,
    0)
Called Logger.log_message(
    'testlog',
    'info',
    'In total 2 simulation units processed')
Called Logger.log_message(
    'testlog',
    'info',
    'In total 2 simulation units imported')
False

>>> imp.errors
set([])
```

With skipfirst. If used like here when the first row shouldn't really be skipped, results in orphan lower data level objects in the database. Also tests id generation; the strata for stand2 have missing ids, so they'll get ids 1 and 2:

```
>>> imp.import_data('inlined', [inline2], 'simulation', skip_first=True)
...     # doctest: +ELLIPSIS
...     # doctest: +NORMALIZE_WHITESPACE
Called Logger.log_message('testlog', 'info', 'Importing data...')
Called DataDB.add_data_from_dictionary(
    {'simulation': [(datetime.date(...), {'oid': 'simulation',
'values': [], 'id': 'simulation', 'parent id': None})]},
    0,
    0)
Called DataDB.add_data_from_dictionary(
    {'stratum': [(None, {'oid': 'stratum1_1', 'values': [('BA', 200.0), ('BT', 'pretty')],
'id': 'stratum1_1', 'parent id': None})]},
    0,
    0)
Called DataDB.add_data_from_dictionary(
    {'stratum': [(None, {'oid': 'stratum1_2', 'values': [('BA', 22.0), ('BT', 'frekin')],
'id': 'stratum1_2', 'parent id': None})]},
    0,
    0)
Called DataDB.add_data_from_dictionary(
    {'comp_unit': [(datetime.date(2009, 12, 31), {'oid': 'stand2',
'values': [('MAIN_GROUP', 1), ('SOMETHING_ELSE', 99),
('Inventory_date', 733772), ('USE_RESTRICTION_SILVIC', '0'),
('USE_RESTRICTION_HARVEST', '0')], 'id': 'stand2', 'parent id': 'simulation'})]},
    0,
    0)
Called DataDB.add_data_from_dictionary(
    {'stratum': [(datetime.date(2009, 12, 31), {'oid': '1',
'values': [('BA', 31.0), ('BT', 'hmm')], 'id': 'stand2-1',
'parent id': 'stand2'})]},
    0,
    0)
Called DataDB.add_data_from_dictionary(
    {'stratum': [(datetime.date(2009, 12, 31), {'oid': '2',
'values': [('BA', 1.0), ('BT', 'wait')], 'id': 'stand2-2',
'parent id': 'stand2'})]},
    0,
    0)
Called Logger.log_message(
    'testlog',
    'info',
    'In total 1 simulation units processed')
Called Logger.log_message(
    'testlog',
    'info',
    'In total 1 simulation units imported')
False

>>> imp.errors
set([])
```

Specifying a separator to be used instead of the default whitespace:

```
>>> imp.import_data('inlined', [inline3], 'simulation', separator=';')
...     # doctest: +ELLIPSIS
...     # doctest: +NORMALIZE_WHITESPACE
Called...
    'In total 2 simulation units imported')
False
```

By level import for only one, not top level, level with the given data date:

```
>>> from datetime import date
>>> data_date = date(2009, 5, 6)
>>> imp.import_data('by_level', by_level2, 'simulation', level_ind=[1],
...                 data_date=data_date, empty_database=False)
...     # doctest: +ELLIPSIS
...     # doctest: +NORMALIZE_WHITESPACE
Called Logger.log_message('testlog', 'info', 'Importing data...')
Called DataDB.add_data_from_dictionary(
    {'stratum': [(datetime.date(2009, 5, 6),
    {'oid': 'stratum1_1', 'values': [('BA', 200.0), ('BT', 'oh')],
    'id': 'stand1-stratum1_1', 'parent id': 'stand1'})]],
    0,
    0)
Called DataDB.add_data_from_dictionary(
    {'stratum': [(datetime.date(2009, 5, 6),
    {'oid': 'stratum1_2', 'values': [('BA', 22.0), ('BT', 'which')],
    'id': 'stand1-stratum1_2', 'parent id': 'stand1'})]],
    0,
    0)
Called DataDB.add_data_from_dictionary(
    {'stratum': [(datetime.date(2009, 5, 6),
    {'oid': 'stratum2_1', 'values': [('BA', 31.0), ('BT', 'is')],
    'id': 'stand2-stratum2_1', 'parent id': 'stand2'})]],
    0,
    0)
Called DataDB.add_data_from_dictionary(
    {'stratum': [(datetime.date(2009, 5, 6),
    {'oid': 'stratum2_2', 'values': [('BA', 1.0), ('BT', 'infact')],
    'id': 'stand2-stratum2_2', 'parent id': 'stand2'})]],
    0,
    0)
Called Logger.log_message(
    'testlog',
    'info',
    'In total 0 simulation units processed')
Called Logger.log_message(
    'testlog',
    'info',
    'In total 0 simulation units imported')
False
```

### 34.1.1 def \_construct\_unique\_id(self, level, oid, parent\_id):

Construct a unique id for an object:

```
>>> imp._construct_unique_id(0, '1', 'simulation')
'1'
>>> imp._construct_unique_id(1, '1', 'stratum1_2')
Called Logger.log_message(
    'testlog',
    'error',
    "No parent path available from 'stratum' to 'stratum'!")
'stratum1_2-1'
>>> imp._construct_unique_id(1, '1', 'stand1')
'stand1-1'
```

### 34.1.2 `def _parse_date(self, datestr):`

Parse a date string into a datetime object.

```
>>> dates = ['230209', '23.07.09', '23-07-09', '23/07/09',
...         '23072009', '23.07.2009', '23-07-2009', '23/07/2009',
...         '2009-07-23', 'fail']
>>> [imp._parse_date(date) for date in dates] #doctest: +NORMALIZE_WHITESPACE
Called Logger.log_message('testlog', 'error', "Invalid date format 'fail'")
[datetime.date(2009, 2, 23), datetime.date(2009, 7, 23),
 datetime.date(2009, 7, 23), datetime.date(2009, 7, 23),
 datetime.date(2009, 7, 23), datetime.date(2009, 7, 23),
 datetime.date(2009, 7, 23), datetime.date(2009, 7, 23),
 datetime.date(2009, 7, 23), None]
```





# IMPORTOPS.PY

## 35.1 class OperationImporter(object):

Parameters:

- inputdb – input database, SimoDB instance
- fopsdb – database for forced operations, SimoDB instance
- oformat – operation format as string; one of: ‘text’, ‘xml’, ‘db’
- convdef – OperationConversion instance
- op2mc – Operation2Modelchain instance
- logger – SimoLogger instance
- logname – import log name as string

Importing operations from ‘text’ format; i.e. the data for each operation in a plain text file:

```
>>> from simo.input.importops import OperationImporter
>>> execfile('input/test/mock4importops.py')
>>> imp = OperationImporter(inputdb, fopsdb, 'text', opconv, op2mc, logger, logname)
>>> imp.import_operations('text', textops, separator)
...     # doctest: +ELLIPSIS
...     # doctest: +NORMALIZE_WHITESPACE
Called Logger.log_message('testlog', 'info',
                          'Importing forced operations...')
Called OperationConversion.operation_conv.id_delim.join(['stand1'])
Called OperationDB.add_operation(
    None,
    'comp_unit',
    0,
    0,
    'clearcut',
    'step',
    2,
    None,
    'year',
    ['Cut', 'Cut some more'],
    None)
Called DataDB.fill_dictionary('comp_unit', None)
Called DataDB.add_data_from_dictionary(
    {'comp_unit': [(datetime.date(2009, 3, 27),
    {'values': [('AREA', 2.1000000000000001), ('SC', 1.2),
    ('DEV_CLASS', 2.2999999999999998),
    ('REGEN_SP', 3)]},
    'id': 'stand1', 'parent id': 'sim1')]}],
    0,
```

```
0)
Called OperationConversion.operation_conv.id_delim.join(['stand2'])
Called OperationDB.add_operation(
    None,
    'comp_unit',
    0,
    0,
    'first_thinning',
    'step',
    7,
    None,
    'year',
    ['Calculate thinning limits'],
    None)
Called DataDB.fill_dictionary('comp_unit', None)
Called DataDB.add_data_from_dictionary(
    {'comp_unit': [(datetime.date(2009, 3, 27),
    {'values': [('AREA', 2.1000000000000001), ('SC', 1.2),
                ('DEV_CLASS', 2.2999999999999998),
                ('REGEN_SP', 3), ('TARGET_N', 500.0)],
    'id': 'stand1', 'parent id': 'sim1'})]}],
    0,
    0)
Called OperationConversion.operation_conv.id_delim.join(['stand3'])
Called Logger.log_message('testlog', 'info', 'Forced operations imported!')
```

Importing operations with timing type date and with defined operation id:

```
>>> imp.op_conv.op_id_rowpos = 8
>>> imp.op_conv.timing.type = 'date'
>>> imp.import_operations('text', textops2, separator)
...     # doctest: +ELLIPSIS
...     # doctest: +NORMALIZE_WHITESPACE
Called Logger.log_message('testlog', 'info', 'Importing forced operations...')
Called OperationConversion.operation_conv.id_delim.join(['stand1'])
Called OperationDB.add_operation(
    None,
    'comp_unit',
    0,
    0,
    'clearcut',
    'date',
    None,
    datetime.date(2009, 5, 16),
    None,
    ['Cut', 'Cut some more'],
    'op1')
Called DataDB.fill_dictionary('comp_unit', None)
Called DataDB.add_data_from_dictionary(
    {'comp_unit': [(datetime.date(2009, 3, 27),
    {'values': [('AREA', 2.1000000000000001), ('SC', 1.2),
                ('DEV_CLASS', 2.2999999999999998), ('REGEN_SP', 3),
                ('TARGET_N', 500.0)],
    'id': 'stand1', 'parent id': 'sim1'})]}],
    0,
    0)
Called OperationConversion.operation_conv.id_delim.join(['stand2'])
Called OperationDB.add_operation(
    None,
    'comp_unit',
    0,
    0,
```

```

        'first_thinning',
        'date',
        None,
        datetime.date(2009, 5, 6),
        None,
        ['Calculate thinning limits'],
        'op2')
Called DataDB.fill_dictionary('comp_unit', None)
Called DataDB.add_data_from_dictionary(
    {'comp_unit': [(datetime.date(2009, 3, 27),
        {'values': [('AREA', 2.1000000000000001), ('SC', 1.2),
            ('DEV_CLASS', 2.2999999999999998), ('REGEN_SP', 3),
            ('TARGET_N', 500.0)]},
        'id': 'stand1', 'parent id': 'sim1'})]],
    0,
    0)
Called Logger.log_message('testlog', 'info', 'Forced operations imported!')

```

#### Importing operations from a database:

```

>>> imp.import_operations('db', opdb, separator)
...     # doctest: +NORMALIZE_WHITESPACE
Called Logger.log_message('testlog', 'info',
    'Importing forced operations...')
Called operation_DataDB.conn.execute(
    'SELECT DISTINCT object_id, level, date, op_name, iteration
    FROM op_res ORDER BY date, op_id')
Called OperationDB.add_operation(
    'stand1',
    'comp_unit',
    0,
    0,
    'clearcut',
    'date',
    None,
    datetime.date(2009, 3, 27),
    None,
    ['Cut', 'Cut some more'],
    None)
Called Logger.log_message('testlog', 'info', 'Forced operations imported!')
Called operation_DataDB.close()

```

#### Data matrix modules for internal data handling of the simulator:



# BRANCHER.PY

```
>>> from minimock import Mock
>>> import numpy as np
>>> import datetime as dt
```

## 36.1 class Brancher(object):

Simulator branch construction handler.

Attributes: - `_branching_groups`: dictionary containing the branching groups and branch tasks - `_task_index`: mapping from branching group and branch task name to task indice, dictionary - `_group_index`: mapping from branching group name to a deque of task indices - `_amap`: active branching task array, a boolean numpy array with dimensions (iterations, branches, objects, branch\_tasks) - `_parent_branches`: current parent branch target index in a structure similar to data Handler's target index - `_new_branches`: new branch target index in a structure similar to data Handler's target index - `_cur_task`: current branching task object - `_is_branching`: is branching on at the moment, boolean - `_aborted`: was branching aborted for object, boolean array - `_branching_info`: branching history container, a dictionary where (iteration, new branch, object) -tuples are keys and (parent branch, operation group, operation name, branch task) -tuples are values.

Branching groups are given in a dictionary, where branching group names are the keys and the values are dictionaries. The value dictionaries themselves have branch task names as keys and lists of branch names as values

```
>>> bgroups = {
...     'bg1': (None, { 'task1': ['branch1', 'branch2'],
...                       'task2': ['branch3'] }),
...     'bg2': (None, { 'task3': ['branch4'] }) }
```

### 36.1.1 def \_\_init\_\_(self, iterations, bgroups, no\_op\_branch, branch\_limit):

Initialize a branching handler:

```
>>> from simo.matrix.brancher import Brancher
>>> br = Brancher(5, bgroups, False, None)
```

### 36.1.2 def \_set\_branch\_groups(self):

Construct branching task and group indices from the branching group dictionaries:

```
>>> br._set_branch_groups()
>>> br._task_index['bg1'][('task1', 'branch1')]
0
>>> br._task_index['bg1'][('task1', 'branch2')]
```

```
1
>>> br._task_index['bg1'] [('task2', 'branch3')]
2
>>> br._task_index['bg2'] [('task3', 'branch4')]
3
>>> br._group_index['bg1']
deque([0, 1, 2])
>>> br._group_index['bg2']
deque([3])
>>> br._amap
array([], shape=(5, 1, 0, 4), dtype=bool)
>>> br._parent_branches
>>> br._new_branches
```

### 36.1.3 def add\_object(self, branch, num):

Add a number of main level objects to the brancher. This method is needed when constructing the data matrix for the simulation. The method increases active branching task mapping (amap) size in object (3rd) dimension

```
>>> br.add_object(1)
>>> br._amap[0,0,:,:]
array([[ True,  True,  True,  True]], dtype=bool)

>>> br.add_object(1)
>>> br._amap[0,0,:,:]
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)
```

### 36.1.4 def add\_branch(self, iteration, frombranch, tobranch, obj):

Add a new branch to the data matrix by copying values from “parent” branch. The method increases active branching task mapping (amap) size in branch (2nd) dimension

```
>>> br._amap[0,0,1,1] = False
>>> br.add_branch(0, 0, 1, 1)
>>> br._amap[0,1,:,:]
array([[ True,  True,  True,  True],
       [ True, False,  True,  True]], dtype=bool)
```

### 36.1.5 def start(self, task, branchname, data, simlevel, depth, pred\_mem, oper\_mem):

Start branching for objects defined in the target index (tind):

```
>>> br._amap[1,0,1,1:] = False
>>> br._amap[4,0,0,(2,3)] = False
>>> task = Mock('Task')
>>> task.branching_group = 'bg1'
>>> task.name = 'task1'
>>> data = Mock('Data')
>>> data.get_tind.mock_returns = (
...     np.array([[0,0,0,0,0],
...               [1,0,1,0,0],
...               [4,0,0,0,0],
...               [4,0,1,0,0]], dtype=int),
...     None)
```

```

>>> data.add_branch.mock_returns = \
...     np.array([[0,1,0,0,0],
...               [1,2,1,0,0],
...               [4,3,1,0,0]], dtype=int)
>>> oper_mem = Mock('OperationMemory')
>>> pred_mem = Mock('PredictionMemory')
>>> br.start(task, 'branch1', data, 1, 0, pred_mem, oper_mem)
Called Data.get_tind(1, 0)
Called Data.add_branch(
    array([[0, 0, 0, 0, 0],
           [4, 0, 0, 0, 0],
           [4, 0, 1, 0, 0]]))
Called Data.block(
    1,
    0,
    array([[0, 0, 0, 0, 0],
           [4, 0, 0, 0, 0],
           [4, 0, 1, 0, 0]]))
Called PredictionMemory.add_branch(
    array([[0, 0, 0, 0, 0],
           [4, 0, 0, 0, 0],
           [4, 0, 1, 0, 0]]),
    array([[0, 1, 0, 0, 0],
           [1, 2, 1, 0, 0],
           [4, 3, 1, 0, 0]]))
Called OperationMemory.add_branch(
    array([[0, 0, 0, 0, 0],
           [4, 0, 0, 0, 0],
           [4, 0, 1, 0, 0]]),
    array([[0, 1, 0, 0, 0],
           [1, 2, 1, 0, 0],
           [4, 3, 1, 0, 0]]))
>>> br._parent_branches
array([[0, 0, 0, 0, 0],
       [1, 0, 1, 0, 0],
       [4, 0, 0, 0, 0],
       [4, 0, 1, 0, 0]])
>>> br._new_branches
array([[0, 1, 0, 0, 0],
       [1, 0, 1, 0, 0],
       [1, 2, 1, 0, 0],
       [4, 3, 1, 0, 0]])
>>> br.is_branching
True
>>> br._task # doctest: +ELLIPSIS
<Mock ... Task>

```

**Data handler is responsible for adding new branches using brancher's add\_branch method::**

```

>>> br.add_branch(0, 0, 1, 0)
>>> br.add_branch(1, 0, 2, 0)
>>> br.add_branch(4, 0, 3, 0)

```

### 36.1.6 def check\_operation(self, model, data, level, depth):

Check if the current operation is a branching operation and if a new branch should be created or not. Simulator uses data handler's get\_tind method to deduce if the operation was done to any of the current branching objects.

```

>>> data.get_tind.mock_returns = (
...     np.array([[0,1,0,0,0]], dtype=int),

```

```
...     None)
>>> task.branching_ops = (1,5,6)
>>> model = Mock('Model')
>>> model.group = 'mock group'
>>> model.name = 'mock name'
>>> model.index = 0
>>> model.dropped = np.array([[0,1,0,0,0]], dtype=int)
>>> br.check_operation(model, data, 1, 0)
>>> br._aborted
array([ True,  True,  True,  True], dtype=bool)

>>> model.index = 5
>>> br.check_operation(model, data, 1, 0)
Called Data.get_tind(1, 0)
>>> br._aborted
array([ True,  True,  True,  True], dtype=bool)
>>> model.dropped = None
>>> br.check_operation(model, data, 1, 0)
Called Data.get_tind(1, 0)
>>> br._aborted
array([False,  True,  True,  True], dtype=bool)

>>> data.get_tind.mock_returns = (
...     np.array([[4,3,1,0,0]], dtype=int),
...     None)
>>> br.check_operation(model, data, 1, 0)
Called Data.get_tind(1, 0)
>>> br._aborted
array([False,  True,  True, False], dtype=bool)
```

### 36.1.7 def stop(self, branchname, data, timespan, simlevel, depth, pred\_mem, oper\_mem):

Stop branching and reset initialized branching targets and branching task:

```
>>> data.get_date.mock_returns = np.array([dt.date(2000,1,1),
...                                         dt.date(2000,1,1),
...                                         dt.date(2000,1,1),
...                                         dt.date(2000,1,1)],
...                                         dtype=dt.date)
>>> timespan = Mock('Timespan')
>>> timespan.time_step = 3
>>> timespan.unit = 'year'
>>> br.stop('branch1', data, timespan, 1, 0, pred_mem, oper_mem)
Called Data.del_branch(array([[1, 0, 1, 0, 0],
                               [1, 2, 1, 0, 0]]))
Called PredictionMemory.del_objects(1, 0, [1], 1)
Called OperationMemory.del_objects(1, 0, [1], 1)
Called PredictionMemory.del_objects(1, 2, [1], 1)
Called OperationMemory.del_objects(1, 2, [1], 1)
Called Data.get_date(
    array([[0, 1, 0, 0, 0],
           [1, 0, 1, 0, 0],
           [1, 2, 1, 0, 0],
           [4, 3, 1, 0, 0]]))
Called Data.release(
    1,
    0,
    array([[0, 0, 0, 0, 0],
           [1, 0, 1, 0, 0],
```



```
        [4, 0, 0, 0, 0],
        [4, 0, 1, 0, 0]])
>>> br._branching_info[(0,1,0)] # doctest: +ELLIPSIS
(0, 'mock group', 'mock name', 'branch1', datetime.date(2002, 12, 31))
>>> br._branching_info[(4,3,1)] # doctest: +ELLIPSIS
(0, 'mock group', 'mock name', 'branch1', datetime.date(2002, 12, 31))
>>> br.is_branching
False
>>> br._parent_branches
>>> br._new_branches
>>> br._task
```

### 36.1.8 def is\_blocked(self, task, branchname):

Check if current branching group / task combination is blocked, i.e. all tasks in the group have been done for all units already

```
>>> br.is_blocked(task, 'branch1')
False
>>> br._amap[:, :, :, :] = False
>>> br.is_blocked(task, 'branch1')
True
```



# HANDLER.PY

```
>>> from minimock import Mock
```

## 37.1 class Handler(object):

Matrix handler class for accessing data during simulation. Keeps the data matrix and the object linking information consistent.

### 37.1.1 def \_\_init\_\_(self, iterations, lexicon, attributes, mcdepth, main-level, branchgroups, noopbranch, sim, objcount=50, branchcount=1, branch\_limit=None):

Initialize a data handler taking the lexicon with the hierarchy description; object index to data hierarchy level ordinal and object index to data lineage information mapping ; and number of attributes as input:

```
>>> hr = {}
>>> hr[0] = {'ordinal':0, 'lineage':set([0]), 'parent': None}
>>> hr[1] = {'ordinal':1, 'lineage':set([0]), 'parent': 0}
>>> hr[2] = {'ordinal':2, 'lineage':set([0]), 'parent': 1}
>>> lexicon = Mock('Lexicon')
>>> sim = Mock('simulator')
>>> lexicon.hierarchy = hr
>>> lexicon.get_variable_name.mock_returns = "VARNAME"
>>> lexicon.get_level_name.mock_returns = "LEVNAME"
>>> from simo.matrix.handler import Handler

>>> bgroups = {}
>>> h = Handler(1, lexicon, 10, 1, 1, bgroups, False, sim, 0)

>>> bgroups = {'group1': (None, {'task1': ['branch1']})}
>>> h = Handler(1, lexicon, 10, 1, 1, bgroups, False, sim, 0)
```

### 37.1.2 def add\_objects(self, iteration, branch, level, num, parlev=None, parind=None, oids=None):

Add number of objects to given branch and level in the data matrix. Optionally take the parent level and object indices and a list of object ids as input. Return a target index (similar as from get\_tind) of the created object indices:

```
>>> h.add_objects(0, 0, 0, 1, None, None, ['SIMULATION'])
array([[0, 0, 0, 0, 0]])
>>> h.add_objects(0, 0, 1, 3, 0, 0, ['UID001', 'UID002', 'UID003'])
```

```
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0]])
>>> h.linkage.links[(0,0)][(0,0)]
{1: [0, 1, 2]}
>>> h.matrix.omap
array([[[[ True, False, False],
         [ True,  True,  True],
         [False, False, False]]]], dtype=bool)
>>> h.matrix.emap
array([[[[ True, False, False]],
<BLANKLINE>
         [[ True,  True,  True]],
<BLANKLINE>
         [[False, False, False]]]], dtype=bool)
>>> h.add_objects(0, 0, 1, 2, 0, 0)
array([[0, 0, 3, 0, 0],
       [0, 0, 4, 0, 0]])
>>> h.matrix.omap
array([[[[ True, False, False, False, False],
         [ True,  True,  True,  True,  True],
         [False, False, False, False, False]]]], dtype=bool)
>>> h.matrix.emap
array([[[[ True, False, False, False, False]],
<BLANKLINE>
         [[ True,  True,  True,  True,  True]],
<BLANKLINE>
         [[False, False, False, False, False]]]], dtype=bool)
>>> h.linkage.links[(0,0)][(0,0)]
{1: [0, 1, 2, 3, 4]}
>>> h.add_objects(0, 0, 2, 1, 1, 3)
array([[0, 0, 0, 0, 0]])
>>> h.linkage.links[(0,0)][(0,0)]
{1: [0, 1, 2, 3, 4], 2: [0]}
>>> h.add_objects(0, 0, 2, 2, 1, 4)
array([[0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0]])
>>> h.linkage.links[(0,0)][(1,4)]
{0: [0], 2: [1, 2]}
>>> h.linkage.links[(0,0)][(1,2)]
{0: [0]}
>>> h.matrix.emap
array([[[[ True, False, False, False, False]],
<BLANKLINE>
         [[ True,  True,  True,  True,  True]],
<BLANKLINE>
         [[ True,  True,  True, False, False]]]], dtype=bool)
```

### 37.1.3 def add\_branch(self, tind):

Add a new branch to the data matrix by copying values from “parent” branch:

```
>>> import numpy
>>> tind = numpy.array([0,0,4,0,0], dtype=int)
>>> h.add_branch(tind)
array([[0, 1, 4, 0, 0]])
>>> h.matrix.omap[0,:2,:,:]
array([[[ True, False, False, False, False],
         [ True,  True,  True,  True,  True],
         [ True,  True,  True, False, False]],
```

```

<BLANKLINE>
[[ True, False, False, False, False],
 [False, False, False, False, True],
 [ True,  True, False, False, False]], dtype=bool)
>>> h.matrix.emap[0,:2, :, 0, :]
array([[ True, False, False, False, False],
       [ True,  True,  True,  True,  True],
       [ True,  True,  True, False, False]],
<BLANKLINE>
[[ True, False, False, False, False],
 [False, False, False, False,  True],
 [ True,  True, False, False, False]], dtype=bool)

>>> keys = h.linkage.links[(0,1)].keys()
>>> keys.sort()
>>> keys
[(0, 0), (1, 4), (2, 0), (2, 1)]

>>> li = h.linkage.links[(0,1)]
>>> li[(2, 0)]
{0: [0], 1: [4]}
>>> li[(0, 0)]
{1: [4], 2: [0, 1]}
>>> li[(2, 1)]
{0: [0], 1: [4]}
>>> li[(1, 4)]
{0: [0], 2: [0, 1]}

>>> keys = h.ind2id.ids[(0,1)].keys()
>>> keys.sort()
>>> keys
[(0, 0), (1, 4), (2, 0), (2, 1)]

>>> ind = h.ind2id.ids[(0,1)]
>>> ind[(2, 0)]
'SIMULATION-1-0'
>>> ind[(0, 0)]
'simulation'
>>> ind[(2, 1)]
'SIMULATION-1-1'
>>> ind[(1, 4)]
'SIMULATION-1'

>>> tind = numpy.array([[0,0,0,0,0],[0,0,3,0,0]], dtype=int)
>>> h.add_branch(tind)
array([[0, 1, 0, 0, 0],
       [0, 1, 3, 0, 0]])
>>> h.matrix.omap[0,:2, :, :]
array([[ True, False, False, False, False],
       [ True,  True,  True,  True,  True],
       [ True,  True,  True, False, False]],
<BLANKLINE>
[[ True, False, False, False, False],
 [ True, False, False,  True,  True],
 [ True,  True,  True, False, False]], dtype=bool)
>>> h.matrix.emap[0,:2, :, 0, :]
array([[ True, False, False, False, False],
       [ True,  True,  True,  True,  True],
       [ True,  True,  True, False, False]],
<BLANKLINE>
[[ True, False, False, False, False],
 [ True, False, False,  True,  True],

```

```
[ True,  True,  True, False, False]], dtype=bool)

>>> keys = h.linkage.links[(0,1)].keys()
>>> keys.sort()
>>> keys
[(0, 0), (1, 0), (1, 3), (1, 4), (2, 0), (2, 1), (2, 2)]

>>> li = h.linkage.links[(0,1)]
>>> li[(1, 3)]
{0: [0], 2: [2]}
>>> li[(2, 1)]
{0: [0], 1: [4]}
>>> li[(2, 0)]
{0: [0], 1: [4]}
>>> li[(0, 0)]
{1: [0, 3, 4], 2: [0, 1, 2]}
>>> li[(1, 4)]
{0: [0], 2: [0, 1]}
>>> li[(2, 2)]
{0: [0], 1: [3]}
>>> li[(1, 0)]
{0: [0]}

>>> keys = h.ind2id.ids[(0,1)].keys()
>>> keys.sort()
>>> keys
[(0, 0), (1, 0), (1, 3), (1, 4), (2, 0), (2, 1), (2, 2)]

>>> ind = h.ind2id.ids[(0,1)]
>>> ind[(1, 3)]
'SIMULATION-0'
>>> ind[(2, 1)]
'SIMULATION-1-1'
>>> ind[(2, 0)]
'SIMULATION-1-0'
>>> ind[(0, 0)]
'simulation'
>>> ind[(1, 4)]
'SIMULATION-1'
>>> ind[(2, 2)]
'SIMULATION-0-0'
>>> ind[(1, 0)]
'UID001'
```

### 37.1.4 def add\_iteration(self, fromit):

```
>>> h2 = Handler(4, lexicon, 10, 1, 1, bgroups, False, sim, 0)
>>> h2.add_objects(0, 0, 0, 1, None, None, ['SIMULATION'])
array([[0, 0, 0, 0, 0]])
>>> h2.add_objects(0, 0, 1, 3, 0, 0, ['UID001', 'UID002', 'UID003'])
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0]])
>>> h2.add_iteration(0)
>>> h2.matrix.omap
array([[[[ True, False, False],
         [ True,  True,  True],
         [False, False, False]]],
       <BLANKLINE>
       <BLANKLINE>
       <BLANKLINE>])
```

```

[[[ True, False, False],
  [ True,  True,  True],
  [False, False, False]]],
<BLANKLINE>
[[[ True, False, False],
  [ True,  True,  True],
  [False, False, False]]],
<BLANKLINE>
[[[ True, False, False],
  [ True,  True,  True],
  [False, False, False]]], dtype=bool)
>>> h2.linkage.links[(2,0)][(0,0)]
{1: [0, 1, 2]}
>>> h2.ind2id.ids[(2,0)][(0,0)]
'SIMULATION'
>>> h2.ind2id.ids[(3,0)][(1,2)]
'UID003'
```

### 37.1.5 def set\_value(self, level, tind, attrs, values, modelname=None, indicate-change=False):

Store attribute value(s) into the data matrix on the given level using the provided target index (tind) combinations. Each target index consists of branch and object indices. Attributes (attrs) and values are iterable objects. The length of tind and values must be equal. Furthermore, the length of each item in values must match the length of attrs:

```

>>> epsilon = 0.000001
>>> tind = numpy.array([[0,0,0],[0,1,0]], dtype=int)
>>> h.set_value(0, tind, (0,), [(10.1, 20.1)])
>>> h.matrix.mx[0,0,0,0,0] - 10.1 < epsilon
True
>>> tind = numpy.array([[0,0,0],[0,0,1],[0,0,2],[0,0,3],[0,0,4],[0,1,0],
...                    [0,1,1],[0,1,2],[0,1,3],[0,1,4]], dtype=int)
>>> h.set_value(1, tind, (0,1),
...            [[1.1, 2.1, 2.2, 2.3, 2.4, 1.1, 2.1, 2.2, 2.3, 2.4],
...             [5.0, 5.1, 5.2, 5.3, 5.4, 5.0, 5.1, 5.2, 5.3, 5.4]])
>>> h.matrix.mx[0,0,1,2,1] - 5.2 < epsilon
True
>>> tind=numpy.array([[0,1,0],[0,1,1],[0,1,2],[0,1,3],[0,1,4]],dtype=int)
>>> h.set_value(1, tind, (0,), [[3.1, 3.1, 3.2, 3.3, 3.4]])
>>> h.matrix.mx[0,1,1,2,0] - 3.2 < epsilon
True
>>> tind = numpy.array([[0,0,0],[0,0,1],[0,0,2]], dtype=int)
>>> h.set_value(2, tind, (0,), [[.5, .6, .7]])
>>> h.matrix.mx[0,0,2,1,0] - .6 < epsilon
True
>>> h.set_value(2, tind, 0, [.5, .6, .7])
>>> h.matrix.mx[0,0,2,1,0] - .6 < epsilon
True
```

### 37.1.6 def del\_objects(self, iteration, branch, level, objs):

Delete objects from data by nullifying data and removing the relationship links:

```

>>> h.del_objects(0, 0, 1, [0, 1])
>>> h.matrix.mx[0,0,1,0,:] # doctest: +NORMALIZE_WHITESPACE
```

```
... # doctest: +ELLIPSIS
array([ NaN,  NaN,  NaN,  NaN,  NaN,  NaN,  NaN,  NaN,  NaN]...)
>>> h.matrix.omap[0,0,1,1]
False
>>> h.matrix.omap[0,0,1,2]
True
>>> h.linkage.links[(0,0)][(0,0)]
{1: [2, 3, 4], 2: [0, 1, 2]}
>>> h.del_objects(0, 0, 1, [2, 3])
>>> h.add_objects(0, 0, 1, 4, 0, 0)
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 3, 0, 0]])
>>> h.del_objects(0, 0, 1, [0, 1])
>>> h.set_value(1, numpy.array([[0,0,2,0,0],[0,0,3,0,0]]), [0,1],
...           [[2.2,2.3], [5.2,5.3]])
```

### 37.1.7 def get\_tind(self, level, depthind, datalevel=None, throughlevel=None, vals4all=True, order=None):

Get index of target branches and objects for the specified data level (evaluation level in model chain) and the model chain depth. If datalevel differs from the (evaluation) level, each branch object index pair is augmented with the start and stop indices that map the rows in the data matrix returned by get\_value to the objects at the (evaluation) level. Setting the throughlevel attribute forces the collection of relatives determined by the datalevel attribute to be collected via the through level; e.g. for a stand-stratum-tree hierarchy; for each stratum all the trees of a stand are returned set as data objects if the level == stratum, datalevel == tree and throughlevel == stand. If throughlevel is defined, the values are either for all data-level objects, or for self (this can happen only in aggregation model conditions). Parameter valuelevel can be defined in situations when datalevel differs from level, but the actual values should be from a level other than datalevel. The order of the targets can be forced using order -parameter, but this requires that the indices of all targets are known beforehand.

Dimensions:

- 1. dimension: iteration
- 1. dimension: branch
- 1. dimension: object index at the evaluation level
- 1. dimension: start index of data belonging to the object in the result data matrix of get\_value
- 1. dimension: stop index of data belonging to the object in the result data matrix of get\_value

```
>>> h.matrix.emap[:,2,::,::]
array([[[[ True, False, False, False, False]],
<BLANKLINE>
      [[False, False,  True,  True,  True]],
<BLANKLINE>
      [[ True,  True,  True, False, False]],
<BLANKLINE>
<BLANKLINE>
      [[ True, False, False, False, False]],
<BLANKLINE>
      [[ True, False, False,  True,  True]],
<BLANKLINE>
      [[ True,  True,  True, False, False]]]], dtype=bool)
>>> h.get_tind(0, 0)
(array([[0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0]]), set([]))
>>> h.get_tind(1, 0)
(array([[0, 0, 2, 0, 0],
```



```

        [0, 0, 3, 0, 0],
        [0, 0, 4, 0, 0],
        [0, 1, 0, 0, 0],
        [0, 1, 3, 0, 0],
        [0, 1, 4, 0, 0]], set([]))
>>> h.get_tind(0, 0, 1)
(array([[0, 0, 0, 0, 3],
        [0, 1, 0, 3, 6]]), set([]))
>>> h.get_tind(1, 0, 0)
(array([[0, 0, 2, 0, 1],
        [0, 0, 3, 1, 2],
        [0, 0, 4, 2, 3],
        [0, 1, 0, 3, 4],
        [0, 1, 3, 4, 5],
        [0, 1, 4, 5, 6]]), set([]))
>>> tind = numpy.array([[0,0,2,0,0]], dtype=int)
>>> h.add_branch(tind)
array([[0, 1, 2, 0, 0]])
>>> h.matrix.omap[0,:2,:,:]
array([[ [ True, False, False, False, False],
         [False, False,  True,  True,  True],
         [ True,  True,  True, False, False]],
<BLANKLINE>
        [ [ True, False, False, False, False],
          [ True, False,  True,  True,  True],
          [ True,  True,  True, False, False]], dtype=bool)
>>> tind = numpy.array([[0,1,0,0,0]], dtype=int)
>>> h.add_branch(tind)
array([[0, 0, 0, 0, 0]])
>>> h.matrix.omap[0,:2,:,:]
array([[ [ True, False, False, False, False],
         [ True, False,  True,  True,  True],
         [ True,  True,  True, False, False]],
<BLANKLINE>
        [ [ True, False, False, False, False],
          [ True, False,  True,  True,  True],
          [ True,  True,  True, False, False]], dtype=bool)
>>> h.get_tind(0,0)
(array([[0, 0, 0, 0, 0],
        [0, 1, 0, 0, 0]]), set([]))
>>> h.get_tind(1,0)
(array([[0, 0, 0, 0, 0],
        [0, 0, 2, 0, 0],
        [0, 0, 3, 0, 0],
        [0, 0, 4, 0, 0],
        [0, 1, 0, 0, 0],
        [0, 1, 2, 0, 0],
        [0, 1, 3, 0, 0],
        [0, 1, 4, 0, 0]]), set([]))
>>> h.get_tind(0, 0, 1)
(array([[0, 0, 0, 0, 4],
        [0, 1, 0, 4, 8]]), set([]))
>>> h.get_tind(1, 0, 0)
(array([[0, 0, 0, 0, 1],
        [0, 0, 2, 1, 2],
        [0, 0, 3, 2, 3],
        [0, 0, 4, 3, 4],
        [0, 1, 0, 4, 5],
        [0, 1, 2, 5, 6],
        [0, 1, 3, 6, 7],
        [0, 1, 4, 7, 8]]), set([]))

```

Get target index using throughlevel value:

```
>>> h.get_tind(1, 0, 2, 0)
(array([[ 0,  0,  0,  0,  3],
       [ 0,  0,  2,  3,  6],
       [ 0,  0,  3,  6,  9],
       [ 0,  0,  4,  9, 12],
       [ 0,  1,  0, 12, 15],
       [ 0,  1,  2, 15, 18],
       [ 0,  1,  3, 18, 21],
       [ 0,  1,  4, 21, 24]]), set([]))

>>> h.active_dataobjects
array([[0, 0, 0],
       [0, 0, 1],
       [0, 0, 2],
       [0, 0, 0],
       [0, 0, 1],
       [0, 0, 2],
       [0, 0, 0],
       [0, 0, 1],
       [0, 0, 2],
       [0, 0, 0],
       [0, 0, 1],
       [0, 0, 2],
       [0, 1, 0],
       [0, 1, 1],
       [0, 1, 2],
       [0, 1, 0],
       [0, 1, 1],
       [0, 1, 2],
       [0, 1, 0],
       [0, 1, 1],
       [0, 1, 2],
       [0, 1, 0],
       [0, 1, 1],
       [0, 1, 2]])

>>> h.get_tind(1, 0, 1, 0)
(array([[ 0,  0,  0,  0,  4],
       [ 0,  0,  2,  4,  8],
       [ 0,  0,  3,  8, 12],
       [ 0,  0,  4, 12, 16],
       [ 0,  1,  0, 16, 20],
       [ 0,  1,  2, 20, 24],
       [ 0,  1,  3, 24, 28],
       [ 0,  1,  4, 28, 32]]), set([]))

>>> h.active_dataobjects
array([[0, 0, 0],
       [0, 0, 2],
       [0, 0, 3],
       [0, 0, 4],
       [0, 0, 0],
       [0, 0, 2],
       [0, 0, 3],
       [0, 0, 4],
       [0, 0, 0],
       [0, 0, 2],
       [0, 0, 3],
       [0, 0, 4],
       [0, 0, 0],
       [0, 0, 2],
       [0, 0, 3],
       [0, 0, 4],
       [0, 1, 0],
       [0, 1, 2],
```

```

[0, 1, 3],
[0, 1, 4],
[0, 1, 0],
[0, 1, 2],
[0, 1, 3],
[0, 1, 4],
[0, 1, 0],
[0, 1, 2],
[0, 1, 3],
[0, 1, 4],
[0, 1, 0],
[0, 1, 2],
[0, 1, 3],
[0, 1, 4]])
>>> h.get_tind(2, 0, 2, 0)
(array([[ 0,  0,  0,  0,  3],
       [ 0,  0,  1,  3,  6],
       [ 0,  0,  2,  6,  9],
       [ 0,  1,  0,  9, 12],
       [ 0,  1,  1, 12, 15],
       [ 0,  1,  2, 15, 18]]), set([]))
>>> h.active_dataobjects
array([[0, 0, 0],
       [0, 0, 1],
       [0, 0, 2],
       [0, 0, 0],
       [0, 0, 1],
       [0, 0, 2],
       [0, 0, 0],
       [0, 0, 1],
       [0, 0, 2],
       [0, 1, 0],
       [0, 1, 1],
       [0, 1, 2],
       [0, 1, 0],
       [0, 1, 1],
       [0, 1, 2],
       [0, 1, 0],
       [0, 1, 1],
       [0, 1, 2]])

```

Get target index using throughlevel and so that vals4all attribute is False:

```

>>> h.get_tind(1, 0, 2, 0, False)
(array([[ 0,  0,  0,  0,  3],
       [ 0,  0,  2,  3,  6],
       [ 0,  0,  3,  6,  9],
       [ 0,  0,  4,  9, 12],
       [ 0,  1,  0, 12, 15],
       [ 0,  1,  2, 15, 18],
       [ 0,  1,  3, 18, 21],
       [ 0,  1,  4, 21, 24]]), set([]))
>>> h.active_dataobjects
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 2],
       [0, 0, 2],
       [0, 0, 2],
       [0, 0, 3],
       [0, 0, 3],
       [0, 0, 3],

```

```
[0, 0, 4],
[0, 0, 4],
[0, 0, 4],
[0, 1, 0],
[0, 1, 0],
[0, 1, 0],
[0, 1, 2],
[0, 1, 2],
[0, 1, 2],
[0, 1, 3],
[0, 1, 3],
[0, 1, 3],
[0, 1, 4],
[0, 1, 4],
[0, 1, 4]])
>>> h.get_tind(1, 0, 1, 0, False)
(array([[ 0,  0,  0,  0,  4],
       [ 0,  0,  2,  4,  8],
       [ 0,  0,  3,  8, 12],
       [ 0,  0,  4, 12, 16],
       [ 0,  1,  0, 16, 20],
       [ 0,  1,  2, 20, 24],
       [ 0,  1,  3, 24, 28],
       [ 0,  1,  4, 28, 32]]), set([]))
>>> h.active_dataobjects
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 2],
       [0, 0, 2],
       [0, 0, 2],
       [0, 0, 2],
       [0, 0, 3],
       [0, 0, 3],
       [0, 0, 3],
       [0, 0, 3],
       [0, 0, 4],
       [0, 0, 4],
       [0, 0, 4],
       [0, 0, 4],
       [0, 1, 0],
       [0, 1, 0],
       [0, 1, 0],
       [0, 1, 0],
       [0, 1, 2],
       [0, 1, 2],
       [0, 1, 2],
       [0, 1, 2],
       [0, 1, 3],
       [0, 1, 3],
       [0, 1, 3],
       [0, 1, 3],
       [0, 1, 4],
       [0, 1, 4],
       [0, 1, 4],
       [0, 1, 4]])
>>> h.get_tind(2, 0, 2, 0, False)
(array([[ 0,  0,  0,  0,  3],
       [ 0,  0,  1,  3,  6],
       [ 0,  0,  2,  6,  9],
       [ 0,  1,  0,  9, 12],
```

```

        [ 0,  1,  1, 12, 15],
        [ 0,  1,  2, 15, 18]]), set({}))
>>> h.active_dataobjects
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 1],
       [0, 0, 1],
       [0, 0, 1],
       [0, 0, 2],
       [0, 0, 2],
       [0, 0, 2],
       [0, 1, 0],
       [0, 1, 0],
       [0, 1, 0],
       [0, 1, 1],
       [0, 1, 1],
       [0, 1, 1],
       [0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])

```

Get target index, but with a predefined order:

```

>>> order = numpy.array([[0,0,1], [0,0,0]], dtype=int)
>>> h.get_tind(0, 0, None, None, True, None, order)
(array([[0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0]]), set({}))
>>> order = numpy.array([[0,1,4],[0,1,3],[0,1,2],[0,1,0],[0,0,4],
...                      [0,0,3],[0,0,2],[0,0,0]], dtype=int)
>>> h.get_tind(1, 0, 2, None, True, None, order)
(array([[0, 1, 4, 0, 2],
       [0, 1, 3, 2, 3],
       [0, 1, 2, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 4, 3, 5],
       [0, 0, 3, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 0, 0, 0]]), set([2, 3, 5, 6, 7]))
>>> h.active_dataobjects
array([[0, 1, 0],
       [0, 1, 1],
       [0, 1, 2],
       [0, 0, 1],
       [0, 0, 2]])

```

### 37.1.8 def get\_value(self, tind, attrs, ignore\_missing=False):

Get attribute value(s) from the data matrix. If any of the target objects are missing values and ignore\_missing is False, return also error messages, an error object target index (for error logging), and a set of object indices that were missing values

The return value is (if some values are missing): (values, ([error\_messages], error\_tind, removable\_objects)) And if no values are missing: (values, None)

Get values so that all objects have missing values (and all target objects will be removed)

```

>>> from numpy import array, float32, NaN
>>> EPSILON = 0.00001
>>> tind, toremove = h.get_tind(0, 0)
>>> vals, err = h.get_value(tind, [0,1]) # doctest: +NORMALIZE_WHITESPACE

```

```
Called Lexicon.get_variable_name(0, 1)
Called Lexicon.get_level_name(0)
>>> expected = array([[10.1, 20.1], [NaN, NaN]], dtype=float32)
>>> expected - vals < EPSILON
array([[ True,  True],
       [False, False]], dtype=bool)
>>> err[0] # doctest: +NORMALIZE_WHITESPACE
[("value for variable 'VARNAME' at level 'LEVNAME' not found",
array([0, 1]))]
>>> err[1]
array([[0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0]])
>>> err[2]
set([0, 1])
```

Get values so that no values are missing

```
>>> tind, toremove = h.get_tind(1, 0)
>>> vals, err = h.get_value(tind, 0)
>>> expected = array([3.1, 2.2, 2.3, 2.4, 3.1, 2.2, 3.3, 3.4],
...                  dtype=float32)
>>> expected - vals < EPSILON # doctest: +NORMALIZE_WHITESPACE
array([ True,  True,  True,  True,  True,  True,  True,  True],
      dtype=bool)
>>> err

>>> vals, err = h.get_value(tind, [0,1])
>>> expected = array([[ 3.1,  2.2,  2.3,  2.4,  3.1,  2.2,  3.3,  3.4],
...                  [ 5. ,  5.2,  5.3,  5.4,  5. ,  5.2,  5.3,  5.4]],
...                  dtype=float32)
>>> expected - vals < EPSILON # doctest: +NORMALIZE_WHITESPACE
array([[ True,  True,  True,  True,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True,  True,  True,  True]],
      dtype=bool)
>>> err

>>> tind, toremove = h.get_tind(0, 0, 1)
>>> vals, err = h.get_value(tind, 1)
>>> expected = array([ 5. ,  5.2,  5.3,  5.4,  5. ,  5.2,  5.3,  5.4],
...                  dtype=float32)
>>> expected -vals < EPSILON # doctest: +NORMALIZE_WHITESPACE
array([ True,  True,  True,  True,  True,  True,  True,  True],
      dtype=bool)
>>> err

>>> vals, err = h.get_value(tind, [0,1])
>>> expected = array([[ 3.1,  2.2,  2.3,  2.4,  3.1,  2.2,  3.3,  3.4],
...                  [ 5. ,  5.2,  5.3,  5.4,  5. ,  5.2,  5.3,  5.4]],
...                  dtype=float32)
>>> expected - vals < EPSILON # doctest: +NORMALIZE_WHITESPACE
array([[ True,  True,  True,  True,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True,  True,  True,  True]],
      dtype=bool)
>>> err
```

Get values from child level so that some values are missing

```
>>> tind, toremove = h.get_tind(1, 0)
>>> h.matrix.mx[0,0,1,(0,3),0] = numpy.NaN
>>> h.matrix.mx[0,0,1,0,1] = numpy.NaN
>>> vals, err = h.get_value(tind, [0,1])
Called Lexicon.get_variable_name(1, 0)
```

```

Called Lexicon.get_level_name(1)
Called Lexicon.get_variable_name(1, 1)
Called Lexicon.get_level_name(1)
>>> expected = array([[ NaN,  2.2,  NaN,  2.4,  3.1,  2.2,  3.3,  3.4],
...                  [ NaN,  5.2,  5.3,  5.4,  5. ,  5.2,  5.3,  5.4]],
...                  dtype=float32)
>>> expected - vals < EPSILON # doctest: +NORMALIZE_WHITESPACE
array([[False,  True, False,  True,  True,  True,  True,  True],
       [False,  True,  True,  True,  True,  True,  True,  True]],
       dtype=bool)
>>> for e in err[0]: print e # doctest: +NORMALIZE_WHITESPACE
("value for variable 'VARNAME' at level 'LEVNAME' not found",
array([0, 2]))
("value for variable 'VARNAME' at level 'LEVNAME' not found",
array([0]))
>>> err[1]
array([[0, 0, 0, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 3, 0, 0],
       [0, 0, 4, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 1, 2, 0, 0],
       [0, 1, 3, 0, 0],
       [0, 1, 4, 0, 0]])
>>> err[2]
set([0, 2])
>>> vals, err = h.get_value(tind, 0)
Called Lexicon.get_variable_name(1, 0)
Called Lexicon.get_level_name(1)
>>> expected = array([ NaN,  2.2,  NaN,  2.4,  3.1,  2.2,  3.3,  3.4],
...                  dtype=float32)
>>> expected - vals < EPSILON # doctest: +NORMALIZE_WHITESPACE
array([False,  True, False,  True,  True,  True,  True,  True],
       dtype=bool)
>>> for e in err[0]: print e # doctest: +NORMALIZE_WHITESPACE
("value for variable 'VARNAME' at level 'LEVNAME' not found",
array([0, 2]))
>>> err[1]
array([[0, 0, 0, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 3, 0, 0],
       [0, 0, 4, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 1, 2, 0, 0],
       [0, 1, 3, 0, 0],
       [0, 1, 4, 0, 0]])
>>> err[2]
set([0, 2])

```

Get values with empty tind

```

>>> etind = numpy.zeros((0,0), dtype=int)
>>> vals, err = h.get_value(etind, 0)

```

### 37.1.9 def get\_exists(self, tind, attrs):

Get attribute value(s) from the data matrix. If the target object has a value, return True(s) and return False(s) for those which are missing a value

```
>>> exts = h.get_exists(tind, [0,1])
>>> exts # doctest: +NORMALIZE_WHITESPACE
array([[False,  True, False,  True,  True,  True,  True,  True],
       [False,  True,  True,  True,  True,  True,  True,  True]],
      dtype=bool)
```

### 37.1.10 def set\_date(self, tind, dates, set\_init\_date=False):

Store date(s) into the date matrix and if set\_init\_date is True, set the initial dates also

```
>>> import datetime
>>> tind = numpy.array([[0,0,0],[0,0,1]], dtype=int)

>>> h.set_date(tind, numpy.array([datetime.date(2000,1,1),
...                               datetime.date(2002,2,2)], dtype=datetime.date))
>>> h.matrix.dmx[:] # doctest: +NORMALIZE_WHITESPACE
array([2000-01-01, 2002-02-02, 0001-01-01, 0001-01-01, 0001-01-01],
      dtype=object)
>>> tind = numpy.array([[0,2,3],[0,3,4]], dtype=int)
>>> dates = numpy.array([datetime.date(2003,3,3),
...                       datetime.date(2005,5,5)], dtype=datetime.date)

>>> h.set_date(tind, dates)
>>> h.matrix.dmx[:] # doctest: +NORMALIZE_WHITESPACE
array([2000-01-01, 2002-02-02, 0001-01-01, 2003-03-03, 2005-05-05],
      dtype=object)
>>> h.matrix.init_dmx[:] # doctest: +NORMALIZE_WHITESPACE
array([0001-01-01, 0001-01-01, 0001-01-01, 0001-01-01, 0001-01-01],
      dtype=object)

>>> h.set_date(tind, dates, True)
>>> h.matrix.init_dmx[:] # doctest: +NORMALIZE_WHITESPACE
array([0001-01-01, 0001-01-01, 0001-01-01, 2003-03-03, 2005-05-05],
      dtype=object)
```

### 37.1.11 def get\_date(self, tind):

Get date(s) from the date matrix:

```
>>> tind = numpy.array([[0,0,0],[0,1,0],[0,2,0]], dtype=int)
>>> h.get_date(tind)
array([2000-01-01, 2000-01-01, 2000-01-01], dtype=object)

>>> tind = numpy.array([[0,0,0],[0,0,1],[0,0,2],[0,0,3]], dtype=int)
>>> h.get_date(tind)
array([2000-01-01, 2002-02-02, 0001-01-01, 2003-03-03], dtype=object)
```

### 37.1.12 def get\_init\_date(self, tind):

```
>>> h.get_init_date(tind)
array([0001-01-01, 0001-01-01, 0001-01-01, 2003-03-03], dtype=object)

>>> tind = numpy.array([[0,0,0],[0,0,1],[0,0,2],[0,0,3]], dtype=int)
>>> h.get_init_date(tind)
array([0001-01-01, 0001-01-01, 0001-01-01, 2003-03-03], dtype=object)
```



Get initial date(s) from the date matrix

### 37.1.13 def get\_id(self, level, tind):

Get id of objects defined in target index:

```
>>> tind = numpy.array([[0,0,2,0,0]], dtype=int)
>>> h.get_id(1, tind)
['SIMULATION-4']
```

### 37.1.14 def get\_parent(self, level, tind, parentlevel, filterby=None):

Get parent target index for objects in the target index

```
>>> tind = numpy.array([[0,0,0,0,0],
...                     [0,0,1,0,0],
...                     [0,0,2,0,0]], dtype=int)
>>> h.get_parent(2, tind, 1)
(array([[0, 0, 4, 0, 0],
       [0, 0, 4, 0, 0]]), set([0]))

>>> h.get_parent(2, tind[(1,):, :], 1)
(array([[0, 0, 4, 0, 0]]), set([]))

>>> h.get_parent(2, tind, 0)
(array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]]), set([]))

>>> tind = numpy.array([[0,0,0,0,0],
...                     [0,0,2,0,0],
...                     [0,0,3,0,0],
...                     [0,0,4,0,0]], dtype=int)
>>> h.get_parent(1, tind, 0)
(array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]]), set([]))
```

### 37.1.15 def get\_children(self, iteration, branch, obj, level, childlevel):

Get the children of an object from child level in given iteration and branch

```
>>> h.get_children(0, 1, 0, 0, 1)
[0, 2, 3, 4]
>>> h.get_children(0, 1, 0, 0, 2)
[0, 1, 2]
>>> h.get_children(0, 1, 3, 1, 2)
[2]
>>> h.get_children(0, 1, 4, 1, 2)
[0, 1]
```

### 37.1.16 def get\_object\_map(self, level, tind):

Get object mapping for targets on given level

```
>>> h.get_object_map(2, tind)
array([ True,  True, False, False], dtype=bool)
```

### 37.1.17 def block(self, level, depth, tind, block\_children=False):

Block evaluation of objects at given model chain depth level and all levels below the given depth level (prevents evaluation of blocked objects):

```
>>> tind = numpy.array([[0,0,0,0,0]], dtype=int)
>>> h.matrix.emap[0,0,0,:,:)
array([[ True, False, False, False, False]], dtype=bool)
>>> h.block(0, 0, tind)
>>> h.matrix.emap[0,0,0,:,:)
array([[False, False, False, False, False]], dtype=bool)
>>> tind = numpy.array([[0,0,3,0,0],[0,0,4,0,0]], dtype=int)
>>> h.matrix.emap[0,0,1,:,:)
array([[ True, False, True,  True,  True]], dtype=bool)
>>> h.block(1, 0, tind)
>>> h.matrix.emap[0,0,1,:,:)
array([[ True, False,  True, False, False]], dtype=bool)
>>> h.matrix.emap[0,0,2,:,:)
array([[ True,  True,  True, False, False]], dtype=bool)
>>> h.block(1, 0, tind, True)
>>> h.matrix.emap[0,0,1,:,:)
array([[ True, False,  True, False, False]], dtype=bool)
>>> h.matrix.emap[0,0,2,:,:)
array([[ True, False, False, False, False]], dtype=bool)
>>> h.block(0, 0, numpy.array([[0,0,0,0,0]], dtype=int), True)
>>> h.matrix.emap[0,0,1,:,:)
array([[False, False, False, False, False]],
<BLANKLINE>
      [[False, False, False, False, False]], dtype=bool)
```

### 37.1.18 def release(self, level, depth, tind, release\_children=False release\_all=False):

Release objects at given level and model chain depth for evaluation. If level is not defined, release all levels. If depth is not defined release objects from all depths and if target index is not defined, release all objects.:

```
>>> h.matrix.omap[0,0,0,:])
array([ True, False, False, False, False], dtype=bool)
>>> h.release(0, 0, None)
>>> h.matrix.emap[0,0,0,:,:)
array([[ True, False, False, False, False]], dtype=bool)
>>> tind = numpy.array([[0,0,2,0,0]], dtype=int)
>>> h.release(1, 0, tind)
>>> h.matrix.emap[0,0,1,:,:)
array([[False, False,  True, False, False]], dtype=bool)
>>> tind = numpy.array([[0,0,4,0,0]], dtype=int)
>>> h.release(1, 0, tind, True)
>>> h.matrix.emap[0,0,1,:,:)
array([[False, False,  True, False,  True]],
<BLANKLINE>
      [[False,  True,  True, False, False]], dtype=bool)
>>> h.release(None, None, None, False, True)
>>> h.matrix.emap[0,0,1,:,:)
array([[ True, False,  True,  True,  True]],
<BLANKLINE>
```

```

[[ True,  True,  True, False, False]], dtype=bool)
>>> h.matrix.omap[0,0,1:,:]
array([[ True, False,  True,  True,  True],
       [ True,  True,  True, False, False]], dtype=bool)

```

### 37.1.19 def block\_children(self, depth, level, childlevel):

Find the blocked objects from 'level' and block all relatives from lower levels until childlevel

```

>>> h.block(1, 0, numpy.array([[0,0,4,0,0]], dtype=int))
>>> h.matrix.emap[0,0,2,0,:]
array([ True,  True,  True, False, False], dtype=bool)
>>> h.block_children(0, 1, 2)
defaultdict(<type 'list'>, {2: array([[0, 0, 1],
      [0, 0, 2]])})
>>> h.matrix.emap[0,0,2,0,:]
array([ True, False, False, False, False], dtype=bool)

```

Block simulation level and the all it's children

```

>>> h.block(0, 0, numpy.array([[0,0,0,0,0]], dtype=int))
>>> bl = h.block_children(0, 0, 2)
>>> bl[1]
array([[0, 0, 0],
       [0, 0, 2],
       [0, 0, 3]])
>>> bl[2]
array([[0, 0, 0]])

```

### 37.1.20 def del\_branch(self, tind):

Delete branches defined in the target index:

```

>>> tind = numpy.array([[0,1,4,0,0]], dtype=int)
>>> links = h.linkage.links[(0,1)].keys()
>>> links.sort()
>>> links
[(0, 0), (1, 0), (1, 2), (1, 3), (1, 4), (2, 0), (2, 1), (2, 2)]
>>> h.linkage.links[(0,1)][(2,0)].keys()
[0, 1]
>>> h.linkage.links[(0,1)][(2,1)].keys()
[0, 1]
>>> ids = h.ind2id.ids[(0,1)].keys()
>>> ids.sort()
>>> ids
[(0, 0), (1, 0), (1, 2), (1, 3), (1, 4), (2, 0), (2, 1), (2, 2)]
>>> h.del_branch(tind)
>>> links = h.linkage.links[(0,1)].keys()
>>> links.sort()
>>> links
[(0, 0), (1, 0), (1, 2), (1, 3), (2, 2)]
>>> ids = h.ind2id.ids[(0,1)].keys()
>>> ids.sort()
>>> ids
[(0, 0), (1, 0), (1, 2), (1, 3), (2, 2)]

```

**37.1.21 def \_squeeze\_objects(self, iteration, branch, level, num):**

Find empty space for number of objects in given iteration, branch and level. Increase matrix size to make space if needed.

**37.1.22 def \_copy\_matrices(self, iteration, frombranch, tobranch, level, fromobj, toobj):**

Copy values in all matrices from parent branch/object to target branch/object

**37.1.23 def \_find\_empty\_branches(self, iteration, level, objs):**

Find empty branches for objects on given level. Resize matrices in branch dimension if necessary.:

```
>>> h.matrix.omap[0,:4,1,:]
array([[ True, False,  True,  True,  True],
       [ True, False,  True,  True, False],
       [False, False, False, False, False],
       [False, False, False, False, False]], dtype=bool)
>>> h._find_empty_branches([0,0,0,0,0], 1, [0,1,2,3,4])
array([2, 0, 2, 2, 1])
```

Find empty branches when object list includes non-unique indices (new branches for one object with different parent branches):

```
>>> h._find_empty_branches([0,0,0,0], 1, [0,4,4,4])
array([2, 1, 2, 3])
```

Find empty branches with non-unique objects and so that matrix dimensions have to be modified:

```
>>> h.matrix.omap[0,:,1,0] = True
>>> h._find_empty_branches([0,0,0], 1, [0,0,0])
array([11, 12, 13])
```

# IND2ID.PY

## 38.1 class Ind2Id(object):

Object id handler for data matrix.

### 38.1.1 def \_\_init\_\_(self):

Initialize an id handler.

```
>>> from simo.matrix.ind2id import Ind2Id
>>> ii = Ind2Id()
>>> ii.ids
defaultdict(<type 'dict'>, {})
```

### 38.1.2 def add\_id(self, iteration, branch, level, ind, oid, parlev=None, parind=None):

Add id for a new object:

```
>>> ii.add_id(0, 0, 0, 0, 'SIMULATION')
>>> dict(ii.ids)
{(0, 0): {(0, 0): 'SIMULATION'}}
>>> dict(ii.inds)
{(0, 0): {(0, 'SIMULATION'): 0}}
>>> ii.add_id(0, 0, 1, 0, 'UID000')
>>> ii.ids[(0, 0)][(1, 0)]
'UID000'
>>> ii.add_id(0, 0, 1, 1, 'UID001')
>>> ii.ids[(0, 0)][(1, 1)]
'UID001'
>>> ii.add_id(0, 0, 1, 2, 'UID002')
>>> ii.ids[(0, 0)][(1, 2)]
'UID002'
>>> ii.add_id(0, 0, 2, 0, None, 1, 0)
>>> ii.ids[(0, 0)][(2, 0)]
'UID000-0'
>>> ii.add_id(0, 0, 2, 1, None, 1, 1)
>>> ii.ids[(0, 0)][(2, 1)]
'UID001-0'
>>> ii.add_id(0, 0, 2, 2, None, 1, 1)
>>> ii.add_id(0, 0, 3, 0, None, 2, 2)
>>> ii.ids[(0, 0)][(3, 0)]
'UID001-1-0'
>>> ii.inds[(0, 0)][(3, 'UID001-1-0')]
```

```
0
>>> ii.indxs[(0, 0)][(2, 'UID001-1')]
2
```

### 38.1.3 def get\_id(self, iteration, branch, level, oind):

Get object id:

```
>>> ii.get_id(0, 0, 1, 0)
'UID000'
>>> ii.get_id(0, 0, 1, 1)
'UID001'
>>> ii.get_id(0, 0, 2, 1)
'UID001-0'
```

### 38.1.4 def get\_ind(self, iteration, branch, level, oid):

Get object index:

```
>>> ii.get_ind(0, 0, 1, 'UID000')
0
```

### 38.1.5 def remove\_id(self, iteration, branch, level, oind):

```
>>> ii.remove_id(0, 0, 1, 0)
>>> ii.get_id(0, 0, 1, 0)
>>> ii.get_ind(0, 0, 1, 0)
```

# LINKAGE.PY

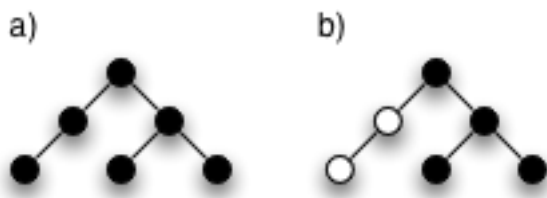
## 39.1 class Linkage(object):

The complete object linking for the data matrix. For each object linking to all other objects in the data hierarchy lineage is stored as a set of object indices by data level.

### 39.1.1 def \_\_init\_\_(self, hierarchy):

Attributes:

- **hierarchy:** a dictionary with level indices as keys. The data is a dictionary with 'ordinal' and 'lineage' as keys.
  - ordinal contains the ordinal number of the data level in the data hierarchy starting from 0 for the root data level.
  - lineage contains a set of lineage identifiers for the data level:



In the figure there is only a single lineage in the a) case as there is only one sub level for each data level. In the b) case the root level has two different sub levels and hence there are two lineages in the data hierarchy. The black nodes belong to the lineage 0 and the white nodes belong to the lineage 1. However, the root node is an exception to this rule as it belongs to both lineages 0 and 1 being the parent level for both.

- **links:** {(iteration, branch):{(level index, object index):<link dict>}}
- link dict: level indices as key and a set of object indices as data. The set contains the object indices to which the object identified by the combination key of (simulation unit, branch)(level index, object index) is connected to

```
>>> hierarchy = {0: {'ordinal':0, 'lineage':set([0, 1])},
...               1: {'ordinal':1, 'lineage':set([0])},
...               2: {'ordinal':1, 'lineage':set([1])},
...               3: {'ordinal':2, 'lineage':set([0])},
...               4: {'ordinal':2, 'lineage':set([1])}}
>>> from simo.matrix.linkage import Linkage
>>> l = Linkage()
```

### 39.1.2 def add\_object(self, hierarchy, iteration, branch, objkey, parentkey, update\_existing=True):

Add linking information for a new object and all nodes connected to it. Test data here is for the b-case in the figure; i.e., five different data levels, level 0 being the root level having two child levels 1 and 2. Level 3 is the child of level 1 and level 4 the child of level 2. In the data structure the first key indicates the iteration and branch, and the second tuple the data level and object index, while the third is the data level and object index of the parent node:

```
>>> data = [(0, (0, 0), (None, None)), (0, (1, 0), (0, 0)),
...         (0, (2, 0), (0, 0)), (0, (3, 0), (1, 0)),
...         (0, (4, 1), (2, 0)), (0, (4, 2), (2, 0)),
...         (0, (4, 0), (2, 0))]
>>> iteration = 0
>>> for node in data:
...     branch, objkey, parentkey = node
...     l.add_object(hierarchy, iteration, branch, objkey, parentkey)
>>> l.links[(0, 0)][(0, 0)]
{1: [0], 2: [0], 3: [0], 4: [0, 1, 2]}
>>> l.links[(0, 0)][(1, 0)]
{0: [0], 3: [0]}
>>> l.links[(0, 0)][(2, 0)]
{0: [0], 4: [0, 1, 2]}
>>> l.links[(0, 0)][(3, 0)]
{0: [0], 1: [0]}
>>> l.links[(0, 0)][(4, 0)]
{0: [0], 2: [0]}
>>> l.links[(0, 0)][(4, 1)]
{0: [0], 2: [0]}
>>> l.links[(0, 0)][(4, 2)]
{0: [0], 2: [0]}
```

Construct another linkage for testing adding object relations between existing objects.

```
>>> l2 = Linkage()
>>> data = [(0, (0, 0), (None, None)), (0, (2, 0), (0, 0)),
...         (0, (2, 1), (0, 0)), (0, (2, 2), (0, 0)),
...         (0, (1, 0), (0, 0)), (0, (1, 1), (0, 0)),]
>>> for node in data:
...     branch, objkey, parentkey = node
...     l2.add_object(hierarchy, 0, 0, objkey, parentkey)
>>> l2.links[(0, 0)][(0, 0)]
{1: [0, 1], 2: [0, 1, 2]}
>>> l2.links[(0, 0)][(2, 0)]
{0: [0]}
```

Add objects without updating existing linkage info. This is needed when adding object relations between existing objects instead of new objects (when for example creating new objects by grouping existing objects):

```
>>> l2.add_object(hierarchy, 0, 0, (2, 0), (1, 0), False)
>>> l2.add_object(hierarchy, 0, 0, (2, 1), (1, 0), False)
>>> l2.add_object(hierarchy, 0, 0, (2, 2), (1, 1), False)
>>> l2.links[(0, 0)][(0, 0)]
{1: [0, 1], 2: [0, 1, 2]}
>>> l2.links[(0, 0)][(1, 0)]
{0: [0], 2: [0, 1]}
>>> l2.links[(0, 0)][(1, 1)]
{0: [0], 2: [2]}
>>> l2.links[(0, 0)][(2, 0)]
{0: [0], 1: [0]}
>>> l2.links[(0, 0)][(2, 1)]
```



```
{0: [0], 1: [0]}
>>> l2.links[(0, 0)][(2, 2)]
{0: [0], 1: [1]}
```

### 39.1.3 def remove\_object(self, simobj, obj, children=False):

Remove the references to the object from all the connected nodes, and the object from the links structure:

```
>>> l.links[(0, 0)][(1, 0)]
{0: [0], 3: [0]}
>>> l.links[(0, 0)][(2, 0)]
{0: [0], 4: [0, 1, 2]}
>>> l.links[(0, 0)][(3, 0)]
{0: [0], 1: [0]}
>>> l.links[(0, 0)][(4, 0)]
{0: [0], 2: [0]}
>>> l.remove_object(0, 0, (4, 0))
{4: [0]}
>>> l.links[(0, 0)][(2, 0)]
{0: [0], 4: [1, 2]}
```

Child removal can be forced with children-parameter:

```
>>> l.remove_object(0, 0, (2, 0), children=True)
{2: [0], 4: [1, 2]}
>>> l.links[(0, 0)][(0, 0)]
{1: [0], 3: [0]}
```



# MATRIX.PY

```
>>> from minimock import Mock
```

## 40.1 class Matrix(object):

Five dimensional Numpy array that acts as the simulation data core, stored as mx attribute.

Dimensions:

- 1. dimension: iteration
- 1. dimension: branch
- 1. dimension: level
- 1. dimension: object
- 1. dimension: attribute

The data matrix is accompanied with three auxiliary matrices: omap, emap and dmx.

omap (object map) is a four dimensional boolean Numpy array, having dimensions:

- 1. dimension: iteration
- 1. dimension: branch
- 1. dimension: level
- 1. dimension: object

The omap values indicate the object existence for the particular index combination in the data matrix; i.e., if omap[0, 0, 1, 3] has the value of False, the values at mx[0, 0, 1, 3, :] are not data but regarded as random data. If omap[0, 0, 1, 3] has the value of True, however, the values at mx[0, 0, 1, 3, :] are the attribute values for the object 3 at data level 1 at data branch 0 and iteration 0.

emap (evaluation map) is a five dimensional boolean Numpy array, having dimensions:

- 1. dimension: iteration
- 1. dimension: branch
- 1. dimension: level
- 1. dimension: model chain depth
- 1. dimension: object

The emap values indicate the current model chain task evaluation status at the given model chain depth.

dmx (date matrix) contains current date for each simulation unit. Date matrix is a one dimensional datetime array:

- 1. dimension: object

`init_dmx` (initial date matrix) contains simulation starting date for each simulation unit. Initial date matrix has same shape as date matrix

#### 40.1.1 `def __init__(self, levels, attributes, mcdepth, objcount):`

Initialize a data matrix:

```
>>> from simo.matrix.matrix import Matrix
>>> sim = Mock('simulator')
>>> d = Matrix(1, 2, 10, 5, sim, 0)
>>> d.shape
(1, 1, 2, 0, 10)
>>> d.branches
1
>>> d.objects
0
>>> d.omap
array([], shape=(1, 1, 2, 0), dtype=bool)
>>> d.emap
array([], shape=(1, 1, 2, 5, 0), dtype=bool)
>>> d.dmx
array([], dtype=object)
>>> d.init_dmx
array([], dtype=object)
>>> d.errormx
array([], shape=(1, 1, 0), dtype=int32)
```

#### 40.1.2 `def reset(self):`

Reset all matrix values:

```
>>> d.reset()
```

#### 40.1.3 `def modify_matrix(self, axis, inc):`

Modify data matrix by increasing array size along given axis:

```
>>> d.modify_matrix(0, 1)
Called simulator.add_error(
    'Bad axis (expected 1 or 3, got 0), the size of the matrix can not be modified.',
    None,
    None)
False
>>> d.modify_matrix(1, 1)
True
>>> d.branches
2
>>> d.objects
0
>>> d.shape
(1, 2, 2, 0, 10)
>>> d.modify_matrix(3, 5)
True
>>> d.branches
2
>>> d.objects
5
>>> d.shape
```

```
(1, 2, 2, 5, 10)
>>> d.modify_matrix(3, 5)
True
>>> d.branches
2
>>> d.objects
10
>>> d.shape
(1, 2, 2, 10, 10)
>>> d.modify_matrix(1, 2)
True
>>> d.branches
4
>>> d.objects
10
>>> d.shape
(1, 4, 2, 10, 10)
>>> d.omap.shape
(1, 4, 2, 10)
>>> d.emap.shape
(1, 4, 2, 5, 10)
>>> d.omap[0, 3, 1, 9]
False
>>> d.emap.shape
(1, 4, 2, 5, 10)
>>> d.dmx.shape
(10,)
>>> d.branches
4
>>> d.objects
10
>>> d.shape
(1, 4, 2, 10, 10)
>>> d.dmx.shape
(10,)
>>> d.init_dmx.shape
(10,)
```

Optimization modules:



# HERO.PY

## 41.1 class Hero(Optimizer):

SIMO metaheuristic HERO optimization algorithm.

Parameters:

- ...

### 41.1.1 def \_\_init\_\_(self, logger, logname, optlogger, taskdef, simdbin, opdbin, simdbout, opdbout, \*\*keywords):

Initialize optimizer:

```
>>> epsilon = 0.00001
>>> from simo.optimization.hero import HERO
>>> execfile('optimization/test/mocks4optimizer.py')
>>> chart_path = 'optimization/test'

>>> hero = HERO(logger, logname, taskdef, simdbin, simdbout, chart_path,
...             keyword1='test-kw')
Called Logger.log_message(
    'optimization-test',
    'ERROR',
    'Parameter "maxiterations" missing!')

>>> hero = HERO(logger, logname, taskdef, simdbin, simdbout, chart_path,
...             maxiterations=1)
```

### 41.1.2 def optimize(self):

Run HERO optimization algorithm

```
>>> hero._stat_logger = ologger # replace stat logger with a mock
>>> hero._data = omatrix1 # replace data handler with a mock
>>> hero.optimize() # doctest: +NORMALIZE_WHITESPACE
Called Logger.log_message(
    'optimization-test',
    'ERROR',
    'Cannot generate random solution: optimization data was not
    constructed successfully')
>>> hero._data = omatrix2 # replace data handler with another mock
>>> hero.set_data() # doctest: +ELLIPSIS
Called OMatrix.construct_data(<Mock ... SimInputDB>)
True
```

```
>>> hero.optimize() # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
Called OMatrix.get_random_solution(0)
Called OMatrix.solution_feasibility(array([0, 0, 0, 0, 0]), 0, first_seek=True)
Called OMatrix.solution_utility(array([0, 0, 0, 0, 0]), 0)
Called OLogger.start(0, array([[2, 5, 3, 4, 1],
                               [2, 5, 3, 4, 1]]))
Called OMatrix.solution_utility(array([1, 0, 0, 0, 0]), 0)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_feasibility(array([1, 0, 0, 0, 0]), 0)
Called OLogger.add_round(0, 0, array([1, 0, 0, 0, 0]), 1.0)
Called OMatrix.solution_utility(array([1, 1, 0, 0, 0]), 0)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_utility(array([1, 2, 0, 0, 0]), 0)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_utility(array([1, 3, 0, 0, 0]), 0)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_utility(array([1, 4, 0, 0, 0]), 0)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_feasibility(array([1, 4, 0, 0, 0]), 0)
Called OLogger.add_round(0, 1, array([1, 4, 0, 0, 0]), 1.0)
Called OMatrix.solution_utility(array([1, 4, 1, 0, 0]), 0)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_utility(array([1, 4, 2, 0, 0]), 0)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_feasibility(array([1, 4, 2, 0, 0]), 0)
Called OLogger.add_round(0, 2, array([1, 4, 2, 0, 0]), 1.0)
Called OMatrix.solution_utility(array([1, 4, 2, 1, 0]), 0)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_utility(array([1, 4, 2, 2, 0]), 0)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_utility(array([1, 4, 2, 3, 0]), 0)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_feasibility(array([1, 4, 2, 3, 0]), 0)
Called OLogger.add_round(0, 3, array([1, 4, 2, 3, 0]), 1.0)
Called OMatrix.solution_feasibility(array([1, 4, 2, 3, 0]), 0)
Called OLogger.add_round(0, 4, array([1, 4, 2, 3, 0]), 1.0)
Called OLogger.stop(0)
Called OMatrix.get_random_solution(1)
Called OMatrix.solution_feasibility(array([0, 0, 0, 0, 0]), 1, first_seek=True)
Called OMatrix.solution_utility(array([0, 0, 0, 0, 0]), 1)
Called OLogger.start(1, array([[2, 5, 3, 4, 1],
                               [2, 5, 3, 4, 1]]))
Called OMatrix.solution_utility(array([1, 0, 0, 0, 0]), 1)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_feasibility(array([1, 0, 0, 0, 0]), 1)
Called OLogger.add_round(1, 0, array([1, 0, 0, 0, 0]), 1.0)
Called OMatrix.solution_utility(array([1, 1, 0, 0, 0]), 1)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_utility(array([1, 2, 0, 0, 0]), 1)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_utility(array([1, 3, 0, 0, 0]), 1)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_utility(array([1, 4, 0, 0, 0]), 1)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_feasibility(array([1, 4, 0, 0, 0]), 1)
Called OLogger.add_round(1, 1, array([1, 4, 0, 0, 0]), 1.0)
Called OMatrix.solution_utility(array([1, 4, 1, 0, 0]), 1)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_utility(array([1, 4, 2, 0, 0]), 1)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_feasibility(array([1, 4, 2, 0, 0]), 1)
Called OLogger.add_round(1, 2, array([1, 4, 2, 0, 0]), 1.0)
```



```
Called OMatrix.solution_utility(array([1, 4, 2, 1, 0]), 1)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_utility(array([1, 4, 2, 2, 0]), 1)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_utility(array([1, 4, 2, 3, 0]), 1)
Called OMatrix.compare_utilities(1.0, 1.0)
Called OMatrix.solution_feasibility(array([1, 4, 2, 3, 0]), 1)
Called OLogger.add_round(1, 3, array([1, 4, 2, 3, 0]), 1.0)
Called OMatrix.solution_feasibility(array([1, 4, 2, 3, 0]), 1)
Called OLogger.add_round(1, 4, array([1, 4, 2, 3, 0]), 1.0)
Called OLogger.stop(1)
Called Logger.log_message(
    'optimization-test',
    'INFO',
    'Writing optimized treatment schedule to result database...')
Called SimInputDB.copy_to_db(
    [(0, 'UNIT1', 1), (0, 'UNIT2', 4), (0, 'UNIT3', 2), (0, 'UNIT4', 3), (0, 'UNIT5', 0), (1, 'UNIT6', 0)],
    'comp_unit',
    <Mock ... SimOutputDB>)
Called Logger.log_message(
    'optimization-test',
    'INFO',
    'Writing completed in ... min')
```



# JLP.PY

## 42.1 class JLP(Optimizer):

Interface between SIMO optimizer and JLP linear programming library.

Parameters:

- ...

### 42.1.1 def \_\_init\_\_(self, logger, logname, optlogger, taskdef, simdbin, opdbin, simdbout, opdbout, \*\*keywords):

Initialize optimizer:

```
>>> epsilon = 0.00001
>>> from simo.optimization.jlp import JLP
>>> execfile('optimization/test/mocks4optimizer.py')
>>> chart_path = 'optimization/test'

>>> jlp = JLP(logger, logname, taskdef, simdbin, simdbout, chart_path)
Called Logger.log_message(
    'optimization-test',
    'ERROR',
    "Parameter 'maxobjects' missing!")
Called Logger.log_message(
    'optimization-test',
    'ERROR',
    "Parameter 'jfolder' missing!")
Called Logger.log_message(
    'optimization-test',
    'ERROR',
    "Parameter 'prefix' missing!")

>>> jlp = JLP(logger, logname, taskdef, simdbin, simdbout, chart_path,
...           maxobjects=1000, jfolder='', prefix='test')
```

### 42.1.2 def \_process\_branch(self, unitsol, weights):

Process unit split by the JLP algorithm

### 42.1.3 def \_process\_infeasible(self):

Process JLP output when the problem is infeasible

#### 42.1.4 def \_copy\_result\_file(self, respath, iteration):

Parameters:

respath – result file path, str iteration – iteration indice, int

Copy JLP result file to result folder

#### 42.1.5 def \_read\_solution(self, iteration):

Parameters:

iteration – iteration indice, int

Read and process solution from JLP output files

#### 42.1.6 def \_remove\_j\_files(self):

Remove J input and output files

#### 42.1.7 def \_update\_stack(self, stack):

Parameters:

stack – postfix stack, list

Update infix stack:

```
>>> stack = [['a'], ['b']]
>>> jlp._update_stack(stack)
>>> print stack
[['a', 'b']]
```

#### 42.1.8 def \_process\_const\_expr(self, num\_of\_expr, expressions):

Parameters:

num\_of\_expr – number of expressions as integer expressions – SIMO postfix expressions in a list

Process constraint postfix expression back to infix form and try to figure out if the expression is good for JLP:

```
>>> coinfix, covars, coorder = jlp._process_const_expr(\
...     2, taskdef.constraints)
>>> print coinfix
['V2010_2010c0 - V2000_2000c1 > 0.00', 'Volume2020_endc0 - Volume2010_endc1 > 0.00']
>>> print covars
['V2010_2010c0', 'V2000_2000c1', 'Volume2020_endc0', 'Volume2010_endc1']
>>> print coorder
[0, 1, 0, 1]
```

Process invalid constraint postfix expression:

```
>>> jlp._init_ok = True
>>> inv_coinfix, inv_covars, inv_coorder = jlp._process_const_expr(\
...     2, invalid_taskdef.constraints)
Called Logger.log_message(
    'optimization-test',
    'ERROR',
    "Invalid operator for J '/'!")
```

```

Called Logger.log_message(
    'optimization-test',
    'ERROR',
    'JLP condition right hand side must be a numerical value!')
>>> print inv_coinfix
['V2010_endc0 / V2000_endc1 > 1.05', 'Volume2020_endc0 > Volume2010_endc1']
>>> print inv_covars
['V2010_endc0', 'V2000_endc1', 'Volume2020_endc0']
>>> print inv_coorder
[0, 1, 0]
>>> print jlp._init_ok
False

```

#### 42.1.9 def \_write\_j\_par(self, partype):

Parameters:

partype – parameter type, either 'init' or 'close'

Write J parameter file (j.par):

```

>>> jlp._write_j_par('init')
>>> fp = open('j.par', 'r')
>>> contents = fp.read()
>>> print contents
!1000
;incl('test_problem.txt')
>>> fp.close()

```

#### 42.1.10 def \_write\_j\_problem(self):

Write the optimisation problem definition to a file for J:

```

>>> jlp._obj_vars = ['x0', 'x1']
>>> jlp._const_infix = coinfix
>>> jlp._const_vars = covars
>>> jlp._const_var_order = coorder
>>> jlp._write_j_problem()
>>> fp = open('test_problem.txt', 'r')
>>> contents = fp.read()
>>> print contents # doctest: +ELLIPSIS
* READ OBJECTIVE AND CONSTRAINT VARIABLE VALUES FROM FILE *
udat=data(read->(id,nbranch),in->'test_unitdata.txt')
sdat=data(read->(x0,x1,V2010_2010c0,V2000_2000c1,Volume2020_endc0,Volume2010_endc1), in->'test_va
linkdata(data->udat,subdata->sdat,nobs->nbranch)
<BLANKLINE>
* OPTIMISATION PROBLEM DEFINITION *
pr=problem()
<BLANKLINE>
* OBJECTIVE FUNCTION DEFINITION *
0.3*x0+0.7*x1==max
<BLANKLINE>
* CONSTRAINT DEFINITIONS *
V2010_2010c0 - V2000_2000c1 > 0.00
Volume2020_endc0 - Volume2010_endc1 > 0.00
/
jlp(problem->pr,data->udat,report->'...jlp_output.txt')
branch_report=trans()
do(i,1,weights())

```

```
write('test_bresult.txt', $, matrix%udat(unit(i), 1), schedw(i), price%unit(unit(i)))
enddo
/
weight_report=trans()
do(i, 1, partweights())
write('test_wresult.txt', $, partunit(i), partschedw(i), partweight(i))
enddo
/
; if(Feasible); then
call(branch_report)
call(weight_report)
; else
write('test_bresult.txt', '~infeasible problem~')
; endif
end
>>> fp.close()
```

#### 42.1.11 def \_write\_j\_data(self, iteration):

Parameters:

iteration – iteration indice, int

Write the optimisation data to two data files for J. The unit data contains the simulation unit id and number of simulated alternatives for each unit. The variable contains the objective and constraint variable values for each alternative in each simulation unit:

```
>>> omatrix2.task_def = taskdef
>>> jlp._data = omatrix2
>>> jlp._write_j_data(0)
>>> fp = open('test_unitdata.txt', 'r')
>>> print fp.read()
1,2
3,3
2,5
5,1
4,4
<BLANKLINE>
>>> fp.close()
>>> fp = open('test_vardata.txt', 'r')
>>> print fp.read()
0.0,0.1,0.0,0.0,0.2,0.2
1.0,1.1,0.0,0.0,0.2,0.2
2.0,2.1,4.0,4.0,4.2,4.2
3.0,3.1,4.0,4.0,4.2,4.2
4.0,4.1,4.0,4.0,4.2,4.2
1.0,1.1,2.0,2.0,2.2,2.2
2.0,2.1,2.0,2.0,2.2,2.2
3.0,3.1,2.0,2.0,2.2,2.2
4.0,4.1,2.0,2.0,2.2,2.2
5.0,5.1,2.0,2.0,2.2,2.2
4.0,4.1,8.0,8.0,8.2,8.2
3.0,3.1,6.0,6.0,6.2,6.2
4.0,4.1,6.0,6.0,6.2,6.2
5.0,5.1,6.0,6.0,6.2,6.2
6.0,6.1,6.0,6.0,6.2,6.2
<BLANKLINE>
>>> fp.close()
```

**42.1.12 def optimize(self):**

Construct J input files, run J program as a subprocess and process J output





# OPTIMIZER.PY

Super class for optimization algorithms

## 43.1 class Optimizer(object):

Parent class for all heuristic optimization algorithms.

Parameters:

- ...

### 43.1.1 def \_\_init\_\_(self, logger, logname, optlogger, taskdef, simdbin, opdbin, simdbout, opdbout, \*\*keywords):

Initialize optimizer:

```
>>> epsilon = 0.00001
>>> from simo.optimization.optimizer import Optimizer
>>> execfile('optimization/test/mocks4optimizer.py')
>>> chart_path = 'optimization/test'
>>> opt = Optimizer(logger, logname, taskdef, simdbin, simdbout, chart_path,
...                 keyword1='test-kw')
```

### 43.1.2 def \_add\_error(self, msg):

Log error message:

```
>>> opt._add_error('Test error message') # doctest: +NORMALIZE_WHITESPACE
Called Logger.log_message('optimization-test', 'ERROR',
                          'Test error message')
```

### 43.1.3 def \_add\_info(self, msg):

Log info message

```
>>> opt._add_info('Test info message') # doctest: +NORMALIZE_WHITESPACE
Called Logger.log_message('optimization-test', 'INFO',
                          'Test info message')
```

### 43.1.4 def set\_data(self):

Set optimization data: collect values from data and operation result databases into data structure:

```
>>> opt._data = omatrix1 # replace data handler with a mock
>>> opt.set_data() # doctest: +ELLIPSIS
Called OMatrix.construct_data(<Mock ... SimInputDB>)
Called Logger.log_message(
    'optimization-test',
    'ERROR',
    'Could not construct optimization data matrix')
False
>>> opt._data = omatrix2 # replace data handler with another mock
>>> opt.set_data() # doctest: +ELLIPSIS
Called OMatrix.construct_data(<Mock ... SimInputDB>)
True

>>> opt._solutions
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
```

### 43.1.5 def \_store\_solution(self, iteration):

Store the optimal solution of a single iteration

```
>>> opt._elite_solution[:] = [2, 9, 5, 3, 7]
>>> opt._store_solution(0)
>>> opt._elite_solution[:] = [3, 4, 6, 0, 1]
>>> opt._store_solution(1)
>>> opt._solutions
array([[2, 9, 5, 3, 7],
       [3, 4, 6, 0, 1]])
```

### 43.1.6 def \_store\_results(self):

Store the optimum treatment schedules to result databases

```
>>> opt._store_results() # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
Called Logger.log_message(
    'optimization-test',
    'INFO',
    'Writing optimized treatment schedule to result database...')
Called SimInputDB.copy_to_db(
    [(0, 'UNIT1', 2),
     (0, 'UNIT2', 9),
     (0, 'UNIT3', 5),
     (0, 'UNIT4', 3),
     (0, 'UNIT5', 7),
     (1, 'UNIT1', 3),
     (1, 'UNIT2', 4),
     (1, 'UNIT3', 6),
     (1, 'UNIT4', 0),
     (1, 'UNIT5', 1)],
    'comp_unit',
    <Mock ... SimOutputDB>)
Called Logger.log_message(
    'optimization-test',
    'INFO',
    'Writing completed in ... min')
```

# TABUSEARCH.PY

SIMO metaheuristic TABUSEARCH algorithm

```
>>> epsilon = 0.00001
```

## 44.1 class TabuSearch(Optimizer):

TabuSearch heuristic optimization algorithm implementation. Initialize with keyword arguments eg. maxiterations=1000, entrytenure=10, exittenu=20

### 44.1.1 def \_\_init\_\_(self, logger, logname, optlogger, taskdef, simdbin, opdbin, simdbout, opdbout, \*\*keywords):

Initialize optimizer:

```
>>> from simo.optimization.tabusearch import TabuSearch
>>> #execfile('test/mocks4optimizer.py')
>>> execfile('optimization/test/mocks4optimizer.py')
>>> chart_path = 'optimization/test'

>>> ts = TabuSearch(logger, logname, taskdef, simdbin, simdbout, chart_path,
...                 keywordl='test-kw')
Called Logger.log_message(
    'optimization-test',
    'ERROR',
    'Parameter "maxiterations" missing!')
Called Logger.log_message(
    'optimization-test',
    'ERROR',
    'Parameter "entrytenure" missing!')
Called Logger.log_message(
    'optimization-test',
    'ERROR',
    'Parameter "exittenu" missing!')

>>> ts = TabuSearch(logger, logname, taskdef, simdbin, simdbout, chart_path,
...                 maxiterations=0, entrytenure=0, exittenu=0)
```

### 44.1.2 def optimize(self):

Run TABUSEARCH optimization algorithm

```
>>> ts._stat_logger = ologger # replace stat logger with a mock
>>> ts._data = omatrix1 # replace data handler with a mock
>>> ts.optimize() # doctest: +NORMALIZE_WHITESPACE
Called Logger.log_message(
    'optimization-test',
    'ERROR',
    'Cannot generate random solution: optimization data was not
    constructed successfully')
Called Logger.log_message(
    'optimization-test',
    'ERROR',
    'Cannot generate initial solution for iteration 0, optimization
    of iteration aborted!')
>>> ts._data = omatrix2 # replace data handler with another mock
>>> ts.set_data() # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
Called OMatrix.construct_data(<Mock ... SimInputDB>)
True
>>> ts.optimize() # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
Called OMatrix.get_random_solution(0)
Called OMatrix.solution_feasibility(array([0, 0, 0, 0, 0]), 0,
first_seek=True)
Called OMatrix.solution_utility(array([0, 0, 0, 0, 0]), 0)
Called OLogger.start(0, array([[2, 5, 3, 4, 1],
    [2, 5, 3, 4, 1]]))
Called OLogger.stop(0)
Called OMatrix.get_random_solution(1)
Called OMatrix.solution_feasibility(array([0, 0, 0, 0, 0]), 1,
first_seek=True)
Called OMatrix.solution_utility(array([0, 0, 0, 0, 0]), 1)
Called OLogger.start(1, array([[2, 5, 3, 4, 1],
    [2, 5, 3, 4, 1]]))
Called OLogger.stop(1)
Called Logger.log_message(
    'optimization-test',
    'INFO',
    'Writing optimized treatment schedule to result database...')
Called SimInputDB.copy_to_db(
    [(0, 'UNIT1', 0), (0, 'UNIT2', 0), (0, 'UNIT3', 0), (0, 'UNIT4', 0),
    (0, 'UNIT5', 0), (1, 'UNIT1', 0), (1, 'UNIT2', 0), (1, 'UNIT3', 0),
    (1, 'UNIT4', 0), (1, 'UNIT5', 0)],
    'comp_unit',
    <Mock ... SimOutputDB>)
Called Logger.log_message(
    'optimization-test',
    'INFO',
    'Writing completed in ... min')
```

### 44.1.3 def \_update\_tabu\_set(self, unit, exit, entry):

Add a new taboo move to the set and decrease the tenure counter for existing taboos

```
>>> ts._exit_map
array([[False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False]], dtype=bool)
>>> ts._entry_map
array([[False, False, False, False, False],
       [False, False, False, False, False],
```

```

        [False, False, False, False, False],
        [False, False, False, False, False],
        [False, False, False, False, False]], dtype=bool)
>>> ts._exit_timer
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
>>> ts._entry_timer
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
>>> ts._exit_tenure = 4
>>> ts._entry_tenure = 2
>>> ts._update_tabu_set(0, 0, 1)
>>> ts._update_tabu_set(2, 2, 3)
>>> ts._update_tabu_set(3, 0, 2)
>>> ts._update_tabu_set(4, 1, 2)
>>> ts._exit_map
array([[ True, False, False, False, False],
       [False, False, False, False, False],
       [False, False,  True, False, False],
       [ True, False, False, False, False],
       [False,  True, False, False, False]], dtype=bool)
>>> ts._entry_map
array([[False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False,  True, False, False],
       [False, False,  True, False, False]], dtype=bool)
>>> ts._exit_timer
array([[1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 2, 0, 0],
       [3, 0, 0, 0, 0],
       [0, 4, 0, 0, 0]])
>>> ts._entry_timer
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0]])

```

#### 44.1.4 def \_is\_move\_tabu(self, unit, frombranch, tobranch):

True if taboo, False if not

```

>>> ts._is_move_tabu(0, 0, 1)
True
>>> ts._is_move_tabu(1, 0, 1)
False

```

#### 44.1.5 def \_can\_exit(self, unit, frombranch):

True if exit map value at (unit, frombranch) is False

```
>>> ts._can_exit(0,0)
False
>>> ts._can_exit(0,1)
True
>>> ts._can_exit(2,2)
False
```

#### **44.1.6 def \_get\_neighbourhood(self, unit, frombranch, numofbranches):**

Get non-taboo neighbours of (unit, frombranch)

```
>>> ts._get_neighbourhood(4, 0, 3)
array([False,  True, False, False, False], dtype=bool)
>>> ts._get_neighbourhood(4, 0, 5)
array([False,  True, False,  True,  True], dtype=bool)
```

# OLOGGER.PY

SIMO optimization logger

## 45.1 class OLogger(object):

Class for logging the performance and statistics of SIMO optimization algorithms.

### 45.1.1 def \_\_init\_\_(self, logger, logname, chart\_path):

Initialize optimization logger

```
>>> from minimock import Mock
>>> logger = Mock('Logger')
>>> logname = 'MOCK'
>>> chart_path = 'optimization/test'
>>> from simo.optimization.tools.ologger import OLogger
>>> ol = OLogger(logger, logname, chart_path)
```

### 45.1.2 def start(self, iteration, nbranches):

Start logging of the optimization process

```
>>> import numpy
>>> nbranches = numpy.array([[2, 1, 4, 3], [2, 1, 4, 3]], dtype=int)
>>> ol.start(1, nbranches)
Called Logger.log_message('MOCK', 'INFO', 'Optimizing iteration 1')
Called Logger.log_message('MOCK', 'INFO', 'Total number of units: 4')
Called Logger.log_message(
    'MOCK',
    'INFO',
    'Total number of treatment schedules: 10')
Called Logger.log_message(
    'MOCK',
    'INFO',
    'Average number of treatment schedules per unit: 2.50')
Called Logger.log_message('MOCK', 'INFO', 'Total solution space size: 2.50^4')
Called Logger.log_message('MOCK', 'INFO', 'Optimization process started...')
```

### 45.1.3 def add\_round(self, iteration, iunit, currsol, currval):

Add current iterative round parameters to the logger

```
>>> currsol = numpy.zeros(4, dtype=int)
>>> ol.add_round(3, 0, currsol, 1.0)
>>> ol._best_value
1.0
>>> ol.add_round(3, 0, currsol, 1.5)
>>> ol._best_value
1.5
>>> ol.add_round(3, 0, currsol, 1.2)
>>> ol._best_value
1.5
>>> ol.add_round(3, 0, currsol, 2.0)
>>> ol._best_value
2.0
>>> ol.add_round(3, 0, currsol, 1.6)
>>> ol._best_value
2.0
>>> ol.add_round(3, 0, currsol, 1.8)
>>> ol._best_value
2.0
>>> ol._num_of_rounds
6
>>> ol._curr_values
[1.0, 1.5, 1.2, 2.0, 1.6000000000000001, 1.8]
>>> ol._best_values
[1.0, 1.5, 1.5, 2.0, 2.0, 2.0]
```

#### 45.1.4 def stop(self, iteration):

Stop logging of the optimization process

```
>>> ol.stop(5) # doctest: +ELLIPSIS
Called Logger.log_message(
    'MOCK',
    'INFO',
    'Optimization process performed successfully in ... min')
Called Logger.log_message(
    'MOCK',
    'INFO',
    'Saving optimization log for iteration 5...')
Called Logger.log_message(
    'MOCK',
    'INFO',
    'Saving utility value chart to ...utility-iter-5-set-0-6.png')
```

#### 45.1.5 def \_save\_stats(self, iteration):

Save the statistics of the current optimization process

```
>>> ol._save_stats(5) # doctest: +ELLIPSIS
Called Logger.log_message(
    'MOCK',
    'INFO',
    'Saving optimization log for iteration 5...')
Called Logger.log_message(
    'MOCK',
    'INFO',
    'Saving utility value chart to ...utility-iter-5-set-0-6.png')
```



# OMATRIX.PY

SIMO optimization data matrix:

```
>>> import numpy
>>> import datetime
>>> from minimock import Mock
>>> epsilon = 0.00001
```

## 46.1 def calculate\_NPV(value, discountrate, presentdate, event-date):

Calculate the discounted NPV (Net Present Value) for a single value or vector

```
>>> from simo.optimization.tools.omatrix import calculate_NPV
>>> npv = calculate_NPV(100.0, 5.0,
...                     datetime.date(2000,1,1), datetime.date(2010,1,1))
>>> abs(npv - 61.391325) < epsilon
True
>>> values = numpy.array([100, 100, 100], dtype=float)
>>> npv = calculate_NPV(values, 5.0,
...                     datetime.date(2000,1,1), datetime.date(2010,1,1))
>>> abs(npv - 61.391325) < epsilon
array([ True,  True,  True], dtype=bool)
```

### 46.1.1 class OMatrix(object):

Class for in-memory storage of SIMO optimization data structures. Get the wanted data from input database, stores it into multidimensional numpy arrays for fast access during optimization process.

OMatrix has two matrices, one for subobjective data and one for constraint data: - Subobjective data matrix (dmx) has dimensions: (iteration, object, branch, subobjective index) - Constraint data matrix (cmx) has dimensions: (iteration, object, branch, constraint index, constraint item, time step)

## 46.2 def \_\_init\_\_(self):

```
>>> taskdef1 = Mock('TaskDef')
>>> taskdef1.main_level = u'comp_unit'
>>> taskdef1.num_of_subobjs = 1
>>> taskdef1.num_of_constraints = 1
>>> taskdef1.constraint_sizes = [1]
>>> taskdef1.subobjective_sizes = [1]
```

```
>>> taskdef1.discount_rate = 10.0
>>> taskdef1.init_date = datetime.date(1999,1,1)
>>> taskdef1.subobjectives = [Mock('Subobjective')]
>>> taskdef1.subobjectives[0].expr = [('variable',
...                                   (u'comp_unit', 'PV'),
...                                   (datetime.date(1999,1,1),None),
...                                   None,
...                                   False)]
>>> taskdef1.constraints = [('operation',
...                          'cash_flow',
...                          (datetime.date(2000,1,1),None),
...                          None,
...                          False)]
>>> add_error = Mock('add_error')
```

Initialize OMatrix:

```
>>> from simo.optimization.tools.omatrix import OMatrix
>>> om = OMatrix(taskdef1, add_error)
```

## 46.3 def construct\_data(self, db):

Construct optimization data matrix and fill it with data from input databases

```
>>> db = Mock('DB')
>>> db.get_ids.mock_returns = [u'001', u'002', u'003']
>>> db.get_max_iter.mock_returns = 1
>>> db.get_max_branch.mock_returns = 1
>>> db.get_dates.mock_returns = [datetime.date(1999,1,1),
...                               datetime.date(2000,1,1)]
>>> db.get_data_from_level.mock_returns = [
...     ['001', 0, 0, datetime.date(1999,1,1), 1000.0],
...     ['001', 1, 0, datetime.date(1999,1,1), 1100.0],
...     ['001', 0, 1, datetime.date(1999,1,1), 1010.0],
...     ['001', 1, 1, datetime.date(1999,1,1), 1110.0],
...     ['001', 0, 0, datetime.date(2000,1,1), 1001.0],
...     ['001', 1, 0, datetime.date(2000,1,1), 1101.0],
...     ['001', 0, 1, datetime.date(2000,1,1), 1011.0],
...     ['001', 1, 1, datetime.date(2000,1,1), 1111.0],
...     ['002', 0, 0, datetime.date(1999,1,1), 2000.0],
...     ['002', 1, 0, datetime.date(1999,1,1), 2100.0],
...     ['003', 1, 0, datetime.date(1999,1,1), 3100.0]]
>>> evaluator = Mock('Evaluator')
>>> evaluator.evaluate.mock_returns = True
>>> om._evaluator = evaluator # substitute the real evaluator with a mock

>>> om.construct_data(db) # doctest: +NORMALIZE_WHITESPACE
Called DB.get_ids('data', u'comp_unit')
Called DB.get_max_iter('data', u'comp_unit')
Called DB.get_max_branch('data', u'comp_unit')
Called DB.get_dates('data', u'comp_unit', (datetime.date(1999, 1, 1), None))
Called DB.get_data_from_level(
    'data',
    ['id', 'iteration', 'branch', 'date', 'PV'],
    u'comp_unit',
    dates=(datetime.date(1999, 1, 1), None))
Called Evaluator.evaluate(
    [('variable', (u'comp_unit', 'PV'), (datetime.date(1999, 1, 1), None), None, False)],
    array([[ 1000., 1001.]])
```

```

<BLANKLINE>
    [[ 2000.,    NaN]],
<BLANKLINE>
    [[    NaN,    NaN]], dtype=float32),
    3,
    False)
Called Evaluator.evaluate(
    [('variable', (u'comp_unit', 'PV'), (datetime.date(1999, 1, 1), None), None, False)],
    array([[[ 1010.,  1011.]]],
<BLANKLINE>
    [[    NaN,    NaN]],
<BLANKLINE>
    [[    NaN,    NaN]], dtype=float32),
    3,
    False)
Called Evaluator.evaluate(
    [('variable', (u'comp_unit', 'PV'), (datetime.date(1999, 1, 1), None), None, False)],
    array([[[ 1100.,  1101.]]],
<BLANKLINE>
    [[ 2100.,    NaN]],
<BLANKLINE>
    [[ 3100.,    NaN]], dtype=float32),
    3,
    False)
Called Evaluator.evaluate(
    [('variable', (u'comp_unit', 'PV'), (datetime.date(1999, 1, 1), None), None, False)],
    array([[[ 1110.,  1111.]]],
<BLANKLINE>
    [[    NaN,    NaN]],
<BLANKLINE>
    [[    NaN,    NaN]], dtype=float32),
    3,
    False)
Called DB.get_data_from_level(
    'op_res',
    ['object_id', 'iteration', 'branch', 'date', 'cash_flow'],
    0,
    dates=(datetime.date(2000, 1, 1), None),
    required=['cash_flow'])
True

>>> om.data_level
u'comp_unit'
>>> om._earliest_date
datetime.date(1999, 1, 1)
>>> om._latest_date
>>> om.max_iterations
2
>>> om.num_of_units
3
>>> om._max_branches
2
>>> om.num_of_branches
array([[2, 1, 1],
       [2, 1, 1]])
>>> om._date_index[datetime.date(1999,1,1)]
0
>>> om._date_index[datetime.date(2000,1,1)]
1
>>> om.unit_index[u'001']
0
>>> om.unit_index[u'002']

```

```
1
>>> om.unit_index[u'003']
2
>>> om._unit_range
array([0, 1, 2])
>>> so_array = om.data.root.subobjectives
>>> co_array = om.data.root.constraints
>>> so_array.shape
(2, 3, 2, 1)
>>> co_array.shape
(2, 3, 2, 1, 1, 2)
>>> co_array[0,0,0,0,0,0]
1000.0
>>> co_array[1,0,0,0,0,1]
1101.0
```

## 46.4 def \_get\_matrix\_dimensions(self):

Go through subobjectives, constraints and input database and try to deduce matrix dimensions

## 46.5 def \_create\_empty\_matrices(self):

Construct subobjective and constraint data matrices

## 46.6 def \_process\_exprs(self, exprs, db, exprtype, subobj=True):

Go through the expressions and try to deduce data level and earliest and latest dates

## 46.7 def \_fill\_matrices(self, db):

Fill subobjective and constraint matrices with values from input databases

## 46.8 def \_collect\_values(self, expr, matrix):

Get values for a single expression operand from either data or operation result database.

Collect values for a simple expression with only 'variable' type operands

```
>>> om._db = db
>>> matrix = numpy.zeros([2,3,2,2,2], dtype=float)

>>> dt = (datetime.date(1999,1,1), None)
>>> expr = [['variable', (u'comp_unit', 'PV'), dt, None, False],
...         ['variable', (u'comp_unit', 'AREA'), dt, None, True],
...         ['ari', (lambda a,b: a + b)],
...         ['aggr', numpy.sum]]

>>> om._collect_values(expr, matrix)
Called DB.get_data_from_level(
    'data',
    ['id', 'iteration', 'branch', 'date', 'PV'],
```

```

        u'comp_unit',
        dates=(datetime.date(1999, 1, 1), None))
Called DB.get_data_from_level(
    'data',
    ['id', 'iteration', 'branch', 'date', 'AREA'],
    u'comp_unit',
    dates=(datetime.date(1999, 1, 1), None))
True

```

```

>>> matrix[0,0,0,:,0]
array([ 1000.,  1000.])
>>> matrix[0,0,0,0,:]
array([ 1000.,  1001.])
>>> matrix[:, :, 0, 0, 0]
array([[ 1000.,  2000.,   0.],
       [ 1100.,  2100.,  3100.]])

```

Collect values for an expression with 'operation' type operands

```

>>> expr = [['operation', u'cash_flow', dt, None, True],
...         ['variable', (u'comp_unit', 'AREA'), dt, None, False],
...         ['ari', (lambda a,b: a + b)],
...         ['aggr', numpy.sum]]
>>> om._collect_values(expr, matrix) # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
Called DB.get_data_from_level(
    'op_res',
    ['object_id', 'iteration', 'branch', 'date', u'cash_flow'],
    0,
    dates=(datetime.date(1999, 1, 1), None),
    required=[u'cash_flow'])
Called DB.get_data_from_level(
    'data',
    ['id', 'iteration', 'branch', 'date', 'AREA'],
    u'comp_unit',
    dates=(datetime.date(1999, 1, 1), None))
True

```

Collect values for an expression with conditions in 'variable' type operands

```

>>> dt = (datetime.date(1999,1,1), None)
>>> cond = [('variable', u'SP'), ('value', 1000), ('gt', (lambda a,b: a>b))]
>>> expr = [['variable', (u'comp_unit', 'PV'), dt, cond, False],
...         ['variable', (u'comp_unit', 'AREA'), dt, None, True],
...         ['ari', (lambda a,b: a + b)],
...         ['aggr', numpy.sum]]
>>> om._collect_values(expr, matrix) # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
Called DB.get_data_from_level(
    'data',
    ['id', 'iteration', 'branch', 'date', 'PV'],
    u'comp_unit',
    dates=(datetime.date(1999, 1, 1), None))
Called DB.get_data_from_level(
    'data',
    ['id', 'iteration', 'branch', 'date', u'SP'],
    u'comp_unit',
    dates=None)
Called Evaluator.evaluate(
    [('variable', u'SP'), ('value', 1000), ('gt', <function <lambda> at ...>)],
    array([[ 1000.,  1001.],
<BLANKLINE>
          [[ 2000.,   NaN]],
<BLANKLINE>

```

```
        [[ NaN,    NaN]], dtype=float32))
Called Evaluator.evaluate(
    [('variable', u'SP'), ('value', 1000), ('gt', <function <lambda> at ...>)],
    array([[ 1010.,  1011.]],
<BLANKLINE>
        [[ NaN,    NaN]],
<BLANKLINE>
        [[ NaN,    NaN]], dtype=float32))
Called Evaluator.evaluate(
    [('variable', u'SP'), ('value', 1000), ('gt', <function <lambda> at ...>)],
    array([[ 1100.,  1101.]],
<BLANKLINE>
        [[ 2100.,    NaN]],
<BLANKLINE>
        [[ 3100.,    NaN]], dtype=float32))
Called Evaluator.evaluate(
    [('variable', u'SP'), ('value', 1000), ('gt', <function <lambda> at ...>)],
    array([[ 1110.,  1111.]],
<BLANKLINE>
        [[ NaN,    NaN]],
<BLANKLINE>
        [[ NaN,    NaN]], dtype=float32))
Called DB.get_data_from_level(
    'data',
    ['id', 'iteration', 'branch', 'date', 'AREA'],
    u'comp_unit',
    dates=(datetime.date(1999, 1, 1), None))
True
```

### Collect values for an expression with conditions in 'variable' type operands

```
>>> expr = [['operation', u'cash_flow', dt,
...         [('operation', u'operation_name'),
...         ('value', u'thinning'),
...         ('eq', (lambda a,b: a == b))], True],
...         ['variable', (u'comp_unit', 'AREA'), dt, None, False],
...         ['ari', (lambda a,b: a * b)],
...         ['aggr', numpy.sum]]

>>> db = Mock('DB')
>>> db.get_data_from_level.mock_returns_iter = \
...     iter([[['001', 0, 0, datetime.date(1999,1,1), 1000.0, u'thinning'],
...            ['001', 1, 0, datetime.date(2000,1,1), 2000.0, u'clearcut']],
...          [['001', 0, 0, datetime.date(1999,1,1), 2.5],
...          ['001', 1, 0, datetime.date(2000,1,1), 3.5]]])
>>> om._db = db
>>> om._evaluator = Mock('Evaluator')
>>> om._evaluator.evaluate.mock_returns_iter = iter([True, False])
>>> matrix[...] = 0.0
>>> om._collect_values(expr, matrix) # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
Called DB.get_data_from_level(
    'op_res',
    ['object_id', 'iteration', 'branch', 'date', u'cash_flow',
u'operation_name'],
    0,
    dates=(datetime.date(1999, 1, 1), None),
    required=[u'cash_flow'])
Called Evaluator.evaluate(
    [('operation', u'operation_name'), ('value', u'thinning'),
('eq', <function <lambda> at ...>)],
    [u'thinning'],
    1)
```

```

Called Evaluator.evaluate(
    [('operation', u'operation_name'), ('value', u'thinning'),
 ('eq', <function <lambda> at ...>)],
    [u'clearcut'],
    1)
Called DB.get_data_from_level(
    'data',
    ['id', 'iteration', 'branch', 'date', 'AREA'],
    u'comp_unit',
    dates=(datetime.date(1999, 1, 1), None))
True

>>> matrix[:,0,0,:,:)
array([[ 1000. ,    0. ],
       [   2.5,    0. ]],
<BLANKLINE>
       [[    0. ,    0. ],
       [    0. ,    3.5]])

```

## 46.9 def \_evaluate\_operand\_constraints(self, data, matrix, expr):

Evaluate constraint for a single operand and store only those objects for which the condition evaluated True

## 46.10 def solution\_utility(self, solution, iteration):

Check objective function utility value in given solution

## 46.11 def \_value2utility(self, iso, value, maxvalue):

Transform real values to utilities

## 46.12 def solution\_feasibility(self, solution, iteration):

Check given solution feasibility

## 46.13 def compare\_utilities(self, best, candidate):

Compare utility values of two solutions: is candidate better than the current best solution?

```

>>> om.task_def.objective_func.target = 'max'
>>> om.compare_utilities(0.0, 1.0)
True
>>> om.compare_utilities(2.0, 1.0)
False

>>> om.task_def.objective_func.target = 'min'
>>> om.compare_utilities(0.0, 1.0)
False
>>> om.compare_utilities(2.0, 1.0)
True

```

## 46.14 def get\_random\_solution(self, iteration):

Get a random solution without checkin it's feasibility

```
>>> om.get_random_solution(0).shape  
(3,)
```



# POSTFIXEVAL.PY

SIMO optimization postfix evaluator

## 47.1 class PostfixEvaluator:

Class for evaluating expressions in postfix stacks:

```
>>> epsilon = 0.00001

>>> from simo.optimization.tools.postfixeval import PostfixEvaluator
>>> evaluator = PostfixEvaluator()
```

### 47.1.1 def evaluate(self, postfix, values, dims=3):

Evaluate postfix expression, where postfix is the expression in Reverse Polish Notation (RPN) or postfix form.

Values can be arrays of three possible shapes: either the multiple data values of multiple objects (dims == 3), multiple date values of a single object (dims == 2) or single date for single object (dims == 1).

When the values are for a single date of single unit, the only dimension is (operand,)

```
>>> values0 = [1.0, 2.0]
```

When the values are for multiple dates of a single object, the dimensions are: (operand, time step)

```
>>> import numpy
>>> nan = numpy.NaN

>>> values1 = numpy.array([[nan, nan, nan, 2],
...                        [nan, nan, nan, 2]],
...                        dtype=float)
```

When values contain the values of multiple dates for multiple objects, the data dimensions are: (object, operand, time step)

```
>>> values2 = numpy.array([[nan, nan, nan, 1],
...                        [nan, nan, nan, 1]],
...                        [[nan, nan, nan, 2],
...                        [nan, nan, nan, 2]],
...                        [[nan, nan, nan, 3],
...                        [nan, nan, nan, 3]]],
...                        dtype=float)
```

Evaluate: sum(X + Y), single dimensions

```
>>> postfix = [('variable', 'X',),
...             ('variable', 'Y',),
...             ('ari', (lambda a,b: a + b),)]

>>> abs(evaluator.evaluate(postfix, values0, 1) - 3) < epsilon
True
```

Evaluate:  $\text{sum}(X / Y)$ , single dimensions

```
>>> postfix = [('variable', 'X',),
...             ('variable', 'Y',),
...             ('ari', (lambda a,b: a / b),)]

>>> abs(evaluator.evaluate(postfix, values0, 1) - 0.5) < epsilon
True
```

Evaluate:  $\text{sum}(X + Y)$ , two dimensions

```
>>> postfix = [('variable', 'X',),
...             ('variable', 'Y',),
...             ('ari', (lambda a,b: a + b),),
...             ('aggr', numpy.nansum,)]

>>> abs(evaluator.evaluate(postfix, values1, 2) - 4) < epsilon
True
```

Evaluate:  $\text{sum}(X + Y)$ , three dimensions

```
>>> postfix = [('variable', 'X',),
...             ('variable', 'Y',),
...             ('ari', (lambda a,b: a + b),),
...             ('aggr', numpy.nansum,)]
>>> abs(evaluator.evaluate(postfix, values2) - 12) < epsilon
True
```

Evaluate:  $\text{sum}(X * Y)$

```
>>> postfix = [('variable', 'X',),
...             ('variable', 'Y',),
...             ('ari', (lambda a,b: a * b),),
...             ('aggr', numpy.nansum,)]
>>> abs(evaluator.evaluate(postfix, values2) - 14) < epsilon
True
```

Evaluate:  $\text{sum}(X) == \text{sum}(Y)$

```
>>> postfix = [('variable', 'X',),
...             ('aggr', numpy.nansum,),
...             ('variable', 'Y',),
...             ('aggr', numpy.nansum,),
...             ('eq', (lambda a,b: a == b),)]
>>> evaluator.evaluate(postfix, values2)
True
```

Evaluate:  $\text{sum}(X + Y / Z)$

```
>>> values3 = numpy.array([[nan, nan, nan, 1],
...                        [nan, nan, nan, 1],
...                        [nan, nan, nan, 2]],
...                        [[nan, nan, nan, 2],
...                        [nan, nan, nan, 2],
```

```

...             [nan, nan, nan, 5]],
...             [[nan, nan, nan, 3],
...             [nan, nan, nan, 3],
...             [nan, nan, nan, 10]]],
...             dtype=float)

>>> postfix = [('variable', 'X',),
...             ('variable', 'Y',),
...             ('variable', 'Z',),
...             ('ari', (lambda a,b: a / b)),
...             ('ari', (lambda a,b: a + b)),
...             ('aggr', numpy.nansum,)]
>>> abs(evaluator.evaluate(postfix, values3) - 7.2) < epsilon
True

```

Evaluate:  $\text{sum}(X * Y / Z)$

```

>>> postfix = [('variable', 'X',),
...             ('variable', 'Y',),
...             ('ari', (lambda a,b: a * b)),
...             ('variable', 'Z',),
...             ('ari', (lambda a,b: a / b)),
...             ('aggr', numpy.nansum,)]
>>> abs(evaluator.evaluate(postfix, values3) - 2.2) < epsilon
True

```

Evaluate:  $\text{sum}(X + Y) < \text{sum}(Z)$

```

>>> postfix = [('variable', 'X',),
...             ('variable', 'Y',),
...             ('ari', (lambda a,b: a + b)),
...             ('aggr', numpy.nansum,),
...             ('variable', 'Z',),
...             ('aggr', numpy.nansum,),
...             ('eq', (lambda a,b: a < b))]
>>> evaluator.evaluate(postfix, values3)
True

```

Evaluate:  $\text{sum}(X + Y) < \text{sum}(Z)$

```

>>> postfix = [('variable', 'X',),
...             ('variable', 'Y',),
...             ('ari', (lambda a,b: a + b)),
...             ('aggr', numpy.nansum,),
...             ('variable', 'Z',),
...             ('aggr', numpy.nansum,),
...             ('eq', (lambda a,b: a < b))]
>>> evaluator.evaluate(postfix, values3)
True

```

Evaluate:  $(\text{sum}(X) / \text{sum}(Y)) == 1.0$

```

>>> postfix = [('variable', 'X',),
...             ('aggr', numpy.nansum,),
...             ('variable', 'Y',),
...             ('aggr', numpy.nansum,),
...             ('ari', (lambda a,b: a / b)),
...             ('value', 1.0,),
...             ('eq', (lambda a,b: a == b))]
>>> evaluator.evaluate(postfix, values3)
True

```

Evaluate:  $\text{sum}((X + Y) * X) == 84.0$

```
>>> postfix = [('variable', 'X'),
...            ('variable', 'Y'),
...            ('ari', (lambda a,b: a + b)),
...            ('variable', 'Z'),
...            ('ari', (lambda a,b: a * b)),
...            ('aggr', numpy.nansum),
...            ('value', 84),
...            ('eq', (lambda a,b: a == b))]
>>> evaluator.evaluate(postfix, values3)
True
```

Output modules:

# AGGR.PY

## 48.1 class OutputAggr(Output):

Module for data output in hierarchical “data in columns” format. (writing tables to files)

**48.1.1 def \_\_init\_\_(self, datadb, data\_type, id\_list, main\_level, result\_type, output\_formats, output\_filenames, output\_constraint=None, default\_decimal\_places=1, archiving=False, result\_padding=True, aggregation\_def=None, expression\_def=None, opres\_vars=None):**

Creates the output class. fully inherited, executes self.run()

```
>>> from simo.output.test.init_objects import *
>>> from collections import defaultdict
>>> import datetime
>>> import numpy
>>> idata = InitData()
>>> idata.init()
>>> testdb = idata.testdb
>>> const_obj = idata.const_obj
>>> aggr_obj = idata.aggr_obj
>>> expr_obj = idata.expr_obj
>>> from simo.output.aggr import OutputAggr
>>> out = OutputAggr(testdb, 'result', ['stand1'], 'stratum',
...                  'optimized',
...                  'output/test/aggregation.txt', TestLogger(),
...                  const_obj, 1, False, True, aggr_obj, expr_obj,
...                  ['cash_flow', 'Income', 'Scrapwood', 'Volume',
...                  'BIOMASS_branches', 'BIOMASS_stumps'])
>>> try: # doctest: +ELLIPSIS
...     file = open('output/test/aggregation.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
Results for expression (5 keys): sum[-1:-1](stratum:HgM[SP eq 2 and DgM gt 19])
stratum:SP; stratum:DgM; 08/02/09-08/02/09
1.0; 20.0; NaN
2.0; 0.0; NaN
2.0; 18.0; NaN
2.0; 30.0; 62.0
3.0; 27.0; NaN
```

Errors (key: stratum:SP, stratum:DgM):

No values for stratum:HgM between 2009-02-08 and 2009-02-08;1.0, 20.0;2.0, 0.0;2.0, 18.0;3.0, 27.0  
<BLANKLINE>

```
Proportions for expression (5 keys): sum[0:99] (stratum:SP*stratum:HgM)
  stratum:HgM; stratum:DgM; 01/01/09-08/02/09
    0.0;          0.0;          0.0
    18.5;         18.0;         0.2
    19.0;         20.0;         0.1
    26.0;         27.0;         0.4
    31.0;         30.0;         0.3
Errors (key: stratum:HgM, stratum:DgM):
<BLANKLINE>
Results for expression (1 keys): sum[0:99] (operation:Volume)
  01/01/09-08/02/09
    123.2
Errors (key: ):
<BLANKLINE>
Results for expression (1 keys): sum[0:99] (operation:cash_flow)
  01/01/09-08/02/09
    3000.0
Errors (key: ):
<BLANKLINE>
Results for expression (2 keys): sum[0:99] (operation:cash_flow)
  comp_unit:AREA; 01/01/09-08/02/09
    1.0;          3000.0
    2.0;          NaN
Errors (key: comp_unit:AREA):
No values for operation:cash_flow between 2009-01-01 and 2009-02-08;2.0
<BLANKLINE>
```

#### 48.1.2 def run(self):

Uses data from init to run the class-specific output (self.aggregation)

#### 48.1.3 def \_run\_aggr(self, aggr\_dict):

Executes the given aggregation dictionary for all reporting periods and return the results in a list of lists of strings.

#### 48.1.4 def \_execute\_var(self, discount\_rate, time\_unit, time\_length, var):

Executes the given aggregation set with the given values and returns results as a list of strings.

#### 48.1.5 def \_parse\_experssion(self, expression, header, init\_date):

Will parse a string expression into a list of tuples. Returns None if there are errors.

```
>>> out._parse_expression('sum[3:5] (comp_unit:AREA)',
...                       'sum[3:5] (comp_unit:AREA)',
...                       datetime.date(2009, 1, 1))
... #doctest: +NORMALIZE_WHITESPACE +ELLIPSIS
[('variable', ('comp_unit', u'AREA'),
 (datetime.date(2011, 1, 1), datetime.date(2013, 12, 31)), None, False),
 ('aggr', <function nansum at ...>)]
```

#### 48.1.6 def \_get\_classifiers(self, keys):

Build and return classifiers for the given grouping

```
>>> out._get_classifiers(['comp_unit:AREA', 'operation:Volume'])
[(1.0, 123.2)]
```

#### 48.1.7 def \_get\_values(self, expr, date\_ranges, grouping, errors, date\_dict):

Get the values from the database into the date\_dict

```
>>> expr = [('operation', u'Volume', (datetime.date(2009, 2, 7),
...     datetime.date(2108, 2, 6)), None, False), ('aggr', numpy.nansum)]
>>> date_ranges = [(datetime.date(2009, 2, 7), datetime.date(2009, 2, 8),
...     'year', True)]
>>> grouping = None
>>> errors = defaultdict(set)
>>> date_dict = {}
>>> date_dict, classifiers, errors = out._get_values(expr, date_ranges,
...     grouping, errors,
...     date_dict,
...     datetime.date(2009, 2, 7),
...     3.0)
>>> date_dict, classifiers, errors #doctest: +NORMALIZE_WHITESPACE
((datetime.date(2009, 2, 7), datetime.date(2009, 2, 8), 'year', True):
    defaultdict(<type 'list'>,
    {'': [array([ 123.2])]})), [], defaultdict(<type 'set'>, {}))
```

#### 48.1.8 def \_get\_results(self, date\_dict, grouping, classifiers, expr, proportion):

Evaluate the expression and return a list of results and the header for the result grouping

```
>>> proportion = False
>>> grouping = {}
>>> out._get_results(date_dict, grouping, classifiers, expr, proportion)
[['07/02/09-08/02/09'], ['123.2'], []]
```

#### 48.1.9 def \_check\_date\_range(self, range, val):

Check if the val is inside the given range

```
>>> out._check_date_range(date_ranges[0], datetime.date(2009, 2, 6))
False
>>> out._check_date_range(date_ranges[0], datetime.date(2009, 2, 7))
True
>>> out._check_date_range(date_ranges[0], datetime.date(2009, 2, 8))
True
>>> out._check_date_range((datetime.date(2009, 2, 7),
...     datetime.date(2009, 2, 8), 'year', False),
...     datetime.date(2009, 2, 8))
False
```

#### 48.1.10 def \_get\_date\_ranges(self, init\_date, all\_dates, expr, time\_unit, time\_length):

This gets the date ranges to be used for the data

```
>>> init_date = datetime.date(2009, 2, 7)
>>> all_dates = [datetime.date(2009, 2, 7), datetime.date(2009, 3, 1)]
>>> out._get_date_ranges(init_date, all_dates, expr, 'day', 5)
... #doctest: +NORMALIZE_WHITESPACE
[(datetime.date(2009, 2, 7), datetime.date(2009, 2, 12), 'day', False),
 (datetime.date(2009, 2, 12), datetime.date(2009, 2, 17), 'day', False),
 (datetime.date(2009, 2, 17), datetime.date(2009, 2, 22), 'day', False),
 (datetime.date(2009, 2, 22), datetime.date(2009, 2, 27), 'day', False),
 (datetime.date(2009, 2, 27), datetime.date(2009, 3, 1), 'day', True)]
```

#### 48.1.11 def \_build\_ranges(self, dates, time\_unit, time\_length, first=None, last=None):

Returns a list of tuples of (start\_date, end\_date, time\_unit, is\_last\_range)

```
>>> out._build_ranges(all_dates, 'day', 5, first=datetime.date(2009,2,20),
...                   last=datetime.date(2009,2,28))
... #doctest: +NORMALIZE_WHITESPACE
[(datetime.date(2009, 2, 20), datetime.date(2009, 2, 25), 'day', False),
 (datetime.date(2009, 2, 25), datetime.date(2009, 2, 28), 'day', True)]
```

#### 48.1.12 def \_get\_string\_range(self, rng):

Turns the given range tuple into a string representation

```
>>> out._get_string_range((datetime.date(2009, 2, 20),
...                        datetime.date(2009, 3, 25), 'day', False))
'20/02/09-24/03/09'
>>> out._get_string_range((datetime.date(2009, 2, 20),
...                        datetime.date(2009, 3, 25), 'month', True))
'20/02/09-25/03/09'
>>> out._get_string_range((datetime.date(2009, 2, 20),
...                        datetime.date(2009, 3, 25), 'year', True))
'20/02/09-25/03/09'
```

#### 48.1.13 def \_unpack\_weighted\_average(self, data):

Unpacks a weighted average into a formula the expression parser can handle.

```
>>> out._unpack_weighted_average('sum[1:5](one:thing)')
'sum[1:5](one:thing)'
>>> out._unpack_weighted_average('wavg[1:5](one:thing, another:thing)')
'sum[1:5](one:thing*another:thing)/sum[1:5](another:thing)'
```

def \_get\_values\_from\_db(self, operand, dates, norm, opresconst=None, grouping=None): =====

Get value(s) for a single operand from data or operation result database. Valid grouping pairs (in either way): level-level, operation-“comp\_unit”, level-child, level-parent

```
>>> out._get_values_from_db(('variable', ('stratum', u'SP'),
...                             (datetime.date(2009, 2, 7),
...                             datetime.date(2108, 2, 6)), None, False),
...                             (datetime.date(2009, 2, 7),
...                             datetime.date(2108, 2, 6)), True, None,
...                             {'stratum': ['HgM', 'DgM']})
... #doctest: +NORMALIZE_WHITESPACE
```



```
([[1.0, datetime.date(2009, 2, 7), 19.0, 20.0],
 [2.0, datetime.date(2009, 2, 7), 18.5, 18.0],
 [1.0, datetime.date(2009, 2, 7), 19.0, 20.0],
 [2.0, datetime.date(2009, 2, 7), 18.5, 18.0],
 [2.0, datetime.date(2009, 2, 8), 0.0, 0.0],
 [2.0, datetime.date(2009, 2, 8), 31.0, 30.0],
 [3.0, datetime.date(2009, 2, 8), 26.0, 27.0],
 [2.0, datetime.date(2009, 2, 8), 31.0, 30.0],
 [3.0, datetime.date(2009, 2, 8), 26.0, 27.0],
 [2.0, datetime.date(2009, 2, 8), 0.0, 0.0]],
 [datetime.date(2009, 2, 8), datetime.date(2009, 2, 7)],
 ['stratum:HgM', 'stratum:DgM'])
>>> out._get_values_from_db(('operation', u'cash_flow',
...                           (datetime.date(2009, 2, 7),
...                           datetime.date(2108, 2, 6)), None, False),
...                           (datetime.date(2009, 2, 7),
...                           datetime.date(2108, 2, 6)), True, None,
...                           {'comp_unit': ['AREA']}))
... #doctest: +NORMALIZE_WHITESPACE
([[3456.6500000000001, datetime.date(2009, 2, 8), 1.0],
 [-456.64999999999998, datetime.date(2009, 2, 8), 1.0]],
 [datetime.date(2009, 2, 8)], ['comp_unit:AREA'])
```

#### 48.1.14 def \_calculate\_NPV(self, value, discountrate, presentdate, eventdate):

Calculate the discounted NPV (Net Present Value) for a single value

```
>>> out._calculate_NPV(10.5, 5.4, datetime.date(2009,1,1),
...                     datetime.date(2011,5,4))
9.4516597114453305
```

#### 48.1.15 def aggregation(self, file):

This method will get the data requested from the databases and writes them into the given file.

```
>>> out.aggregation('output/test/aggregation.txt')
>>> try:
...     file = open('output/test/aggregation.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
Results for expression (5 keys): sum[-1:-1] (stratum:HgM[SP eq 2 and DgM gt 19])
stratum:SP; stratum:DgM; 08/02/09-08/02/09
1.0; 20.0; NaN
2.0; 0.0; NaN
2.0; 18.0; NaN
2.0; 30.0; 62.0
3.0; 27.0; NaN
Errors (key: stratum:SP, stratum:DgM):
No values for stratum:HgM between 2009-02-08 and 2009-02-08;1.0, 20.0;2.0, 0.0;2.0, 18.0;3.0, 27.0
<BLANKLINE>
Proportions for expression (5 keys): sum[0:99] (stratum:SP*stratum:HgM)
stratum:HgM; stratum:DgM; 01/01/09-08/02/09
0.0; 0.0; 0.0
18.5; 18.0; 0.2
19.0; 20.0; 0.1
26.0; 27.0; 0.4
31.0; 30.0; 0.3
```

```
Errors (key: stratum:HgM, stratum:DgM):
<BLANKLINE>
Results for expression (1 keys): sum[0:99](operation:Volume)
  01/01/09-08/02/09
                123.2
Errors (key: ):
<BLANKLINE>
Results for expression (1 keys): sum[0:99](operation:cash_flow)
  01/01/09-08/02/09
                3000.0
Errors (key: ):
<BLANKLINE>
Results for expression (2 keys): sum[0:99](operation:cash_flow)
  comp_unit:AREA; 01/01/09-08/02/09
                1.0;                3000.0
                2.0;                NaN
Errors (key: comp_unit:AREA):
No values for operation:cash_flow between 2009-01-01 and 2009-02-08;2.0
<BLANKLINE>
```

# BRANCHING.PY

## 49.1 class OutputOpres(Output):

This module handles writing operation results into files

**49.1.1 def \_\_init\_\_(self, datadb, data\_type, id\_list, main\_level, result\_type, output\_filename, output\_constraint=None, default\_decimal\_places=1, archiving=False, result\_padding=True, aggregation\_def=None, opres\_vars=None, full\_date=False, dates=None):**

Creates the output class. Fully inherited, executes self.run()

```
>>> import os
>>> from glob import glob
>>> removing = glob('output/test/*.dot')
>>> for file in removing:
...     os.remove(file)
>>> from simo.output.test.init_objects import InitData, TestLogger
>>> idata = InitData()
>>> idata.init()
>>> testdb = idata.testdb
>>> const_obj = idata.const_obj
>>> aggr_obj = idata.aggr_obj
>>> expr_obj = idata.expr_obj
>>> from simo.output.branching import OutputBranching
>>> out = OutputBranching(testdb, 'result', ['stand1'], 'stratum',
...                        'optimized',
...                        'output/test', TestLogger(),
...                        const_obj, 1, False, True, aggr_obj, expr_obj,
...                        None, True)
>>> try: # doctest: +ELLIPSIS
...     file = open('output/test/stratum2-1_0.dot', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
digraph branches {
...
b_0_20090208 [group="0"];
b_1_20090208 [group="1"];
...
{ rank = same; 20090208; b_0_20090208; b_1_20090208; }
...
b_0_20090208 -> b_1_20090208 [ label="branch_name" fontsize=14 fontcolor=red];
...

```

### 49.1.2 def run(self):

Uses data from init to run the class-specific output (self.inlined)

### 49.1.3 def \_get\_branch\_root\_str(self, date, item):

Returns a properly formatted string to describe a root point from where a branch starts

```
>>> import datetime
>>> ddate = datetime.date(2000, 1, 1)
>>> out._get_branch_root_str(ddate, 1)
'b_1_20000101 [label="1" fontcolor=red group="1"];
```

### 49.1.4 def \_get\_branch\_node\_strs(self, item):

Returns a properly formatted string list of node points. item is a list containing a date and the appropriate groups (groups are (from\_branch, to\_branch) tuples)

```
>>> out._get_branch_node_strs([ddate, (0, 1), (0, 2)])
['b_1_20000101 [group="1"];', 'b_2_20000101 [group="2"];
```

### 49.1.5 def \_get\_branch\_ranking\_str(self, item):

Returns a properly formatted string used to align different nodes. item is a list containing a date and the appropriate groups (groups are (from\_branch, to\_branch) tuples)

```
>>> out._get_branch_ranking_str([ddate, (0, 1), (0, 2)])
'{ rank = same; 20000101; b_1_20000101; b_2_20000101; }'
```

### 49.1.6 def \_get\_branch\_branch\_str(self, date, group1, group2, name):

Returns a properly formatted string describing a link from one branch to another (groups are branch numbers)

```
>>> out._get_branch_branch_str(ddate, 0, 1, 'move operation')
'b_0_20000101 -> b_1_20000101 [ label="move operation" fontsize=14 fontcolor=red];'
```

### 49.1.7 def \_get\_branch\_link\_str(self, date1, date2, group):

Returns a properly formatted string describing a link from one node to another within a branch (group is the branch number)

```
>>> out._get_branch_link_str(ddate, ddate.replace(day=2), 1)
'b_1_20000101 -> b_1_20000102;'
```

### 49.1.8 def \_get\_branching\_strings(self):

```

>>> ddate = datetime.date(2009, 2, 7)
>>> data = {'stratum': [(ddate, {'id': 'stratum2-1', 'oid': 'branched',
...                               'parent id': 'stand1',
...                               'values': [(('SP', 0.1), ('BA', 1.2),
...                                           ('DgM', 2.3), ('HgM', 3.4))]}]}
>>> out.datadb.add_data_from_dictionary(data, 0, 0)
>>> dates = out.datadb.get_dates_by_id_and_iter(2)
>>> str21 = out._get_branching_strings(dates[('stratum2-1', 0)],
...                                     'stratum2-1', 0)
>>> for i in str21:
...     for j in i:
...         print j
...     print ''
digraph branches {
nodesep=0.01;
ranksep=0.01;
size="8.27,11.69";
ratio=fill;
<BLANKLINE>
{
node [shape=plaintext fontsize=14 height=.01];
<BLANKLINE>
20090207 -> 20090208;
}
<BLANKLINE>
{
node [shape=doublecircle];
<BLANKLINE>
b_0_20090207 [label="0" fontcolor=red group="0"];
b_1_20090208 [label="1" fontcolor=red group="1"];
}
<BLANKLINE>
{
node [shape=point]
<BLANKLINE>
b_0_20090207 [group="0"];
b_0_20090208 [group="0"];
b_1_20090208 [group="1"];
}
<BLANKLINE>
{ rank = same; 20090207; b_0_20090207; }
{ rank = same; 20090208; b_0_20090208; b_1_20090208; }
<BLANKLINE>
b_0_20090208 -> b_1_20090208 [ label="branch_name" fontsize=14 fontcolor=red];
b_0_20090207 -> b_0_20090208;
<BLANKLINE>
}
<BLANKLINE>

```

#### 49.1.9 def \_format\_date(self, date):

Formats date string according to date\_res (either full date or just the year)

```

>>> out._format_date(datetime.date(2000,1,1))
'20000101'
>>> out.date_res = 0
>>> out._format_date(datetime.date(2000,1,1))
'2000'
>>> out.date_res = 1

```

### 49.1.10 def branching(self, path):

Will write branching .dots for each id on the main level from the database to the given path

```
>>> out.branching('output/test')
>>> lines = []
>>> try:
...     file = open('output/test/stratum2-1_0.dot', 'r')
...     for line in file:
...         lines.append(line.rstrip('\n'))
... finally:
...     file.close()
>>> resultlines = []
>>> for item in str21:
...     for line in item:
...         resultlines.append(line)
...         resultlines.append('')
>>> lines == resultlines
True
>>> from glob import glob
>>> set([os.path.basename(file) for file in glob('output/test/*.dot')])==\
...     set(['stratum1-1_0.dot', 'stratum2-2_1.dot', 'stratum2-1_0.dot',
...         'stratum2-2_0.dot', 'stratum2-3_0.dot', 'stratum2-3_1.dot',
...         'stratum1-2_1.dot', 'stratum2-1_1.dot', 'stratum1-1_1.dot',
...         'stratum1-2_0.dot'])
True
>>> r = out.datadb.conn.execute('DELETE FROM stratum WHERE oid="branched"')
```

# BY\_LEVEL.PY

## 50.1 class OutputByLevel(Output):

Module for data output in hierarchical “data in columns” format. (writing tables to files). This module writes the data in a more spreadsheet-compatible manner.

**50.1.1 def \_\_init\_\_(self, datadb, data\_type, id\_list, main\_level, result\_type, output\_filename, output\_constraint=None, default\_decimal\_places=1, archiving=False, result\_padding=True, aggregation\_def=None, expression\_def=None, opres\_vars=None):**

Creates the output class. fully inherited, executes self.run()

```
>>> from pprint import pprint
>>> from simo.output.test.init_objects import InitData, TestLogger
>>> idata = InitData()
>>> idata.init()
>>> testdb = idata.testdb
>>> const_obj = idata.const_obj
>>> origvars = const_obj.variables
>>> const_obj.variables = {('stratum',2):origvars[('stratum', 2)]}
>>> aggr_obj = idata.aggr_obj
>>> expr_obj = idata.expr_obj
>>> from simo.output.by_level import OutputByLevel
>>> out = OutputByLevel(testdb, 'result', ['stand1'], 'stratum',
...                       'optimized',
...                       'output/test/by_level.txt', TestLogger(),
...                       const_obj, 1, False, True, aggr_obj, expr_obj,
...                       ['cash_flow', 'Income', 'Scrapwood', 'Volume',
...                       'BIOMASS_branches', 'BIOMASS_stumps'])
>>> try: # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
...     file = open('output/test/by_level.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
sim unit; branch; iteration; data level;          id; date;  SP;   BA;  HgM
stand1;      0;          0;    stratum; stratum1-1; 2009; 1.0; 17.0; 19.0
stand1;      0;          0;    stratum; stratum1-2; 2009; 2.0;  8.0; 18.5
stand1;      0;          1;    stratum; stratum1-1; 2009; 1.0; 17.0; 19.0
stand1;      0;          1;    stratum; stratum1-2; 2009; 2.0;  8.0; 18.5
>>> const_obj.variables = origvars
>>> out = OutputByLevel(testdb, 'result', ['stand1'], 'stratum',
...                       'optimized',
...                       'output/test/by_level.txt', TestLogger(),
```

```
...         const_obj, 1, False, True, aggr_obj,
...         ['cash_flow', 'Income', 'Scrapwood', 'Volume',
...         'BIOMASS_branches', 'BIOMASS_stumps'])
>>> try: # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
...     file = open('output/test/by_level.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
sim unit; branch; iteration; data level;          id; date; AREA; SC; SP; BA; HgM; d;
stand1; 0; 0; comp_unit; stand1; 2009; 1.0; 3.0; ; ; ; ;
stand1; 0; 0; stratum; stratum1-1; 2009; ; ; 1.0; 17.0; 19.0; ;
stand1; 0; 0; tree; tree1-1-1; 2009; ; ; ; ; ; 17.0; 18.
stand1; 0; 0; stratum; stratum1-2; 2009; ; ; 2.0; 8.0; 18.5; ;
stand1; 0; 0; tree; tree1-2-1; 2009; ; ; ; ; ; 22.0; 21.
stand1; 0; 1; comp_unit; stand1; 2009; 1.0; 3.0; ; ; ; ;
stand1; 0; 1; stratum; stratum1-1; 2009; ; ; 1.0; 17.0; 19.0; ;
stand1; 0; 1; tree; tree1-1-1; 2009; ; ; ; ; ; 17.0; 18.
stand1; 0; 1; stratum; stratum1-2; 2009; ; ; 2.0; 8.0; 18.5; ;
stand1; 0; 1; tree; tree1-2-1; 2009; ; ; ; ; ; 22.0; 21.0
```

### 50.1.2 def run(self):

Uses data from init to run the class-specific output (self.by\_level)

### 50.1.3 def \_flatten\_header(self, header):

This will return a one dimensional list, flattened from a two dimensional list

```
>>> wh = out._get_wide_header()
>>> wh # doctest: +NORMALIZE_WHITESPACE
[['sim unit', 'branch', 'iteration', 'data level', 'id', 'date'],
 ['AREA', 'SC'], ['SP', 'BA', 'HgM'], ['d', 'h']]
>>> out._flatten_header(wh) # doctest: +NORMALIZE_WHITESPACE
['sim unit', 'branch', 'iteration', 'data level', 'id', 'date', 'AREA',
 'SC', 'SP', 'BA', 'HgM', 'd', 'h']
```

### 50.1.4 def \_get\_wide\_header(self):

This function will get a two dimensional header list for all the tables in the datadb, one list per table with the common arguments at the start.

```
>>> out._get_wide_header()
[['sim unit', 'branch', 'iteration', 'data level', 'id', 'date'], ['AREA', 'SC'], ['SP', 'BA', 'HgM', 'd', 'h']]
```

### 50.1.5 def \_set\_wide\_vars(self, item, level, wide\_header):

This function will create a (wide variant) row from the given variables, setting values to their correct places based on level.

```
>>> out._set_wide_vars(['sim unit', 'branch', 'iteration', 'data level', 'id', 'date', 'sp', 'ba', 'hg', 'd', 'h'],
 ['sim unit', 'branch', 'iteration', 'data level', 'id', 'date', '', '', 'sp', 'ba', 'hgm', '', ''])
```



### 50.1.6 def `_get_strings_wide(self, wide_header=None, constraints=None, dates=None, date_res=0, level=1, sim_unit=None)`:

This function functions exactly as `_get_strings`, except that it uses the wide table variant (header from `_get_wide_header`, rows from `_set_wide_vars`)

```
>>> strings = out._get_strings_wide()
>>> pprint(strings)
[' sim unit; branch; iteration; data level;          id; date; AREA;  SC;  SP;   BA;  HgM;    d;
 ' stand1;      0;          0; comp_unit;    stand1; 2009;  1.0; 3.0;    ;    ;    ;    ;
 ' stand1;      0;          0; stratum; stratum1-1; 2009;    ;    ; 1.0; 17.0; 19.0;    ;
 ' stand1;      0;          0; tree; tree1-1-1; 2009;    ;    ;    ;    ;    ; 17.0; 18.0;
 ' stand1;      0;          0; stratum; stratum1-2; 2009;    ;    ; 2.0;  8.0; 18.5;    ;
 ' stand1;      0;          0; tree; tree1-2-1; 2009;    ;    ;    ;    ;    ; 22.0; 21.0;
 ' stand1;      0;          1; comp_unit;    stand1; 2009;  1.0; 3.0;    ;    ;    ;    ;
 ' stand1;      0;          1; stratum; stratum1-1; 2009;    ;    ; 1.0; 17.0; 19.0;    ;
 ' stand1;      0;          1; tree; tree1-1-1; 2009;    ;    ;    ;    ;    ; 17.0; 18.0;
 ' stand1;      0;          1; stratum; stratum1-2; 2009;    ;    ; 2.0;  8.0; 18.5;    ;
 ' stand1;      0;          1; tree; tree1-2-1; 2009;    ;    ;    ;    ;    ; 22.0; 21.0;
 ' stand2;      0;          0; comp_unit;    stand2; 2009;  2.0; 2.0;    ;    ;    ;    ;
 ' stand2;      0;          0; stratum; stratum2-1; 2009;    ;    ; 2.0; 24.0; 31.0;    ;
 ' stand2;      0;          0; tree; tree2-1-1; 2009;    ;    ;    ;    ;    ; 30.0; 29.0;
 ' stand2;      0;          0; stratum; stratum2-2; 2009;    ;    ; 3.0; 12.0; 26.0;    ;
 ' stand2;      0;          0; tree; tree2-2-1; 2009;    ;    ;    ;    ;    ; 26.0; 25.0;
 ' stand2;      0;          0; tree; tree2-2-2; 2009;    ;    ;    ;    ;    ; 28.0; 27.0;
 ' stand2;      0;          1; comp_unit;    stand2; 2009;  2.0; 2.0;    ;    ;    ;    ;
 ' stand2;      0;          1; stratum; stratum2-1; 2009;    ;    ; 2.0; 24.0; 31.0;    ;
 ' stand2;      0;          1; tree; tree2-1-1; 2009;    ;    ;    ;    ;    ; 30.0; 29.0;
 ' stand2;      0;          1; stratum; stratum2-2; 2009;    ;    ; 3.0; 12.0; 26.0;    ;
 ' stand2;      0;          1; tree; tree2-2-1; 2009;    ;    ;    ;    ;    ; 26.0; 25.0;
 ' stand2;      0;          1; tree; tree2-2-2; 2009;    ;    ;    ;    ;    ; 28.0; 27.0;
 ' stand2;      1;          0; comp_unit;    stand2; 2009;  2.0; 2.0;    ;    ;    ;    ;
 ' stand2;      1;          0; stratum; stratum2-3; 2009;    ;    ; 2.0;  0.0;  0.0;    ;
 ' stand2;      1;          1; comp_unit;    stand2; 2009;  2.0; 2.0;    ;    ;    ;    ;
 ' stand2;      1;          1; stratum; stratum2-3; 2009;    ;    ; 2.0;  0.0;  0.0;    ;
```

### 50.1.7 def `by_level(self, file, constraints=None, dates=None, date_res=0)`:

This function will use `_get_strings_wide` with the given restrictions and date resolution, and writes the results into the given file.

```
>>> out.by_level('output/test/by_level.txt')
>>> try:
...     file = open('output/test/by_level.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
 sim unit; branch; iteration; data level;          id; date; AREA;  SC;  SP;   BA;  HgM;    d;
 stand1;      0;          0; comp_unit;    stand1; 2009;  1.0; 3.0;    ;    ;    ;    ;
 stand1;      0;          0; stratum; stratum1-1; 2009;    ;    ; 1.0; 17.0; 19.0;    ;
 stand1;      0;          0; tree; tree1-1-1; 2009;    ;    ;    ;    ;    ; 17.0; 18.0;
 stand1;      0;          0; stratum; stratum1-2; 2009;    ;    ; 2.0;  8.0; 18.5;    ;
 stand1;      0;          0; tree; tree1-2-1; 2009;    ;    ;    ;    ;    ; 22.0; 21.0;
 stand1;      0;          1; comp_unit;    stand1; 2009;  1.0; 3.0;    ;    ;    ;    ;
 stand1;      0;          1; stratum; stratum1-1; 2009;    ;    ; 1.0; 17.0; 19.0;    ;
 stand1;      0;          1; tree; tree1-1-1; 2009;    ;    ;    ;    ;    ; 17.0; 18.0;
 stand1;      0;          1; stratum; stratum1-2; 2009;    ;    ; 2.0;  8.0; 18.5;    ;
 stand1;      0;          1; tree; tree1-2-1; 2009;    ;    ;    ;    ;    ; 22.0; 21.0;
 stand2;      0;          0; comp_unit;    stand2; 2009;  2.0; 2.0;    ;    ;    ;    ;
```

```
stand2;      0;      0;      stratum; stratum2-1; 2009;      ;      ; 2.0; 24.0; 31.0;      ;
stand2;      0;      0;      tree; tree2-1-1; 2009;      ;      ;      ;      ; 30.0; 29.0;
stand2;      0;      0;      stratum; stratum2-2; 2009;      ;      ; 3.0; 12.0; 26.0;      ;
stand2;      0;      0;      tree; tree2-2-1; 2009;      ;      ;      ;      ; 26.0; 24.0;
stand2;      0;      0;      tree; tree2-2-2; 2009;      ;      ;      ;      ; 28.0; 27.0;
stand2;      0;      1; comp_unit;      stand2; 2009; 2.0; 2.0;      ;      ;      ;
stand2;      0;      1;      stratum; stratum2-1; 2009;      ;      ; 2.0; 24.0; 31.0;      ;
stand2;      0;      1;      tree; tree2-1-1; 2009;      ;      ;      ;      ; 30.0; 29.0;
stand2;      0;      1;      stratum; stratum2-2; 2009;      ;      ; 3.0; 12.0; 26.0;      ;
stand2;      0;      1;      tree; tree2-2-1; 2009;      ;      ;      ;      ; 26.0; 24.0;
stand2;      0;      1;      tree; tree2-2-2; 2009;      ;      ;      ;      ; 28.0; 27.0;
stand2;      1;      0; comp_unit;      stand2; 2009; 2.0; 2.0;      ;      ;      ;
stand2;      1;      0;      stratum; stratum2-3; 2009;      ;      ; 2.0; 0.0; 0.0;      ;
stand2;      1;      1; comp_unit;      stand2; 2009; 2.0; 2.0;      ;      ;      ;
stand2;      1;      1;      stratum; stratum2-3; 2009;      ;      ; 2.0; 0.0; 0.0;      ;
```

# EXPRESSION.PY

```
>>> import datetime
```

## 51.1 def calculate\_NPV(value, discountrate, presentdate, eventdate):

Calculate the discounted NPV (Net Present Value) for a single value

```
>>> from simo.output.expression import calculate_NPV
>>> calculate_NPV(10.5, 5.4, datetime.date(2009,1,1),
...             datetime.date(2011,5,4))
9.4516597114453305
```

### 51.1.1 class OutputExpr(Output):

Class for expression output, i.e. output variables defined with a syntax similar to optimisation task definitions

## 51.2 def \_\_init\_\_(self, datadb, data\_type, id\_list, main\_level, result\_type, output\_formats, output\_filenames, output\_constraint=None, default\_decimal\_places=1, archiving=False, result\_padding=True, aggregation\_def=None, expression\_def=None, opres\_vars=None):

Creates the output class. fully inherited, executes self.run()

```
>>> from simo.output.test.init_objects import *
>>> from collections import defaultdict
>>> import datetime
>>> import numpy
>>> idata = InitData()
>>> idata.init()
>>> testdb = idata.testdb
>>> const_obj = idata.const_obj
>>> aggr_obj = idata.aggr_obj
>>> expr_obj = idata.expr_obj
>>> from simo.output.expression import OutputExpr
>>> out = OutputExpr(testdb, 'result', ['standl'], 'stratum',
...                  'optimized',
...                  'output/test/expression.txt', TestLogger(),
```

```
...             const_obj, 1, False, True, aggr_obj, expr_obj,
...             ['cash_flow', 'Income', 'Scrapwood', 'Volume',
...             'BIOMASS_branches', 'BIOMASS_stumps'])
>>> try: # doctest: +ELLIPSIS
...     file = open('output/test/expression.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
level;unit;iteration;branch;1st expression;2nd expression
comp_unit;stand2;0;1;nan;nan
comp_unit;stand2;0;0;0.0;nan
comp_unit;stand2;1;0;0.0;nan
comp_unit;stand2;1;1;nan;nan
comp_unit;stand1;0;0;0.0;nan
comp_unit;stand1;1;0;0.0;nan
stratum;stratum1-1;0;0;0.0;nan
stratum;stratum1-1;1;0;0.0;nan
stratum;stratum1-2;0;0;0.0;nan
stratum;stratum1-2;1;0;0.0;nan
stratum;stratum2-3;0;1;nan;nan
stratum;stratum2-3;1;1;nan;nan
stratum;stratum2-2;0;0;0.0;nan
stratum;stratum2-2;1;0;0.0;nan
stratum;stratum2-1;0;0;31.0;nan
stratum;stratum2-1;1;0;31.0;nan
```

### 51.3 def run(self):

Uses data from init to run the class-specific output (self.expression)

### 51.4 def \_run\_expr(self, expr\_dict):

Executes the given expression dictionary for all reporting periods and return the results in a list of lists of strings.

### 51.5 def \_execute\_var(self, discount\_rate, time\_unit, time\_length, var):

Executes the given aggregation set with the given values and returns results as a list of strings.

### 51.6 def \_analyse\_data(self):

Go through expressions and input database and analyse some data properties.

### 51.7 def \_process\_exprs(self, exprs):

Go through the expressions and try to deduce data levels and earliest and latest dates.

## 51.8 def \_create\_tables(self):

Create expression data tables, using PyTables compressed arrays (CArrays) for huge data sets.

## 51.9 def \_fill\_tables(self):

Fill expression data tables with values from input databases.

## 51.10 def \_collect\_values(self, expr, arr, forced\_level=None):

Get values for a single expression operand from either data or operation result database.

Parameters:

```
expr -- subobjective or constraint expression in postfix form
arr -- target PyTables array
```

## 51.11 def \_get\_values\_from\_db(self, operand, dates, norm, opresconst=None, forced\_level=None):

Get value(s) for a single operand from data or operation result database.

Parameters:

```
operand -- expression operand, i.e. variable, value or operator
dates -- date range
norm --
opresconst --
forced_level -- forced level name, i.e. use this level for getting the data
```

## 51.12 def \_eval\_data\_constraint(self, dates, arr, operand, i\_e, norm):

Evaluate 'variable' (data) operand constraint and store only those objects for which the condition evaluated True.

Parameters:

```
dates -- date range
arr -- temporary data array
operand -- current operand
i_e -- expression operator indice as int
norm --
```

## 51.13 def \_eval\_opres\_constraint(self, dates, arr, operand, i\_e, norm):

Evaluate 'variable' (data) operand constraint and store only those objects for which the condition evaluated True.

Parameters:

```
dates -- date range
arr -- temporary data array
operand -- current operand
i_e -- expression operator indice as int
norm --
```

### 51.14 `def _set_value(self, arr, it, uid, br, i_e, date, value):`

Store a single value into a correct location in a matrix and update branch number container.

Parameters:

```
arr -- PyTables/Numpy array
it -- iteration, int
uid -- object id, str
br -- branch, int
i_e -- expression operand index, int
date -- datetime object
value -- value to store, float
```

### 51.15 `def _construct_strings(self):`

Construct strings from expression value table contents for output.

# INLINE.PY

## 52.1 class OutputInline(Output):

Module for data output in hierarchical “data in columns” format. (writing tables to files)

**52.1.1 def \_\_init\_\_(self, datadb, data\_type, id\_list, main\_level, result\_type, output\_filename, output\_constraint=None, default\_decimal\_places=1, archiving=False, result\_padding=True, aggregation\_def=None, expression\_def=None, opres\_vars=None):**

Creates the output class. fully inherited, executes self.run()

```
>>> import datetime
>>> from pprint import pprint
>>> from simo.output.test.init_objects import InitData, TestLogger
>>> idata = InitData()
>>> idata.init()
>>> testdb = idata.testdb
>>> const_obj = idata.const_obj
>>> aggr_obj = idata.aggr_obj
>>> expr_obj = idata.expr_obj
>>> origvars = const_obj.variables
>>> const_obj.variables = {('stratum',2):origvars[('stratum', 2)]}
>>> from simo.output.inline import OutputInline
>>> out = OutputInline(testdb, 'result', ['stand1'], 'stratum',
...                     'optimized',
...                     'output/test/inlined.txt', TestLogger(),
...                     const_obj, 1, False, True, aggr_obj, expr_obj,
...                     ['cash_flow', 'Income', 'Scrapwood', 'Volume',
...                     'BIOMASS_branches', 'BIOMASS_stumps'])
>>> try: # doctest: +ELLIPSIS
...     file = open('output/test/inlined.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
sim unit; branch; iteration; data level;          id; date; SP;  BA;  HgM
stand1;      0;          1;    stratum; stratum1-1; 2009; 1.0; 17.0; 19.0
stand1;      0;          1;    stratum; stratum1-2; 2009; 2.0;  8.0; 18.5
stand1;      0;          0;    stratum; stratum1-1; 2009; 1.0; 17.0; 19.0
stand1;      0;          0;    stratum; stratum1-2; 2009; 2.0;  8.0; 18.5
>>> const_obj.variables = origvars
>>> out = OutputInline(testdb, 'result', ['stand1'], 'stratum',
...                     'optimized',
...                     'output/test/inlined.txt', TestLogger(),
```

```
...         const_obj, 1, False, True, aggr_obj,
...         ['cash_flow', 'Income', 'Scrapwood', 'Volume',
...         'BIOMASS_branches', 'BIOMASS_stumps'])
>>> try: # doctest: +ELLIPSIS
...     file = open('output/test/inlined.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
sim unit; branch; iteration; data level;          id; date; AREA;   SC
stand1;      0;          1; comp_unit;      stand1; 2009;  1.0;  3.0
sim unit; branch; iteration; data level;          id; date;   SP;   BA;  HgM
stand1;      0;          1; stratum; stratum1-1; 2009;  1.0; 17.0; 19.0
sim unit; branch; iteration; data level;          id; date;    d;    h
stand1;      0;          1; tree; tree1-1-1; 2009; 17.0; 18.0
sim unit; branch; iteration; data level;          id; date;   SP;   BA;  HgM
stand1;      0;          1; stratum; stratum1-2; 2009;  2.0;  8.0; 18.5
sim unit; branch; iteration; data level;          id; date;    d;    h
stand1;      0;          1; tree; tree1-2-1; 2009; 22.0; 21.0
sim unit; branch; iteration; data level;          id; date; AREA;   SC
stand1;      0;          0; comp_unit;      stand1; 2009;  1.0;  3.0
sim unit; branch; iteration; data level;          id; date;   SP;   BA;  HgM
stand1;      0;          0; stratum; stratum1-1; 2009;  1.0; 17.0; 19.0
sim unit; branch; iteration; data level;          id; date;    d;    h
stand1;      0;          0; tree; tree1-1-1; 2009; 17.0; 18.0
sim unit; branch; iteration; data level;          id; date;   SP;   BA;  HgM
stand1;      0;          0; stratum; stratum1-2; 2009;  2.0;  8.0; 18.5
sim unit; branch; iteration; data level;          id; date;    d;    h
stand1;      0;          0; tree; tree1-2-1; 2009; 22.0; 21.0
```

### 52.1.2 def run(self):

Uses data from init to run the class-specific output (self.inlined)

### 52.1.3 def \_get\_level\_headers(self, level):

Returns the headers for a given level in datadb

```
>>> headers = out._get_level_headers(3)
>>> headers
['branch', 'iteration', 'id', 'oid', 'date', 'd', 'h']
>>> headers = out._get_level_headers(2, out._get_level_constraints(2))
```

### 52.1.4 def \_get\_level\_constraints(self, level):

Returns the headers to be used for a given level in datadb

```
>>> out._get_level_constraints(1)
[('AREA', 0), ('SC', 1)]
>>> out._get_level_constraints(2)
[('SP', 0), ('BA', 1), ('HgM', 3)]
>>> out._get_level_constraints(3)
[('d', 0), ('h', 1)]
```



### 52.1.5 def \_get\_strings(self, constraints=None, dates=None, date\_res=0, level=1, sim\_unit=None):

Returns a formatted table as a list of rows, starting from the given level, using the given constraints (a dictionary of key: [op, good value, good value, ...] pairs (several values only if op=='in')), date range (dates = (datetime.date(X,Y,Z), datetime.date(X,Y,Z))), date resolution (0 = year, 1 = full date) and simulation unit (this shouldn't be set by the user)

```
>>> result = out._get_strings([('iteration', 'eq', 0)], (datetime.date(2008, 1, 1), datetime.date(2011, 12, 30)), 0, 1, 'h')
>>> pprint(result)
[' sim unit; branch; iteration; data level;          id; date;      d;      h',
 '   test;        0;          0;      tree; tree1-1-1; 2009; 17.0; 18.0',
 '   test;        0;          0;      tree; tree1-2-1; 2009; 22.0; 21.0',
 '   test;        0;          0;      tree; tree2-1-1; 2009; 30.0; 29.0',
 '   test;        0;          0;      tree; tree2-2-1; 2009; 26.0; 24.0',
 '   test;        0;          0;      tree; tree2-2-2; 2009; 28.0; 27.0']
>>> result = out._get_strings([('iteration', 'eq', 0)], (datetime.date(2008, 1, 1), datetime.date(2011, 12, 30)), 0, 1, 'h')
>>> pprint(result)
[' sim unit; branch; iteration; data level;          id;      date;      d;      h',
 '   test;        0;          0;      tree; tree1-1-1; 2009-02-07; 17.0; 18.0',
 '   test;        0;          0;      tree; tree1-2-1; 2009-02-07; 22.0; 21.0',
 '   test;        0;          0;      tree; tree2-1-1; 2009-02-08; 30.0; 29.0',
 '   test;        0;          0;      tree; tree2-2-1; 2009-02-08; 26.0; 24.0',
 '   test;        0;          0;      tree; tree2-2-2; 2009-02-08; 28.0; 27.0']
>>> out._get_strings([('iteration', 'eq', 0)], (datetime.date(2011, 5, 7), datetime.date(2011, 12, 30)), 0, 1, 'h')
[]
>>> out._get_strings([('iteration', 'eq', 0)], (datetime.date(2008, 5, 7), datetime.date(2008, 12, 30)), 0, 1, 'h')
[]
>>> result2 = out._get_strings(dates=(datetime.date(2008, 1, 1), datetime.date(2011, 12, 30)), constraints=None, date_res=0, level=1, sim_unit=None)
>>> for item in out._get_strings():
...     found = False
...     for ritem in result2:
...         if ritem == item:
...             found = True
...     if not found:
...         print item, 'not found!'
```

### 52.1.6 def inlined(self, file, constraints=None, dates=None, date\_res=0):

This function will get the datadb strings (with \_get\_strings, using the given restrictions and date resolution) and print them to the given file.

```
>>> out.inlined('output/test/inlined.txt')
>>> try:
...     file = open('output/test/inlined.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
sim unit; branch; iteration; data level;          id; date; AREA;   SC
stand2;        1;          0; comp_unit;      stand2; 2009;  2.0;  2.0
sim unit; branch; iteration; data level;          id; date;   SP;   BA;   HgM
stand2;        1;          0; stratum; stratum2-3; 2009;  2.0;  0.0;  0.0
sim unit; branch; iteration; data level;          id; date; AREA;   SC
stand1;        0;          1; comp_unit;      stand1; 2009;  1.0;  3.0
sim unit; branch; iteration; data level;          id; date;   SP;   BA;   HgM
stand1;        0;          1; stratum; stratum1-1; 2009;  1.0; 17.0; 19.0
sim unit; branch; iteration; data level;          id; date;      d;      h
stand1;        0;          1;      tree; tree1-1-1; 2009; 17.0; 18.0
sim unit; branch; iteration; data level;          id; date;   SP;   BA;   HgM
```

```
stand1;      0;      1;      stratum; stratum1-2; 2009; 2.0; 8.0; 18.5
sim unit; branch; iteration; data level;      id; date; d; h
stand1;      0;      1;      tree; tree1-2-1; 2009; 22.0; 21.0
sim unit; branch; iteration; data level;      id; date; AREA; SC
stand2;      0;      1; comp_unit; stand2; 2009; 2.0; 2.0
sim unit; branch; iteration; data level;      id; date; SP; BA; HgM
stand2;      0;      1; stratum; stratum2-1; 2009; 2.0; 24.0; 31.0
sim unit; branch; iteration; data level;      id; date; d; h
stand2;      0;      1; tree; tree2-1-1; 2009; 30.0; 29.0
sim unit; branch; iteration; data level;      id; date; SP; BA; HgM
stand2;      0;      1; stratum; stratum2-2; 2009; 3.0; 12.0; 26.0
sim unit; branch; iteration; data level;      id; date; d; h
stand2;      0;      1; tree; tree2-2-1; 2009; 26.0; 24.0
stand2;      0;      1; tree; tree2-2-2; 2009; 28.0; 27.0
sim unit; branch; iteration; data level;      id; date; AREA; SC
stand1;      0;      0; comp_unit; stand1; 2009; 1.0; 3.0
sim unit; branch; iteration; data level;      id; date; SP; BA; HgM
stand1;      0;      0; stratum; stratum1-1; 2009; 1.0; 17.0; 19.0
sim unit; branch; iteration; data level;      id; date; d; h
stand1;      0;      0; tree; tree1-1-1; 2009; 17.0; 18.0
sim unit; branch; iteration; data level;      id; date; SP; BA; HgM
stand1;      0;      0; stratum; stratum1-2; 2009; 2.0; 8.0; 18.5
sim unit; branch; iteration; data level;      id; date; d; h
stand1;      0;      0; tree; tree1-2-1; 2009; 22.0; 21.0
sim unit; branch; iteration; data level;      id; date; AREA; SC
stand2;      0;      0; comp_unit; stand2; 2009; 2.0; 2.0
sim unit; branch; iteration; data level;      id; date; SP; BA; HgM
stand2;      0;      0; stratum; stratum2-1; 2009; 2.0; 24.0; 31.0
sim unit; branch; iteration; data level;      id; date; d; h
stand2;      0;      0; tree; tree2-1-1; 2009; 30.0; 29.0
sim unit; branch; iteration; data level;      id; date; SP; BA; HgM
stand2;      0;      0; stratum; stratum2-2; 2009; 3.0; 12.0; 26.0
sim unit; branch; iteration; data level;      id; date; d; h
stand2;      0;      0; tree; tree2-2-1; 2009; 26.0; 24.0
stand2;      0;      0; tree; tree2-2-2; 2009; 28.0; 27.0
sim unit; branch; iteration; data level;      id; date; AREA; SC
stand2;      1;      1; comp_unit; stand2; 2009; 2.0; 2.0
sim unit; branch; iteration; data level;      id; date; SP; BA; HgM
stand2;      1;      1; stratum; stratum2-3; 2009; 2.0; 0.0; 0.0
```

# OPRES.PY

## 53.1 class OutputOpres(Output):

This module handles writing operation results into files

**53.1.1 def \_\_init\_\_(self, datadb, data\_type, id\_list, main\_level, result\_type, output\_filename, output\_constraint=None, default\_decimal\_places=1, archiving=False, result\_padding=True, aggregation\_def=None, expression\_def=None, opres\_vars=None):**

Creates the output class. fully inherited, executes self.run()

```
>>> from pprint import pprint
>>> from simo.output.test.init_objects import InitData, TestLogger
>>> idata = InitData()
>>> idata.init()
>>> testdb = idata.testdb
>>> const_obj = idata.const_obj
>>> aggr_obj = idata.aggr_obj
>>> expr_obj = idata.expr_obj
>>> from simo.output.opres import OutputOpres
>>> out = OutputOpres(testdb, 'result', ['stand1'], 'stratum',
...                   'optimized',
...                   'output/test/opres.txt', TestLogger(),
...                   const_obj, 1, False, True, aggr_obj, expr_obj,
...                   ['cash_flow', 'Income', 'Scrapwood', 'Volume',
...                   'BIOMASS_branches', 'BIOMASS_stumps', 'assortment',
...                   'SP'])
>>> try: # doctest: +ELLIPSIS
...     file = open('output/test/opres.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
level; object_id; date; iteration; branch;      op_group;  op_name; Volume; assortment;  SP;
comp_unit;  stand1; 2009;          0;      0; final_harvest; clearcut; 123.2;          5.0; 1.0;
comp_unit;  stand1; 2009;          0;      0; final_harvest; clearcut;      ;          ;      ;
comp_unit;  stand1; 2009;          0;      0; regeneration; planting;      ;          ;      ;
```

## 53.1.2 def run(self):

Uses data from init to run the class-specific output (self.opres)

### 53.1.3 def `_get_opres_strings(self, constraints=None, dates=None, date_res=0):`

This function will get the operation results from the datadb and returns them in a formatted list of rows. The constraints, dates and date\_res are as above. Classifiers are prepended to the end of the row, where applicable

```
>>> headers = ['id', 'level', 'object_id', 'date',
...            'iteration', 'branch', 'op_group', 'op_name'] + \
...            ['Volume', 'assortment', 'SP', 'cash_flow']
>>> opres = out._get_opres_strings(headers, constraints=None, dates=None, date_res=0)
>>> pprint(opres)
['      level; object_id; date; iteration; branch;      op_group; op_name; Volume; assortment; SP;
' comp_unit;   stand1; 2009;          0;          0; final_harvest; clearcut; 123.2;          5.0; 1.
' comp_unit;   stand1; 2009;          0;          0; final_harvest; clearcut;          ;          ;
' comp_unit;   stand1; 2009;          0;          0; regeneration; planting;          ;          ;
>>> pprint(out._get_opres_strings(['id', 'level', 'object_id', 'date',
...                               'iteration', 'branch', 'op_group', 'op_name', 'cash_flow'],
...                               constraints=None, dates=None, date_res=0))
['      level; object_id; date; iteration; branch;      op_group; op_name; cash_flow',
' comp_unit;   stand1; 2009;          0;          0; final_harvest; clearcut;          ',
' comp_unit;   stand1; 2009;          0;          0; final_harvest; clearcut; 3456.7',
' comp_unit;   stand1; 2009;          0;          0; regeneration; planting; -456.7']
```

### 53.1.4 def `opres(self, file, constraints=None, dates=None, date_res=0):`

This function will use `_get_opres_strings` with the given restrictions and date resolution, and writes the results into the given file

```
>>> out.opres('test.txt')
>>> lines = []
>>> try:
...     file = open('test.txt', 'r')
...     for line in file:
...         lines.append(line.rstrip('\n'))
... finally:
...     file.close()
>>> lines == opres
True
```

# OUT.PY

## 54.1 class Out(object):

This class executes all the given output types and writes them to the given output files.

**54.1.1** `def __init__(self, datadb, data_type, id_list, main_level, result_type, output_formats, output_filenames, output_constraint=None, default_decimal_places=1, archiving=False, result_padding=True, aggregation_def=None, expression_def=None, opres_vars=None, full_date=False, dates=None):`

Creates the output class

```
>>> from simo.output.test.init_objects import InitData, TestLogger
>>> idata = InitData()
>>> idata.init()
>>> testdb = idata.testdb
>>> const_obj = idata.const_obj
>>> aggr_obj = idata.aggr_obj
>>> expr_obj = idata.expr_obj
>>> from simo.output.out import Out
>>> out = Out(testdb, 'result', ['standl', 'stratum',
...                             'optimized', ['inlined', 'by_level', 'operation_result',
...                             'aggregation', 'branching_graph', 'smt'], 'output',
...                             ['test/inlined.txt',
...                             'test/by_level.txt',
...                             'test/opres.txt',
...                             'test/aggregation.txt',
...                             'test',
...                             'test/smt.txt']], TestLogger(),
...          const_obj, 1, False, True, aggr_obj, expr_obj,
...          ['cash_flow', 'Income', 'Scrapwood', 'Volume',
...          'BIOMASS_branches', 'BIOMASS_stumps', 'assortment',
...          'SP'])
... #doctest: +ELLIPSIS
output info : Writing inlined
output info : finished in ...
output info : Writing by_level
output info : finished in ...
output info : Writing operation_result
output info : finished in ...
output info : Writing aggregation
output info : finished in ...
output info : Writing branching_graph
output info : finished in ...
```

```
output info : Writing smt
smt error : Bad stratum id 'stratum1-1' on sim unit stand1, year 2009 in results
smt error : Bad stratum id 'stratum1-2' on sim unit stand1, year 2009 in results
smt error : Bad stratum id 'stratum1-1' on sim unit stand1, year 2009 in results
smt error : Bad stratum id 'stratum1-2' on sim unit stand1, year 2009 in results
output info : finished in ...
```

```
>>> try: # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
...     file = open('output/test/inlined.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
sim unit; branch; iteration; data level;          id; date; AREA;   SC
stand1;      0;          1; comp_unit;      stand1; 2009; 1.0; 3.0
sim unit; branch; iteration; data level;          id; date;   SP;   BA;   HgM
stand1;      0;          1; stratum; stratum1-1; 2009; 1.0; 17.0; 19.0
sim unit; branch; iteration; data level;          id; date;    d;    h
stand1;      0;          1; tree; tree1-1-1; 2009; 17.0; 18.0
sim unit; branch; iteration; data level;          id; date;   SP;   BA;   HgM
stand1;      0;          1; stratum; stratum1-2; 2009; 2.0; 8.0; 18.5
sim unit; branch; iteration; data level;          id; date;    d;    h
stand1;      0;          1; tree; tree1-2-1; 2009; 22.0; 21.0
sim unit; branch; iteration; data level;          id; date; AREA;   SC
stand1;      0;          0; comp_unit;      stand1; 2009; 1.0; 3.0
sim unit; branch; iteration; data level;          id; date;   SP;   BA;   HgM
stand1;      0;          0; stratum; stratum1-1; 2009; 1.0; 17.0; 19.0
sim unit; branch; iteration; data level;          id; date;    d;    h
stand1;      0;          0; tree; tree1-1-1; 2009; 17.0; 18.0
sim unit; branch; iteration; data level;          id; date;   SP;   BA;   HgM
stand1;      0;          0; stratum; stratum1-2; 2009; 2.0; 8.0; 18.5
sim unit; branch; iteration; data level;          id; date;    d;    h
stand1;      0;          0; tree; tree1-2-1; 2009; 22.0; 21.0
```

```
>>> try: # doctest: +ELLIPSIS
...     file = open('output/test/by_level.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
sim unit; branch; iteration; data level;          id; date; AREA;   SC;   SP;   BA;   HgM;    d;
stand1;      0;          0; comp_unit;      stand1; 2009; 1.0; 3.0;   ;   ;   ;   ;
stand1;      0;          0; stratum; stratum1-1; 2009;   ;   ; 1.0; 17.0; 19.0;   ;
stand1;      0;          0; tree; tree1-1-1; 2009;   ;   ;   ;   ;   ; 17.0; 18.
stand1;      0;          0; stratum; stratum1-2; 2009;   ;   ; 2.0; 8.0; 18.5;   ;
stand1;      0;          0; tree; tree1-2-1; 2009;   ;   ;   ;   ;   ; 22.0; 21.
stand1;      0;          1; comp_unit;      stand1; 2009; 1.0; 3.0;   ;   ;   ;   ;
stand1;      0;          1; stratum; stratum1-1; 2009;   ;   ; 1.0; 17.0; 19.0;   ;
stand1;      0;          1; tree; tree1-1-1; 2009;   ;   ;   ;   ;   ; 17.0; 18.
stand1;      0;          1; stratum; stratum1-2; 2009;   ;   ; 2.0; 8.0; 18.5;   ;
stand1;      0;          1; tree; tree1-2-1; 2009;   ;   ;   ;   ;   ; 22.0; 21.
```

```
>>> try: # doctest: +ELLIPSIS
...     file = open('output/test/opres.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
level; object_id; date; iteration; branch;      op_group; op_name; Volume; assortment; SP;
comp_unit; stand1; 2009;      0;      0; final_harvest; clearcut; 123.2;      5.0; 1.0;
comp_unit; stand1; 2009;      0;      0; final_harvest; clearcut;      ;      ;      ;
comp_unit; stand1; 2009;      0;      0; regeneration; planting;      ;      ;      ;
```

```

>>> try: # doctest: +ELLIPSIS
...     file = open('output/test/aggregation.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
Results for expression (5 keys): sum[-1:-1](stratum:HgM[SP eq 2 and DgM gt 19])
  stratum:SP; stratum:DgM; 08/02/09-08/02/09
    1.0;          20.0;          NaN
    2.0;           0.0;          NaN
    2.0;          18.0;          NaN
    2.0;          30.0;         62.0
    3.0;          27.0;          NaN
Errors (key: stratum:SP, stratum:DgM):
No values for stratum:HgM between 2009-02-08 and 2009-02-08;1.0, 20.0;2.0, 0.0;2.0, 18.0;3.0, 27.0
<BLANKLINE>
Proportions for expression (5 keys): sum[0:99](stratum:SP*stratum:HgM)
  stratum:HgM; stratum:DgM; 01/01/09-08/02/09
    0.0;          0.0;          0.0
    18.5;         18.0;          0.2
    19.0;         20.0;          0.1
    26.0;         27.0;          0.4
    31.0;         30.0;          0.3
Errors (key: stratum:HgM, stratum:DgM):
<BLANKLINE>
Results for expression (1 keys): sum[0:99](operation:Volume)
  01/01/09-08/02/09
    123.2
Errors (key: ):
<BLANKLINE>
Results for expression (1 keys): sum[0:99](operation:cash_flow)
  01/01/09-08/02/09
    3000.0
Errors (key: ):
<BLANKLINE>
Results for expression (2 keys): sum[0:99](operation:cash_flow)
  comp_unit:AREA; 01/01/09-08/02/09
    1.0;          3000.0
    2.0;          NaN
Errors (key: comp_unit:AREA):
No values for operation:cash_flow between 2009-01-01 and 2009-02-08;2.0
<BLANKLINE>
>>> try: # doctest: +ELLIPSIS
...     file = open('output/test/stratum2-1_0.dot', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
digraph branches {
nodesep=0.01;
ranksep=0.01;
size="8.27,11.69";
ratio=fill;
<BLANKLINE>
{
node [shape=plaintext fontsize=14 height=.01];
<BLANKLINE>
2009;
}
<BLANKLINE>
{
node [shape=doublecircle];

```

```
<BLANKLINE>
b_0_2009 [label="0" fontcolor=red group="0"];
b_1_2009 [label="1" fontcolor=red group="1"];
}
<BLANKLINE>
{
node [shape=point]
<BLANKLINE>
b_0_2009 [group="0"];
b_1_2009 [group="1"];
}
<BLANKLINE>
{ rank = same; 2009; b_0_2009; b_1_2009; }
<BLANKLINE>
b_0_2009 -> b_1_2009 [ label="branch_name" fontsize=14 fontcolor=red];
<BLANKLINE>
}
<BLANKLINE>

>>> out = Out(testdb, 'result', ['stand1'], 'comp_unit',
...         'optimized', ['smt'], 'output',
...         ['test/smt.txt'], TestLogger(),
...         const_obj, 1, False, True, aggr_obj,
...         ['cash_flow', 'Income', 'Scrapwood', 'Volume',
...         'BIOMASS_branches', 'BIOMASS_stumps'])
... #doctest: +ELLIPSIS
output info : Writing smt
smt error : Bad stratum id 'stratum1-1' on sim unit stand1, year 2009 in results
smt error : Bad stratum id 'stratum1-2' on sim unit stand1, year 2009 in results
smt error : Bad stratum id 'stratum1-1' on sim unit stand1, year 2009 in results
smt error : Bad stratum id 'stratum1-2' on sim unit stand1, year 2009 in results
output info : finished in ...
>>> try: # doctest:
...     file = open('output/test/smt.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
stand1 stand1 1.0 3.0 2009
stand1 stratum1-1 1.0 17.0 19.0 2009
stand1 stratum1-2 2.0 8.0 18.5 2009
stand1 stand1 1.0 3.0 2009
stand1 stratum1-1 1.0 17.0 19.0 2009
stand1 stratum1-2 2.0 8.0 18.5 2009

>>> try: # doctest:
...     file = open('output/test/expression.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
level;unit;iteration;branch;1st expression;2nd expression
comp_unit;stand2;0;1;nan;nan
comp_unit;stand2;0;0;0.0;nan
comp_unit;stand2;1;0;0.0;nan
comp_unit;stand2;1;1;nan;nan
comp_unit;stand1;0;0;0.0;nan
comp_unit;stand1;1;0;0.0;nan
stratum;stratum1-1;0;0;0.0;nan
stratum;stratum1-1;1;0;0.0;nan
stratum;stratum1-2;0;0;0.0;nan
stratum;stratum1-2;1;0;0.0;nan
```



```
stratum;stratum2-3;0;1;nan;nan  
stratum;stratum2-3;1;1;nan;nan  
stratum;stratum2-2;0;0;0.0;nan  
stratum;stratum2-2;1;0;0.0;nan  
stratum;stratum2-1;0;0;31.0;nan  
stratum;stratum2-1;1;0;31.0;nan
```



# OUTPUT.PY

## 55.1 class Output(object):

A base class for all the output formats. Holds the common functionality.

**55.1.1 def \_\_init\_\_(self, datadb, data\_type, id\_list, main\_level, result\_type, output\_filename, output\_constraint=None, default\_decimal\_places=1, archiving=False, result\_padding=True, aggregation\_def=None, expression\_def=None, opres\_vars=None):**

Creates the output class, saves all the parameters to self and executes self.run()

```
>>> from simo.output.test.init_objects import InitData, TestLogger
>>> idata = InitData()
>>> idata.init()
>>> testdb = idata.testdb
>>> const_obj = idata.const_obj
>>> aggr_obj = idata.aggr_obj
>>> expr_obj = idata.expr_obj
>>> from simo.output.output import Output
>>> out = Output(testdb, 'result', ['stand1'], 'stratum',
...             'optimized',
...             'output/test/test.txt', TestLogger(),
...             const_obj, 1, False, True, aggr_obj, expr_obj,
...             ['cash_flow', 'Income', 'Scrapwood', 'Volume',
...             'BIOMASS_branches', 'BIOMASS_stumps'])
```

### 55.1.2 def run(self):

Uses data from init to run the class-specific output. Does nothing; override in children.

### 55.1.3 def \_pad(self, cols, pad):

This function will format a single row of columns to the column widths defined in pad

```
>>> out._pad(['1', '11', '111', '1111', '11111'], [5, 4, 3, 2, 1])
['1', '11', '111', '1111', '11111']
```

#### 55.1.4 def \_pad\_all(self, rows):

This function will format a two dimensional list of rows (of columns) so that on every row the columns are the same width (the widest column on any row for that index, with a leading space)

```
>>> rows = [['1', '11', '111', '1111', '11111'],
...         ['111', '1', '11', '111'],
...         ['1', '11', '111', '1', '11', '11111']]
>>> for i in out._pad_all(rows): print i
[' 1', ' 11', ' 111', ' 1111', ' 11111']
['111', ' 1', ' 11', ' 111']
[' 1', ' 11', ' 111', ' 1', ' 11', ' 11111']
```

#### 55.1.5 def \_get\_level\_headers(self, level):

Returns the headers for a given level in datadb

```
>>> headers = out._get_level_headers(3)
>>> headers
['branch', 'iteration', 'id', 'oid', 'date', 'd', 'h']
>>> headers = out._get_level_headers(2, out._get_level_constraints(2))
```

#### 55.1.6 def \_get\_level\_constraints(self, level):

Returns the headers to be used for a given level in datadb

```
>>> out._get_level_constraints(1)
[('AREA', 0), ('SC', 1)]
>>> out._get_level_constraints(2)
[('SP', 0), ('BA', 1), ('HgM', 3)]
>>> out._get_level_constraints(3)
[('d', 0), ('h', 1)]
```

#### 55.1.7 def \_write\_file(self, file, strings):

This will write a list of strings to the given file

```
>>> out._write_file('test.txt', ['test', 'piece', 'of', 'text'])
>>> lines = []
>>> try:
...     file = open('test.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
test
piece
of
text
>>> try:
...     os.remove('test.txt')
... except:
...     pass
```

### 55.1.8 `def _listflt2str(self, list, d):`

Turns a list of floats into strings with `d` decimals. If the list contains non-float values, they are converted to strings with `str()`. This method operates in-place and returns nothing.

```
>>> float_list = [1.0, 2.1111111, 3, 4.34235, '5.43abcde']
>>> out._listflt2str(float_list, 3)
>>> float_list
['1.000', '2.111', '3', '4.342', '5.43abcde']
```

### 55.1.9 `def _has_data(self, level_dict):`

Checks if a level keyed dictionary has data other than empty lists in it.

```
>>> out._has_data({})
False
>>> out._has_data({1: []})
False
>>> out._has_data({1: [True]})
True
```



# SMT.PY

## 56.1 class OutputSMT(Output):

Module for data output in SMT file format: values from the last year in the simulation as whitespace separated fields starting with top-level id and object id and ending in the year for the values

**56.1.1 def \_\_init\_\_(self, datadb, data\_type, id\_list, main\_level, result\_type, output\_filename, output\_constraint=None, default\_decimal\_places=1, archiving=False, result\_padding=True, aggregation\_def=None, expression\_def=None, opres\_vars=None):**

Creates the output class. fully inherited, executes self.run()

```
>>> from simo.output.test.init_objects import InitData, TestLogger
>>> from pprint import pprint
>>> idata = InitData()
>>> idata.init()
>>> testdb = idata.testdb
>>> const_obj = idata.const_obj
>>> aggr_obj = idata.aggr_obj
>>> expr_obj = idata.expr_obj
>>> from simo.output.smt import OutputSMT
>>> out = OutputSMT(testdb, 'result', ['stand1'], 'comp_unit',
...                 'optimized',
...                 'output/test/smt.txt', TestLogger(),
...                 const_obj, 1, False, True, aggr_obj, expr_obj,
...                 ['cash_flow', 'Income', 'Scrapwood', 'Volume',
...                 'BIOMASS_branches', 'BIOMASS_stumps'])
... # doctest: +NORMALIZE_WHITESPACE
smt error : Bad stratum id 'stratum1-1' on sim unit stand1, year 2009 in
results
smt error : Bad stratum id 'stratum1-2' on sim unit stand1, year 2009 in
results
smt error : Bad stratum id 'stratum1-1' on sim unit stand1, year 2009 in
results
smt error : Bad stratum id 'stratum1-2' on sim unit stand1, year 2009 in
results
```

## 56.1.2 def run(self):

Uses data from init to run the class-specific output

### 56.1.3 def \_get\_strings(self, constraints=None, dates=None, sim\_unit=None):

This method will get a list of strings from the datadb formatted in the SMT format

```
>>> result = out._get_strings() # doctest: +NORMALIZE_WHITESPACE
smt error : Bad stratum id 'stratum1-1' on sim unit stand1, year 2009 in
results
smt error : Bad stratum id 'stratum1-2' on sim unit stand1, year 2009 in
results
smt error : Bad stratum id 'stratum1-1' on sim unit stand1, year 2009 in
results
smt error : Bad stratum id 'stratum1-2' on sim unit stand1, year 2009 in
results
smt error : Bad stratum id 'stratum2-3' on sim unit stand2, year 2009 in
results
smt error : Bad stratum id 'stratum2-1' on sim unit stand2, year 2009 in
results
smt error : Bad stratum id 'stratum2-2' on sim unit stand2, year 2009 in
results
smt error : Bad stratum id 'stratum2-1' on sim unit stand2, year 2009 in
results
smt error : Bad stratum id 'stratum2-2' on sim unit stand2, year 2009 in
results
smt error : Bad stratum id 'stratum2-3' on sim unit stand2, year 2009 in
results

>>> pprint(result)
['stand1 stand1 1.0 3.0 2009',
 'stand1 stratum1-1 1.0 17.0 19.0 2009',
 'stand1 stratum1-2 2.0 8.0 18.5 2009',
 'stand1 stand1 1.0 3.0 2009',
 'stand1 stratum1-1 1.0 17.0 19.0 2009',
 'stand1 stratum1-2 2.0 8.0 18.5 2009',
 'stand2 stand2 2.0 2.0 2009',
 'stand2 stratum2-3 2.0 0.0 0.0 2009',
 'stand2 stand2 2.0 2.0 2009',
 'stand2 stratum2-1 2.0 24.0 31.0 2009',
 'stand2 stratum2-2 3.0 12.0 26.0 2009',
 'stand2 stand2 2.0 2.0 2009',
 'stand2 stratum2-1 2.0 24.0 31.0 2009',
 'stand2 stratum2-2 3.0 12.0 26.0 2009',
 'stand2 stand2 2.0 2.0 2009',
 'stand2 stratum2-3 2.0 0.0 0.0 2009']
```

### 56.1.4 def \_get\_strings\_items(self, constraints=None, dates=None, level=1, sim\_unit=None):

This method will fetch the items needed for the SMT output and return them as a list of lists of strings

```
>>> items = out._get_strings_items()
smt error : Bad stratum id 'stratum1-1' on sim unit stand1, year 2009 in results
smt error : Bad stratum id 'stratum1-2' on sim unit stand1, year 2009 in results
smt error : Bad stratum id 'stratum1-1' on sim unit stand1, year 2009 in results
smt error : Bad stratum id 'stratum1-2' on sim unit stand1, year 2009 in results
smt error : Bad stratum id 'stratum2-3' on sim unit stand2, year 2009 in results
smt error : Bad stratum id 'stratum2-1' on sim unit stand2, year 2009 in results
smt error : Bad stratum id 'stratum2-2' on sim unit stand2, year 2009 in results
smt error : Bad stratum id 'stratum2-1' on sim unit stand2, year 2009 in results
smt error : Bad stratum id 'stratum2-2' on sim unit stand2, year 2009 in results
smt error : Bad stratum id 'stratum2-3' on sim unit stand2, year 2009 in results
```



```
>>> pprint(items)
[['stand1', 'stand1', '1.0', '3.0', '2009'],
 ['stand1', 'stratum1-1', '1.0', '17.0', '19.0', '2009'],
 ['stand1', 'stratum1-2', '2.0', '8.0', '18.5', '2009'],
 ['stand1', 'stand1', '1.0', '3.0', '2009'],
 ['stand1', 'stratum1-1', '1.0', '17.0', '19.0', '2009'],
 ['stand1', 'stratum1-2', '2.0', '8.0', '18.5', '2009'],
 ['stand2', 'stand2', '2.0', '2.0', '2009'],
 ['stand2', 'stratum2-3', '2.0', '0.0', '0.0', '2009'],
 ['stand2', 'stand2', '2.0', '2.0', '2009'],
 ['stand2', 'stratum2-1', '2.0', '24.0', '31.0', '2009'],
 ['stand2', 'stratum2-2', '3.0', '12.0', '26.0', '2009'],
 ['stand2', 'stand2', '2.0', '2.0', '2009'],
 ['stand2', 'stratum2-1', '2.0', '24.0', '31.0', '2009'],
 ['stand2', 'stratum2-2', '3.0', '12.0', '26.0', '2009'],
 ['stand2', 'stand2', '2.0', '2.0', '2009'],
 ['stand2', 'stratum2-3', '2.0', '0.0', '0.0', '2009']]
```

### 56.1.5 def smt(self, file, constraints=None, dates=None):

This method will fetch the SMT data with the given constraints and writes it to the given file

### 56.1.6 def inlined(self, file, constraints=None, dates=None, date\_res=0):

This function will get the smt strings (with `_get_strings`, using the given restrictions and date resolution) and print them to the given file.

```
>>> out.smt('output/test/smt.txt')
smt error : Bad stratum id 'stratum1-1' on sim unit stand1, year 2009 in results
smt error : Bad stratum id 'stratum1-2' on sim unit stand1, year 2009 in results
smt error : Bad stratum id 'stratum1-1' on sim unit stand1, year 2009 in results
smt error : Bad stratum id 'stratum1-2' on sim unit stand1, year 2009 in results
smt error : Bad stratum id 'stratum2-3' on sim unit stand2, year 2009 in results
smt error : Bad stratum id 'stratum2-1' on sim unit stand2, year 2009 in results
smt error : Bad stratum id 'stratum2-2' on sim unit stand2, year 2009 in results
smt error : Bad stratum id 'stratum2-1' on sim unit stand2, year 2009 in results
smt error : Bad stratum id 'stratum2-2' on sim unit stand2, year 2009 in results
smt error : Bad stratum id 'stratum2-3' on sim unit stand2, year 2009 in results
>>> try:
...     file = open('output/test/smt.txt', 'r')
...     for line in file:
...         print line.rstrip('\n')
... finally:
...     file.close()
stand1 stand1 1.0 3.0 2009
stand1 stratum1-1 1.0 17.0 19.0 2009
stand1 stratum1-2 2.0 8.0 18.5 2009
stand1 stand1 1.0 3.0 2009
stand1 stratum1-1 1.0 17.0 19.0 2009
stand1 stratum1-2 2.0 8.0 18.5 2009
stand2 stand2 2.0 2.0 2009
stand2 stratum2-3 2.0 0.0 0.0 2009
stand2 stand2 2.0 2.0 2009
stand2 stratum2-1 2.0 24.0 31.0 2009
stand2 stratum2-2 3.0 12.0 26.0 2009
stand2 stand2 2.0 2.0 2009
stand2 stratum2-1 2.0 24.0 31.0 2009
stand2 stratum2-2 3.0 12.0 26.0 2009
```

```
stand2 stand2 2.0 2.0 2009
stand2 stratum2-3 2.0 0.0 0.0 2009
```

Simulation modules:

# CONDEVAL.PY

## 57.1 class ConditionEvaluator(object):

SIMO simulator condition evaluator:

```
>>> import numpy
>>> import datetime as dt
>>> from minimock import Mock
>>> from simo.simulation.condeval import ConditionEvaluator
>>> add_error = Mock('simulator.add_error') # mock error logging method
>>> ce = ConditionEvaluator(add_error)
```

### 57.1.1 def evaluate(self, sim, tind, expr, level, main\_level, depth, opmem, throughlevel=None):

Evaluate conditional expression. Values are retrieved from data matrix using simo.builder.matrix.handler.Handler instance.:

```
>>> data = Mock('Data')
>>> data.get_tind.mock_returns = None, set([])
>>> data.get_value.mock_returns = numpy.arange(5, dtype=float), None
>>> ret = [dt.date(2000,1,1) for i in range(5)]
>>> data.get_date.mock_returns = numpy.array(ret, dtype=dt.date)
>>> data.get_object_map.mock_returns = numpy.array([False, False, False,
...                                              False],
...                                              dtype=bool)
>>> opmem = Mock('OperationMemory')
>>> opmem.since.mock_returns = numpy.array([0,11,5,10,19], dtype=int)
>>> opmem.times.mock_returns = numpy.array([1,0,1,2,1], dtype=int)
```

Import chainfuncs from builder

```
>>> from simo.builder.modelchain.chainfunc import *
```

Evaluate condition 'comp\_unit:Age gt 1.0':

```
>>> expr = [('variable', (1, 1, True)), ('value', 1.0), ('eq', gte)]
>>> ce.evaluate(data, None, expr, 1, 1, 0, opmem)
Called Data.get_active()
Called Data.set_active(None)
Called Data.get_value(None, 1)
array([False, False,  True,  True,  True], dtype=bool)
```

Evaluate condition 'comp\_unit:Age eq comp\_unit:dummy\_age'

```
>>> expr = [('variable', (1, 1, True)),
...         ('variable', (1, 1, True)),
...         ('eq', eee)]
>>> ce.evaluate(data, None, expr, 1, 1, 0, opmem)
Called Data.get_active()
Called Data.set_active(None)
Called Data.get_value(None, 1)
Called Data.set_active(None)
Called Data.get_value(None, 1)
array([ True,  True,  True,  True,  True], dtype=bool)
```

Evaluate condition 'comp\_unit:thinning times\_eq 1.0'

```
>>> expr = [('operation', 1), ('value', 1.0), ('times', eee)]
>>> ce.evaluate(data, None, expr, 1, 1, 0, opmem)
Called Data.get_active()
Called OperationMemory.times(None, 1)
array([ True, False,  True, False,  True], dtype=bool)
```

Evaluate condition 'comp\_unit:thinning since\_gt 10.0'

```
>>> expr = [('operation', 1), ('value', 10), ('since', sgt)]
>>> ce.evaluate(data, None, expr, 1, 1, 0, opmem)
Called Data.get_active()
Called Data.get_date(None)
Called OperationMemory.since(None, 1, array([2000, 2000, 2000, 2000, 2000]))
array([False,  True, False, False,  True], dtype=bool)
```

Evaluate condition 'comp\_unit:thinning since\_gt 10.0 and comp\_unit:thinning times\_eq 1.0'

```
>>> expr = [('operation', 1), ('value', 10), ('since', sgt), \
...         ('operation', 1), ('value', 1), ('times', eee),
...         ('group', and_)]
>>> ce.evaluate(data, None, expr, 1, 1, 0, opmem)
Called Data.get_active()
Called Data.get_date(None)
Called OperationMemory.since(None, 1, array([2000, 2000, 2000, 2000, 2000]))
Called OperationMemory.times(None, 1)
array([False, False, False, False,  True], dtype=bool)
```

Evaluate condition 'comp\_unit:thinning since\_gt 10.0 or comp\_unit:thinning times\_eq 1.0'

```
>>> expr = [('operation', 1), ('value', 10), ('since', sgt),
...         ('operation', 1), ('value', 1), ('times', eee),
...         ('group', or_)]
>>> ce.evaluate(data, None, expr, 1, 1, 0, opmem)
Called Data.get_active()
Called Data.get_date(None)
Called OperationMemory.since(None, 1, array([2000, 2000, 2000, 2000, 2000]))
Called OperationMemory.times(None, 1)
array([ True,  True,  True, False,  True], dtype=bool)
```

Evaluate condition 'comp\_unit exists'

```
>>> expr = [('level', 1), None, ('ex', ext)]
>>> tind = numpy.array([[0,0,1,0,0],
...                    [0,0,2,0,0],
...                    [0,0,3,0,0],
...                    [0,0,6,0,0]], dtype=int)
>>> ce.evaluate(data, tind, expr, 1, 1, 0, opmem)
Called Data.get_active()
```

```

Called Data.set_active(None)
Called Data.get_object_map(
    1,
    array([[0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0],
           [0, 0, 3, 0, 0],
           [0, 0, 6, 0, 0]]))
array([False, False, False, False], dtype=bool)

```

Evaluate condition 'stratum:BA not exists'

```

>>> expr = [('variable', (1,1,True)), None, ('ex', n_ext)]
>>> ce.evaluate(data, None, expr, 1, 1, 0, opmem)
Called Data.get_active()
Called Data.set_active(None)
Called Data.get_exists(None, 1)
False

```

Evaluate condition 'tree:d gt self:d'

```

>>> expr = [('variable', (2, 1, True)),
...         ('variable', (2, 1, False)),
...         ('eq', gte)]
>>> ce.evaluate(data, None, expr, 1, 1, 0, opmem)
Called Data.get_active()
Called Data.get_tind(1, 0, 2, None, True, None)
Called Data.get_active()
Called Data.get_value(None, 1)
Called Data.get_tind(1, 0, 2, None, False, None)
Called Data.get_active()
Called Data.get_value(None, 1)
array([False, False, False, False], dtype=bool)

```

Evaluate condition 'tree:d ex'

```

>>> expr = [('variable', (2, 1, True)),
...         None,
...         ('ex', ext)]
>>> ce.evaluate(data, None, expr, 1, 1, 0, opmem)
Called Data.get_active()
Called Data.get_tind(1, 0, 2, None, True, None)
Called Data.get_active()
Called Data.get_exists(None, 1)

```

Evaluate condition 'function:<mock\_func>'

```

>>> tind = numpy.array([[0, 0, 0, 0, 0], [0, 0, 1, 0, 0]])
>>> expr = [('function', Mock('mock_func'))]
>>> str(ce.evaluate(data, tind, expr, 1, 1, 0, opmem))
Called Data.get_active()
Called mock_func(2)
'None'

```

Evaluate condition 'function:random\_number\_0\_1'

```

>>> expr = [('function', functions['random_number_0_1'])]
>>> result = ce.evaluate(data, tind, expr, 1, 1, 0, opmem)
Called Data.get_active()
>>> for i in result: 0.0 <= i <= 1.0
True
True

```

```
>>> result # doctest: +ELLIPSIS
array([ ..., ...])
```

Evaluate condition 'function:random\_number\_0\_1 gt 0.0'

```
>>> expr = [('function', functions['random_number_0_1']),
...         ('value', 0.0),
...         ('eq', gte)]
>>> ce.evaluate(data, tind, expr, 1, 1, 0, opmem)
Called Data.get_active()
array([ True,  True], dtype=bool)
```

Evaluate condition so that data handler returns errors

```
>>> tind1 = numpy.array([[0, 0, 0, 0, 1], [0, 0, 1, 1, 2]], dtype=int)
>>> tind2 = numpy.array([[0, 0, 0, 0, 1], [0, 0, 1, 1, 2],
...                      [0, 0, 2, 2, 3], [0, 0, 3, 3, 4]], dtype=int)
>>> err = [(numpy.arange(5, dtype=float), (('error message', [0])),
...         tind1, [0])),
...         (numpy.arange(5, dtype=float), (('error message', [1,3])),
...         tind2, [1,3]))]
>>> data = Mock('Data')
>>> data.get_value.mock_returns_iter = iter(err)
>>> data.get_tind.mock_returns = (None, set([1]),)
>>> expr = [('variable', (1, 1, True)),
...         ('variable', (1, 1, True)),
...         ('eq', eee)]
>>> ce.evaluate(data, 'tind', expr, 1, 1, 0, opmem)
... # doctest: +NORMALIZE_WHITESPACE
Called Data.get_active()
Called Data.set_active(None)
Called Data.get_value('tind', 1)
Called Data.set_active(None)
Called Data.get_value('tind', 1)
Called simulator.add_error(
    'error message when evaluating condition',
    None,
    array([[0, 0, 0, 0, 1]]))
Called simulator.add_error(
    'error message when evaluating condition',
    None,
    array([[0, 0, 1, 1, 2],
          [0, 0, 3, 3, 4]]))
array([False, False,  True, False,  True], dtype=bool)
```

# SIM.PY

## 58.1 class Simulator(object):

SIMO simulator module.

Attributes: - `_logger`: simulation log, logger instance - `_log_name`: log name, string - `lexicon`: simulation lexicon, `Lexicon` instance - `_ctrl`: simulation main controls, `SimControl` instance - `_callers`: model and table caller instances, dictionary with model/table type as key and caller instance as value - `_evaluator`: condition evaluator, `ConditionEvaluator` instance - `_input_db`: input database: `DataDB` instance - `_data_db`: data database: `DataDB` instance - `_operation_db`: operation database: `OperationDB` instance - `data`: data matrix handler, `Handler` instance - `_iterations`: number of iterations, integer - `period`: current period, integer - `timespan`: current time span, `Timespan` object - `chain_index`: current model chain index, integer - `_chain_type`: current model chain type: 'init', 'loop' or 'operation' - `_chains`: model chains of the current time span in a dictionary, chain type as key, list of chains as value - `_forced_operations`: forced operations for current simulation units, dictionary - `level`: current evaluation level, integer - `main_level`: simulation main level, integer - `_tasks`: currently active tasks in a list - `task`: currently active task object - `value_fixing`: value fixing, boolean - `_block_growth`: target index of objects for which growth is blocked - `_block_all`: target index of objects that are totally blocked - `terminate`: terminate simulation, boolean

Some of the caller classes need to access certain simulator data attributes. This access is granted with properties that reveal the needed attributes.

Properties: - `operation_memory`

```
58.1.1 def __init__(self, logger, logname, lexicon, ctrl, input_db, data_db,
                    operation_db, warnings, maxunitcount, obj_count_multiplier,
                    branch_count, no_op_branch, branch_limit, max_res_objs, max_errors,
                    max_simunit_errors, deterministic, track_prices):
```

```
>>> import numpy as np
>>> import datetime as dt
>>> from minimock import Mock
>>> execfile('simulation/test/mocks4sim.py')
```

Construct simulator instance:

```
>>> from simo.simulation.sim import Simulator
>>> sim = Simulator(logger, 'test-log', 'simulation', lexicon, ctrl,
...               input_db, data_db, operation_db, True, 11, 1, 1,
...               False, 100, 50, 10, 1, False, True)
```

```
58.1.2 def _reset(self):
```

Reset simulator, erase all information about past simulation and set all settings to default.

### 58.1.3 def simulate(self, iterations):

Run simulation with current simulation setup and data matrix, but substitute data handler and evaluator at this point with mock objects

```
>>> sim.data = data
>>> sim.data.matrix.iterations = 1
>>> sim._evaluator = evaluator
>>> sim.simulate(1) # doctest: +ELLIPSIS
Called InputDB.get_ids('data', 1)
Called Logger.log_message(
    'test-log',
    'INFO',
    '6 simulation units divided into 1 sets')
Called OperationDB.conn.execute('SELECT max(op_id) FROM forced_operation')
Called MockDBCursor.fetchone()
Called Handler.reset()
Called Logger.log_message(
    'test-log',
    'INFO',
    'filling simulation data matrix...')
Called InputDB.fill_matrix(
    <Mock ... Handler>,
    1,
    [u'1', u'2', u'3', u'4', u'5', '6'])
Called Logger.log_message('test-log', 'INFO', 'matrix filling complete')
Called Handler.get_tind(1, 0)
Called Handler.get_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
Called Handler.set_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    array([2000-01-01, 2000-01-01, 2000-01-01], dtype=object))
Called Handler.set_value(
    1,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    3,
    array([ 2000.,  2000.,  2000.]))
Called Logger.log_message(
    'test-log',
    'INFO',
    'simulating iterations 1..1 for set 1/1 with 6 units')
Called Handler.get_tind(0, 0)
Called Handler.set_value(
    0,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    2,
    5)
Called InputDB.get_level_ind(<Mock ... ModelChain.evaluate_at>)
Called Handler.get_tind(None, 0)
Called Evaluator.evaluate(
    <Mock ... Handler>,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
```



```

    <Mock ... ModelChain.condition>,
    None,
    1,
    0,
    <simo.simulation.caller.operationmemory.OperationMemory object at ...>)
Called Handler.block(None, 0, array([[0, 0, 1, 0, 0],
    [0, 0, 2, 0, 0]]))
Called Handler.release(
    None,
    0,
    array([[0, 0, 1, 0, 0],
    [0, 0, 2, 0, 0]]))
Called Handler.get_tind(1, 0)
Called Evaluator.evaluate(
    <Mock ... Handler>,
    array([[0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 2, 0, 0]]),
    <Mock ... Ctrl.stop_logic>,
    1,
    1,
    0,
    <simo.simulation.caller.operationmemory.OperationMemory object at ...>)
Called Handler.block(1, 0, array([[0, 0, 0, 0, 0]]), True)
Called Handler.get_tind(1, 0)
Called Handler.get_date(
    array([[0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 2, 0, 0]]))
Called InputDB.get_level_ind(<Mock ... ModelChain.evaluate_at>)
Called Handler.get_tind(None, 0)
Called Evaluator.evaluate(
    <Mock ... Handler>,
    array([[0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 2, 0, 0]]),
    <Mock ... ModelChain.condition>,
    None,
    1,
    0,
    <simo.simulation.caller.operationmemory.OperationMemory object at ...>)
Called Handler.block(None, 0, array([[0, 0, 1, 0, 0],
    [0, 0, 2, 0, 0]]))
Called Handler.release(
    None,
    0,
    array([[0, 0, 1, 0, 0],
    [0, 0, 2, 0, 0]]))
Called InputDB.get_level_ind(<Mock ... ModelChain.evaluate_at>)
Called Handler.get_tind(None, 0)
Called Evaluator.evaluate(
    <Mock ... Handler>,
    array([[0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 2, 0, 0]]),
    <Mock ... ModelChain.condition>,
    None,
    1,
    0,
    <simo.simulation.caller.operationmemory.OperationMemory object at ...>)
Called Handler.block(None, 0, array([[0, 0, 1, 0, 0],
    [0, 0, 2, 0, 0]]))

```

```
Called Handler.release(  
    None,  
    0,  
    array([[0, 0, 1, 0, 0],  
           [0, 0, 2, 0, 0]]))  
Called Handler.get_tind(1, 0)  
Called Handler.get_date(  
    array([[0, 0, 0, 0, 0],  
           [0, 0, 1, 0, 0],  
           [0, 0, 2, 0, 0]]))  
Called Handler.set_date(  
    array([[0, 0, 0, 0, 0],  
           [0, 0, 1, 0, 0],  
           [0, 0, 2, 0, 0]]),  
    array([2004-12-31, 2004-12-31, 2004-12-31], dtype=object))  
Called DataDB.add_data_from_matrix(  
    <Mock ... Handler.matrix.dmx>,  
    <Mock ... Handler.matrix.mx>,  
    <Mock ... Handler.ind2id>,  
    <Mock ... Handler.linkage.links>,  
    set([(0, 0, 0)]),  
    1,  
    False,  
    0)  
Called DataDB.add_branching_info(  
    1,  
    <Mock ... Handler.brancher.branching_info>,  
    <Mock ... Handler.ind2id.ids>,  
    0)  
Called Handler.brancher.reset_binfo()  
Called Handler.get_tind(1, 0)  
Called Handler.get_date(  
    array([[0, 0, 0, 0, 0],  
           [0, 0, 1, 0, 0],  
           [0, 0, 2, 0, 0]]))  
Called Handler.set_date(  
    array([[0, 0, 0, 0, 0],  
           [0, 0, 1, 0, 0],  
           [0, 0, 2, 0, 0]]),  
    array([2000-01-02, 2000-01-02, 2000-01-02], dtype=object))  
Called Handler.set_value(  
    1,  
    array([[0, 0, 0, 0, 0],  
           [0, 0, 1, 0, 0],  
           [0, 0, 2, 0, 0]]),  
    3,  
    array([ 2000., 2000., 2000.]))  
Called Handler.release(None, None, None, False, True)  
Called Logger.log_message('test-log', 'INFO', 'set simulated in 00:00:0...')  
Called Logger.log_message('test-log', 'INFO', 'avg. time per unit 00:00:0...')  
Called Logger.log_message('test-log', 'INFO', 'avg. time per set 00:00:0...')  
Called Logger.log_message('test-log', 'INFO', 'total time 00:00:0...')
```

NB: The last call to `Handler.set_date` adds a single day to to initial dates. As the `Handler` is a mock object the previous call didn't really change the dates, hence the year discrepancy between the calls.

#### 58.1.4 `def _evaluate_steps(self, forceops):`

Evaluate number of timesteps with possible stopping conditions, simulation chains, forced operations and/or operation chains and update simulator state:

```

>>> sim._evaluate_steps(False) # doctest: +ELLIPSIS
Called Handler.get_tind(1, 0)
Called Evaluator.evaluate(
    <Mock ... Handler>,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    <Mock ... Ctrl.stop_logic>,
    1,
    1,
    0,
    <simo.simulation.caller.operationmemory.OperationMemory object at ...>)
Called Handler.block(1, 0, array([[0, 0, 0, 0, 0]]), True)
Called InputDB.get_level_ind(<Mock ... ModelChain.evaluate_at>)
Called Handler.get_tind(None, 0)
Called Evaluator.evaluate(
    <Mock ... Handler>,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    <Mock ... ModelChain.condition>,
    None,
    1,
    0,
    <simo.simulation.caller.operationmemory.OperationMemory object at ...>)
Called Handler.block(None, 0, array([[0, 0, 1, 0, 0],
                                     [0, 0, 2, 0, 0]]))
Called Handler.release(
    None,
    0,
    array([[0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
Called InputDB.get_level_ind(<Mock ... ModelChain.evaluate_at>)
Called Handler.get_tind(None, 0)
Called Evaluator.evaluate(
    <Mock ... Handler>,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    <Mock ... ModelChain.condition>,
    None,
    1,
    0,
    <simo.simulation.caller.operationmemory.OperationMemory object at ...>)
Called Handler.block(None, 0, array([[0, 0, 1, 0, 0],
                                     [0, 0, 2, 0, 0]]))
Called Handler.release(
    None,
    0,
    array([[0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
Called Handler.get_tind(1, 0)
Called Handler.get_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
Called Handler.set_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    array([2004-12-31, 2004-12-31, 2004-12-31], dtype=object))
Called DataDB.add_data_from_matrix(

```

```
<Mock ... Handler.matrix.dmx>,
<Mock ... Handler.matrix.mx>,
<Mock ... Handler.ind2id>,
<Mock ... Handler.linkage.links>,
set([(0, 0, 0)]),
1,
False,
0)
Called DataDB.add_branching_info(
1,
<Mock ... Handler.brancher.branching_info>,
<Mock ... Handler.ind2id.ids>,
0)
Called Handler.brancher.reset_binfo()
Called Handler.get_tind(1, 0)
Called Handler.get_date(
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0]]))
Called Handler.set_date(
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0]]),
array([2000-01-02, 2000-01-02, 2000-01-02], dtype=object))
Called Handler.set_value(
1,
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0]]),
3,
array([ 2000.,  2000.,  2000.]))
```

### 58.1.5 def \_evaluate\_chains(self, chaintype, ind=None):

The model chains are evaluated so that the method goes through the top level ‘task’ elements and calls a method to process each of the task elements. chaintype is one of ‘init’, ‘simulation’, ‘operation’ or ‘forced\_operation’.

### 58.1.6 def \_evaluate\_task(self, task, depth):

Method evaluates a task element of a model chain and moves on recursively to any subtasks if there are subtasks. If condition element is found, evaluate\_expression Method is called and if model element is found, evaluate\_model Method is called.

Evaluate a task with no condition, model or subtasks:

```
>>> task = Mock('Task')
>>> task.condition = None
>>> task.model = None
>>> task.task_path = 'path/of/this/task'
>>> task.subtasks = []
>>> task.task.task_path = 'path/of/this/task'
>>> sim._evaluate_task(task, 0)
```

Evaluate a task with condition:

```
>>> task.condition = Mock('Expression')
>>> sim._evaluate_task(task, 0) # doctest: +ELLIPSIS
Called Handler.get_tind(None, 0)
Called Evaluator.evaluate(
```

```

    <Mock ... Handler>,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    <Mock ... Expression>,
    None,
    1,
    0,
    <simo.simulation.caller.operationmemory.OperationMemory object at ...>)
Called Handler.block(None, 0, array([[0, 0, 1, 0, 0],
                                     [0, 0, 2, 0, 0]]))
Called Handler.release(
    None,
    0,
    array([[0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))

```

Evaluate a task with subtasks:

```

>>> task = Mock('Task')
>>> task.condition = None
>>> task.model = None
>>> task.task_path = 'path/of/this/task'
>>> subtask_1 = Mock('Task')
>>> subtask_1.condition = None
>>> subtask_1.model = None
>>> subtask_1.task_path = 'path/of/this/task'
>>> subtask_1.subtasks = []
>>> subtask_2 = Mock('Task')
>>> subtask_2.condition = None
>>> subtask_2.model = None
>>> subtask_2.task_path = 'path/of/this/task'
>>> subtask_2.subtasks = [subtask_1, subtask_1]
>>> task.subtasks = [subtask_1, subtask_2]
>>> sim._evaluate_task(task, 0) # doctest: +ELLIPSIS

```

### 58.1.7 def \_evaluate\_subtasks(self, task, depth):

Evaluate a subtask and all it's possible subtasks. In this example the task is a so-called 'branch\_task', which means that it is processed in a bit different way than normal task:

```

>>> task = Mock('Task')
>>> task.type = 'branch_task'
>>> task.task_path = 'path/of/this/task'
>>> task.condition = None
>>> subtask = Mock('Task')
>>> subtask.model = None
>>> subtask.task_path = 'path/of/this/task'
>>> subtask.condition = None
>>> subtask.subtasks = []
>>> sim._tasks = [task]
>>> task.subtasks = [subtask]
>>> task.branch_conditions = [('branch1', 'C1'), ('branch2', 'C2')]
>>> task.branching_group = 'bgroup1'
>>> sim._evaluate_subtasks(task, 0) # doctest: +ELLIPSIS
Called Handler.get_tind(None, 1)
Called Evaluator.evaluate(
    <Mock ... Handler>,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],

```

```
        [0, 0, 2, 0, 0])),
    'C1',
    None,
    1,
    1,
    <simo.simulation.caller.operationmemory.OperationMemory object at ...>)
Called Handler.block(None, 1, array([[0, 0, 1, 0, 0],
    [0, 0, 2, 0, 0]]))
Called Handler.brancher.start(
    <Mock ... Task>,
    'branch1',
    <Mock ... Handler>,
    1,
    1,
    <simo.simulation.caller.predictionmemory.PredictionModelMemory object at ...>,
    <simo.simulation.caller.operationmemory.OperationMemory object at ...>)
Called Handler.brancher.stop(
    'branch1',
    <Mock ... Handler>,
    <Mock ... Timespan>,
    1,
    1,
    <simo.simulation.caller.predictionmemory.PredictionModelMemory object at ...>,
    <simo.simulation.caller.operationmemory.OperationMemory object at ...>)
Called Handler.release(
    None,
    1,
    array([[0, 0, 1, 0, 0],
    [0, 0, 2, 0, 0]]))
Called Handler.get_tind(None, 1)
Called Evaluator.evaluate(
    <Mock ... Handler>,
    array([[0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 2, 0, 0]]),
    'C2',
    None,
    1,
    1,
    <simo.simulation.caller.operationmemory.OperationMemory object at ...>)
Called Handler.block(None, 1, array([[0, 0, 1, 0, 0],
    [0, 0, 2, 0, 0]]))
Called Handler.brancher.start(
    <Mock ... Task>,
    'branch2',
    <Mock ... Handler>,
    1,
    1,
    <simo.simulation.caller.predictionmemory.PredictionModelMemory object at ...>,
    <simo.simulation.caller.operationmemory.OperationMemory object at ...>)
Called Handler.brancher.stop(
    'branch2',
    <Mock ... Handler>,
    <Mock ... Timespan>,
    1,
    1,
    <simo.simulation.caller.predictionmemory.PredictionModelMemory object at ...>,
    <simo.simulation.caller.operationmemory.OperationMemory object at ...>)
Called Handler.release(
    None,
    1,
    array([[0, 0, 1, 0, 0],
```

```
[0, 0, 2, 0, 0]]))
```

### 58.1.8 def \_evaluate\_model(self, model, depth):

Evaluate a model stored in a model chain. Models can be of types: 'prediction', 'aggregation', 'operation', 'management', 'geo\_table' and 'parameter\_table'.

### 58.1.9 def \_evaluate\_expression(self, expr, level, depth=0):

Evaluate a condition expression. Expressions are in postfix deque items and are evaluated with a postfix evaluator. Return target index of all blocked objects and boolean value True if all evaluation level objects are blocked and False if not.

### 58.1.10 def \_evaluate\_stop\_condition(self):

Evaluate simulation stopping condition and stop simulation if condition evaluates True for all simulation level objects. Block the objects for which the condition was met:

```
>>> sim._evaluator.evaluate.mock_returns = np.array([True, False, True],
...                                                dtype=bool)
>>> sim._evaluate_stop_condition() # doctest: +ELLIPSIS
Called Handler.get_tind(1, 0)
Called Evaluator.evaluate(
    <Mock ... Handler>,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    <Mock ... Ctrl.stop_logic>,
    1,
    1,
    0,
    <simo.simulation.caller.operationmemory.OperationMemory object at ...>)
Called Handler.block(
    1,
    0,
    array([[0, 0, 0, 0, 0],
           [0, 0, 2, 0, 0]]),
    True)
False
```

### 58.1.11 def \_evaluate\_end\_date(self):

Check if any of the main level objects have passed the simulation ending date. Block objects that have passed the ending date:

```
>>> sim.data.get_date.mock_returns = np.array([
...     dt.date(2001,1,1),
...     dt.date(1999,1,1),
...     dt.date(2000,1,1)], dtype=dt.date)
>>> sim.timespan.ending.target = dt.date(2000,1,1)
>>> sim._evaluate_end_date() # doctest: +ELLIPSIS
Called Handler.get_tind(1, 0)
Called Handler.get_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
```

```
Called Handler.block(1, 0, array([[0, 0, 0, 0, 0]]), True)
False
>>> sim._block_all
array([[0, 0, 0, 0, 0]])

>>> sim.timespan.ending.target = dt.date(2002,1,1)
>>> sim._evaluate_end_date() # doctest: +ELLIPSIS
Called Handler.get_tind(1, 0)
Called Handler.get_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
False
>>> sim._block_all

>>> sim.timespan.ending.target = dt.date(1998,1,1)
>>> sim._evaluate_end_date() # doctest: +ELLIPSIS
Called Handler.get_tind(1, 0)
Called Handler.get_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
True
>>> sim._block_all
```

### 58.1.12 def \_do\_grow(self):

Check if growth should be calculated for the current time step, ie. should the ‘loop’ model chains be evaluated, and block objects for which the growth simulation should be passed:

```
>>> sim.period = 0
>>> sim._ctrl.growth_season_end_month = 7
>>> sim._ctrl.growth_season_end_day = 1
>>> sim.data.get_date.mock_returns = np.array([
...     dt.date(2000,6,30),
...     dt.date(2000,7,2),
...     dt.date(1999,7,2)], dtype=dt.date)
>>> sim._do_grow()
Called Handler.get_tind(1, 0)
Called Handler.get_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
Called Handler.block(
    1,
    0,
    array([[0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    True)
True
>>> sim._block_growth
array([[0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0]])

>>> sim.data.get_date.mock_returns = np.array([
...     dt.date(2000,7,2),
...     dt.date(1999,7,2),
...     dt.date(2010,7,2)], dtype=dt.date)
>>> sim._reset()
>>> sim._do_grow()
```



```

Called Handler.get_tind(1, 0)
Called Handler.get_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
False
>>> sim._block_growth

```

### 58.1.13 def \_do\_force\_operations(self, timespan):

Check if forced operations should be done to any of the current simulation units during the current timespan

```

>>> sim._operation_db.fill_dictionary.mock_returns = {
...     u'2': 'forced operations for unit 2',
...     u'3': 'forced operations for unit 3',
...     u'5': 'forced operations for unit 5'}
>>> sim.lexicon.get_level_name.mock_returns = u'comp_unit'
>>> sim.data.get_id.mock_returns = [u'2', u'3', u'4']
>>> sim.data.get_tind.mock_returns = (np.array([[0, 0, 0, 0, 0],
...                                           [0, 0, 1, 0, 0],
...                                           [0, 0, 2, 0, 0]],
...                                           dtype=int), None)
>>> timespan.forced_operations = True
>>> chain1 = Mock('Chain')
>>> chain2 = Mock('Chain')
>>> chain3 = Mock('Chain')
>>> chain1.name = 'Chain one'
>>> chain2.name = 'Chain two'
>>> chain3.name = 'Chain three'
>>> mcc = Mock('ModelChainCollection')
>>> mcc.chains = [chain1, chain2]
>>> mcc2 = Mock('ModelChainCollection')
>>> mcc2.chains = [chain3]
>>> timespan.chain_collections['forced_operation'] = [mcc, mcc2]
>>> sim._do_force_operations(timespan)
Called Lexicon.get_level_name(1)
Called OperationDB.fill_dictionary(
    u'comp_unit',
    [u'1', u'2', u'3', u'4', u'5', '6'])
Called Handler.get_tind(1, 0)
Called Handler.get_id(
    1,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
Called Handler.get_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
True

>>> sim._forced_operations[u'2']
'forced operations for unit 2'
>>> sim._forced_operations[u'3']
'forced operations for unit 3'
>>> sim._forced_operations[u'5']
'forced operations for unit 5'

>>> sim._forced_chain_index
{'Chain two': 1, 'Chain one': 0}, {'Chain three': 0}

```

### 58.1.14 `def _evaluate_forced_operations(self):`

Evaluate the forced operations for the current units. Forced operations can be operation that actually have happened when simulating from past to present (updating simulations) or operations can be forced to simulation units for some other reason

```
>>> sim.data.get_date.mock_returns = np.array([
...     dt.date(2005,7,2),
...     dt.date(2004,7,2),
...     dt.date(2015,7,2)], dtype=dt.date)
>>> fop = Mock('ForcedOperation')
>>> fop.id = u'2'
>>> fop.level = u'comp_unit'
>>> fop.name = u'clearcut'
>>> fop.timing_type = 'step'
>>> fop.time_step = 1
>>> fop.date = dt.date(2006,1,1)
>>> fop.step_unit = 'year'
>>> fop.chain_names = ['Chain two', 'Chain three']
>>> fop.chain_indices = None
>>> fop.parameters = None
>>> sim._forced_operations = {(0,0,u'2'): [fop]}
>>> sim.data.get_id.mock_returns = ['2','5','6']
>>> sim._evaluate_chains = Mock('mock_evaluate_chains')

>>> sim._evaluate_forced_operations()
Called Handler.get_tind(1, 0)
Called Handler.get_id(
    1,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
Called Handler.get_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
Called Handler.block(
    1,
    0,
    array([[0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    True)
Called mock_evaluate_chains('forced_operation', [[1], [0]])
Called Handler.release(
    1,
    0,
    array([[0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    True,
    False)
False
```

Set the current dates so that forced operations should not be evaluated yet

```
>>> sim.data.get_date.mock_returns = np.array([
...     dt.date(2000,7,2),
...     dt.date(1999,7,2),
...     dt.date(2005,7,2)], dtype=dt.date)
>>> sim._forced_operations = {(0,0,u'2'): [fop]}
>>> sim._evaluate_forced_operations()
Called Handler.get_tind(1, 0)
Called Handler.get_id(
```

```

1,
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0]])
Called Handler.get_date(
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0]])
True

```

Set the current dates so that forced operations should already have been evaluated

```

>>> sim.data.get_date.mock_returns = np.array([
...     dt.date(2015,7,2),
...     dt.date(2014,7,2),
...     dt.date(2025,7,2)], dtype=dt.date)
>>> sim._evaluate_forced_operations()
Called Handler.get_tind(1, 0)
Called Handler.get_id(
1,
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0]])
Called Handler.get_date(
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0]])
True

```

### 58.1.15 def \_get\_num\_of\_steps(self):

Get the maximum number of timesteps for the current timespan. As each simulation unit might have different date stamp, the number of time steps will be the maximum over all units.:

```

>>> sim.data.get_date.mock_returns = np.array([
...     dt.date(2005,7,2),
...     dt.date(2004,7,2),
...     dt.date(2015,7,2)], dtype=dt.date)
>>> sim.timespan.ending.type = 'end_date'
>>> sim._get_num_of_steps() # doctest: +NORMALIZE_WHITESPACE
Called Handler.get_tind(1, 0)
Called Handler.get_date(
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0]])
Called Logger.log_message(
'test-log',
'ERROR',
'All simulation units are pass simulation ending date, simulation of
timespan aborted!')
0

>>> sim.timespan.ending.target = dt.date(2020,1,1)
>>> sim._get_num_of_steps()
Called Handler.get_tind(1, 0)
Called Handler.get_date(
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0]])
Called Logger.log_message(

```

```
'test-log',
'WARNING',
'The number of years (17) is not divisible with the given time step (5). Forcably rounding up'
4

>>> sim.timespan.ending.type = 'steps'
>>> sim.timespan.ending.target = 10
>>> sim._get_num_of_steps()
10
```

### 58.1.16 def \_update(self):

Update simulator state and move to next time step:

```
>>> sim._update(to='current period end')
Called Handler.get_tind(1, 0)
Called Handler.get_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
Called Handler.set_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    array([2010-07-01, 2009-07-01, 2020-07-01], dtype=object))
>>> sim.period
0
```

Note that the ‘next period beginning’ update only adds a single day to the current day. That’s because the ‘current period end’ is assumed to be made before this call. However, as the mock object used here doesn’t really change the date, a single day is here added to the initial date.

```
>>> sim._update(to='next period beginning')
Called Handler.get_tind(1, 0)
Called Handler.get_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
Called Handler.set_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    array([2005-07-03, 2004-07-03, 2015-07-03], dtype=object))
Called Handler.set_value(
    1,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    3,
    array([ 2005.,  2004.,  2015.]))
>>> sim.period
1
```

### 58.1.17 def \_set\_data(self, iterations, start, ids):

Create data handler if first simulation run and reset existing handler otherwise. Fill data matrix from input database and store init variable values to data matrix

iterations – number of iterations, int start – starting iteration round ids – list of simulation unit id strings

```

>>> sim.data.matrix.iterations = 1
>>> sim._ctrl.init_variables = [(1,1),0.0], [(0,5),15.0]]
>>> sim._set_data(1, 0, ['1', '2', '3']) # doctest: +ELLIPSIS
Called Handler.reset()
Called Logger.log_message(
    'test-log',
    'INFO',
    'filling simulation data matrix...')
Called InputDB.fill_matrix(<Mock ... Handler>, 1, ['1', '2', '3'])
Called Handler.get_tind(0, 0)
Called Handler.set_value(
    0,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    [5],
    [15.0])
Called Handler.get_tind(1, 0)
Called Handler.set_value(
    1,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    [1],
    [0.0])
Called Logger.log_message('test-log', 'INFO', 'matrix filling complete')

```

### 58.1.18 def \_divide\_data(self, iterations, idlist, indseq):

Divide the simulation target data into reasonable blocks:

```

>>> sim._divide_data(2, [u'1', u'2', u'3', u'4'], None)
Called InputDB.get_ids('data', 1)
Called Logger.log_message(
    'test-log',
    'INFO',
    '4 simulation units divided into 1 sets')
[[[u'1', u'2', u'3', u'4'], 2, 0]]
>>> sim._divide_data(1, None, (2,4))
Called InputDB.get_ids('data', 1)
Called Logger.log_message(
    'test-log',
    'INFO',
    '3 simulation units divided into 1 sets')
[[[u'3', u'4', u'5'], 1, 0]]
>>> sim._divide_data(3, None, None)
Called InputDB.get_ids('data', 1)
Called Logger.log_message(
    'test-log',
    'INFO',
    '6 simulation units divided into 2 sets')
[[[u'1', u'2', u'3'], 3, 0], [[u'4', u'5', u'6'], 3, 0]]
>>> sim._divide_data(5, None, None)
Called InputDB.get_ids('data', 1)
Called Logger.log_message(
    'test-log',
    'INFO',
    '6 simulation units divided into 3 sets')
[[[u'1', u'2'], 5, 0], [[u'3', u'4'], 5, 0], [[u'5', u'6'], 5, 0]]
>>> sim._divide_data(6, None, None)

```

```
Called InputDB.get_ids('data', 1)
Called Logger.log_message(
    'test-log',
    'INFO',
    '6 simulation units divided into 6 sets')
[[[u'1'], 6, 0), ([u'2'], 6, 0), ([u'3'], 6, 0), ([u'4'], 6, 0), ([u'5'], 6, 0), ([u'6'], 6, 0)]
>>> sim._divide_data(10, None, None)
Called InputDB.get_ids('data', 1)
Called Logger.log_message(
    'test-log',
    'INFO',
    '6 simulation units divided into 6 sets')
[[[u'1'], 10, 0), ([u'2'], 10, 0), ([u'3'], 10, 0), ([u'4'], 10, 0), ([u'5'], 10, 0), ([u'6'], 10,
```

Info message should be generated if invalid simulation units are given in idlist:

```
>>> sim._input_db.get_ids.returns = [u'1',u'2',u'3',u'4',u'5']
>>> sim._divide_data(2, [u'2',u'3',u'49',u'139'], None)
Called InputDB.get_ids('data', 1)
Called Logger.log_message(
    'test-log',
    'INFO',
    u'following simulation units were not found in input database: 49 139')
Called Logger.log_message(
    'test-log',
    'INFO',
    '2 simulation units divided into 1 sets')
[(set([u'3', u'2']), 2, 0)]
```

### 58.1.19 def \_set\_dates(self):

Set initial dates for all main level objects:

```
>>> sim._ctrl.use_data_date = True
>>> sim._set_dates()
Called Handler.get_tind(1, 0)
Called Handler.get_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
Called Handler.set_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    array([2005-07-02, 2004-07-02, 2015-07-02], dtype=object))
Called Handler.set_value(
    1,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    3,
    array([ 2005.,  2004.,  2015.]))

>>> sim.data.get_date.mock_returns = np.array([
...     dt.date(1999,1,1),
...     0,
...     dt.date(2005,1,1)], dtype=dt.date)
>>> sim._set_dates()
Called Handler.get_tind(1, 0)
Called Handler.get_date(
```

```

        array([[0, 0, 0, 0, 0],
               [0, 0, 1, 0, 0],
               [0, 0, 2, 0, 0]]))
Called Handler.set_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    array([1999-01-01, 2000-01-01, 2005-01-01], dtype=object))
Called Handler.set_value(
    1,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    3,
    array([ 1999.,  2000.,  2005.]))

>>> sim._ctrl.use_data_date = False
>>> sim._ctrl.use_today = False
>>> sim._set_dates()
Called Handler.get_tind(1, 0)
Called Handler.set_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    array([2000-01-01, 2000-01-01, 2000-01-01], dtype=object))
Called Handler.set_value(
    1,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    3,
    array([ 2000.,  2000.,  2000.]))

```

### 58.1.20 def add\_error(self, errorstr, level, obj, tind=None):

Log error message and update error counters.

**Add a general error message ::**

```

>>> sim.add_error("This is a general error", None, None)
Called Logger.log_message('test-log', 'ERROR', 'This is a general error')

```

### 58.1.21 def add\_warning(self, warningstr, level, obj, tind=None):

Log warning message.

**Add a general warning message ::**

```

>>> sim.add_warning("This is a general warning", None, None)
Called Logger.log_message('test-log', 'WARNING', 'This is a general warning')

```

### 58.1.22 def \_add2log(self, msg, level, obj, tind, mtype):

Add messages to log. Messages can be general or object-specific, in which case object indice(s) must be given as an input parameter. Target index is an optional parameter.

```
>>> tind = np.array([[1, 2, 0, 0, 0],
...                  [0, 0, 1, 0, 0],
...                  [0, 0, 2, 0, 0]], dtype=int)
>>> sim.task = Mock('Task')
>>> sim.task.task_path = 'path/of/this/task'
```

#### Add a general message

```
>>> sim._add2log("This is a general message", None, None, 'TEST')
Called Logger.log_message(
    'test-log',
    'TEST',
    'SIMULATION STEP 1, TASK: path/of/this/task, This is a general message')
```

#### Add a message for a single object at simulation main level

```
>>> sim.level = sim.main_level = 1
>>> sim.data.get_id.mock_returns = [u'3']
>>> sim.data.get_date.mock_returns = [dt.date(2003,3,3)]
>>> etind = np.array([[0, 0, 2, 0, 0]], dtype=int)
>>> sim._add2log("This is a message for a single object", 1, etind, 'TEST')
Called Handler.get_id(1, array([[0, 0, 2, 0, 0]]))
Called Handler.get_date(array([[0, 0, 2, 0, 0]]))
Called Lexicon.get_level_name(1)
Called Logger.log_message(
    'test-log',
    'TEST',
    u"TASK: path/of/this/task, DATE: 2003-03-03, SIMULATION UNIT: '3', ITERATION: 0, BRANCH: 0, ORIGIN: 0")
```

#### Add a message for multiple objects at simulation main level

```
>>> sim.level = sim.main_level = 1
>>> sim.data.get_id.mock_returns = [u'3', u'7']
>>> sim.data.get_date.mock_returns = [dt.date(2003,3,3), dt.date(2007,7,7)]
>>> etind = np.array([[1, 2, 0, 0, 0],
...                  [0, 0, 2, 0, 0]], dtype=int)
>>> sim._add2log("This is a message for a single object", 1, etind, 'TEST')
Called Handler.get_id(1, array([[1, 2, 0, 0, 0],
...                             [0, 0, 2, 0, 0]]))
Called Handler.get_date(array([[1, 2, 0, 0, 0],
...                             [0, 0, 2, 0, 0]]))
Called Lexicon.get_level_name(1)
Called Logger.log_message(
    'test-log',
    'TEST',
    u"TASK: path/of/this/task, DATE: 2007-07-07, SIMULATION UNIT: '7', ITERATION: 0, BRANCH: 0, ORIGIN: 0")
Called Logger.log_message(
    'test-log',
    'TEST',
    u"TASK: path/of/this/task, DATE: 2003-03-03, SIMULATION UNIT: '3', ITERATION: 1, BRANCH: 2, ORIGIN: 0")
```

#### Add a message for a single object on simulation main level's child level

```
>>> sim.level = 3
>>> sim.data.get_id = Mock('get_id')
>>> sim.data.get_id.mock_returns_iter = iter([[u'4-2-11'], [u'4']])
>>> sim.data.get_parent.mock_returns = \
...     np.array([[1,3,0,0,0]], dtype=int), None
>>> sim.lexicon.get_level_name.mock_returns = 'tree'
>>> sim.data.get_date.mock_returns = [dt.date(2004,4,4)]
>>> etind = np.array([[1, 2, 0, 0, 0]], dtype=int)
```



```
>>> sim._add2log("This is a message for a single object", 3, etind, 'TEST')
Called Handler.get_parent(3, array([[1, 2, 0, 0, 0]]), 1)
Called get_id(3, array([[1, 2, 0, 0, 0]]))
Called Handler.get_date(array([[1, 3, 0, 0, 0]]))
Called get_id(1, array([[1, 3, 0, 0, 0]]))
Called Lexicon.get_level_name(3)
Called Logger.log_message(
    'test-log',
    'TEST',
    u"TASK: path/of/this/task, DATE: 2004-04-04, SIMULATION UNIT: '4', ITERATION: 1, BRANCH: 2, OI")
```

### 58.1.23 def \_add\_opres\_to\_db(self):

Add the queued operation results to operation database.:

```
>>> import numpy
>>> sim._opres_queue = [(1, numpy.array([[0, 0, 0, 0, 0]]),
...     (0, 0, 'comp_unit', 's1'), [{'cash_flow': -100.0}],
...     'planting', 'silviculture')])
>>> sim._add_opres_to_db(data_db) # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
Called Handler.get_date(array([[0, 0, 0, 0, 0]]))
Called DataDB.add_opres(
    'comp_unit',
    0,
    [{'op_group': 'silviculture',
      'obj_id': 's1',
      'op_name': 'planting',
      'op_id': 1,
      'values': [{'cash_flow': -100.0}],
      'branch': 0,
      'date': datetime.date(2004, 4, 4)}])
```



# AGGREGATIONCALLER.PY

## 59.1 class AggregationModelCaller(Caller):

Class for constructing and executing aggregation model calls

Attributes:

- arg: AggregationArg object

### 59.1.1 def \_\_init\_\_(self, deterministic=False):

Initialize model caller:

```
>>> from simo.simulation.caller.aggregationcaller import AggregationModelCaller
>>> import numpy
>>> from minimock import Mock
>>> ac = AggregationModelCaller()
>>> ac.arg
```

### 59.1.2 def execute(self, model, params, sim, depth):

Execute aggregation model:

```
>>> def mfunc(arg):
...     arg.results = numpy.array([1.0, None], dtype=float)
>>> sim = Mock('Sim')
>>> class Data(object):
...     def __init__(self):
...         self.active_data_level = 1
...         self.tind = [numpy.array([[0,0,0,0,3],
...                                     [0,0,1,3,5]], dtype=int),
...                       numpy.array([[0,0,0,0,1],
...                                     [0,0,1,1,2],
...                                     [0,0,2,2,3],
...                                     [0,0,3,3,4],
...                                     [0,0,4,4,5]], dtype=int),
...                       numpy.array([[0,0,0,0,1],
...                                     [0,0,1,1,2],
...                                     [0,0,2,2,3],
...                                     [0,0,3,3,4]], dtype=int)]
...         self.vals = [numpy.array([[2.1, 2.2, 2.3, 5.1, 5.2]],
...                                     dtype=float),
...                       numpy.array([[3.1, 3.2, 3.3, 6.2, 6.2]],
...                                     dtype=float),
...                       numpy.array([[4.1, 4.2, 4.3, 7.2, 7.2]],
```

```
...         dtype=float)]
...     def get_value(self, tind, attrs, forced=False):
...         if attrs == [0]:
...             return self.vals[0], None
...         elif attrs == [1]:
...             return self.vals[1], None
...         elif attrs == [2]:
...             return self.vals[2], None
...         else:
...             return self.vals[0][0], None
...     def get_tind(self, lev, depth, dlev=None, tlev=None, v4all=None,
...                 valuelevel=None, order=None):
...         if dlev is not None:
...             tlev = dlev
...         else:
...             tlev = lev
...         self.active_data_level = tlev
...         self.active_dataobjects = self.tind[tlev]
...         return self.tind[tlev], set([])
...     def get_active(self):
...         pass
...     def set_active(self, active):
...         pass
>>> data = Data()
>>> data.set_value = Mock('set_value')
>>> linkage = Mock('Linkage')
>>> linkage.links = {(0,0):{(0,0):{1:[0, 1, 2], 2:[0, 1, 2]},
...                          (0,1):{1:[3, 4], 2:[3,4]},
...                          (1,0):{0:[0], 2:[0]},
...                          (1,1):{0:[0], 2:[1]},
...                          (1,2):{0:[0], 2:[2]},
...                          (1,3):{0:[0], 2:[3]},
...                          (1,4):{0:[0], 2:[ ]},
...                          (2,0):{0:[0], 1:[0]},
...                          (2,1):{0:[0], 1:[1]},
...                          (2,2):{0:[0], 1:[2]},
...                          (2,3):{0:[0], 1:[3]},
...                          }},
...                 }
>>> data.linkage = linkage
>>> model = Mock('Model')
>>> model.name = 'sum'
>>> model._v_func = mfunc

>>> op1 = Mock('Operand')
>>> op1.type = 'variable'
>>> op1.variable = (1, 0)
>>> op1.weight = None
>>> op1.default_value = None
>>> op1.default_weight = None

>>> op2 = Mock('Operand')
>>> op2.type = 'variable'
>>> op2.variable = (0, 1)
>>> op2.weight = None
>>> op2.default_value = None
>>> op2.default_weight = None

>>> params = Mock('Parameters')
>>> params.through_level = 1
>>> params.target_tind = (0,3)
>>> params.operands = [op1, op2]
>>> params.condition = None
```

```
>>> params.scalar = True

>>> logger = Mock('Logger')
>>> sim.level = 1
>>> sim.data = data
>>> sim.logger = logger
>>> sim.log_name = 'test-log'
```

Execute model with two operands and no weights (only one value set due to hardwiring the model results to [1.0, None] always:

```
>>> ac.execute(model, params, sim, 0) # doctest: +ELLIPSIS
Called set_value(0, array([[0, 0, 0, 0, 3]]), 3, array([ 1.]), 'sum')
>>> ac._construct_call()
True
>>> ac.arg.target_index
array([[0, 0, 0, 0, 3],
       [0, 0, 1, 3, 5]])
>>> ac.arg.oper_target_index # doctest: +NORMALIZE_WHITESPACE
deque([array([0, 0, 0, 0, 1],
              [0, 0, 1, 1, 2],
              [0, 0, 2, 2, 3],
              [0, 0, 3, 3, 4],
              [0, 0, 4, 4, 5]]),
       array([0, 0, 0, 0, 3],
              [0, 0, 1, 3, 5]))])
>>> ac.arg.oper_data_level
deque([1, 0])
>>> ac.arg.values # doctest: +NORMALIZE_WHITESPACE
deque([array([ 2.1,  2.2,  2.3,  5.1,  5.2]),
       array([ 3.1,  3.2,  3.3,  6.2,  6.2])])
```

Execute a vectorized aggregation model:

```
>>> opl.scalar = False
>>> params.operands = [opl]
>>> params.condition = None
>>> params.through_level = 1
>>> params.target_ind = (0, [3])
>>> ac.execute(model, params, sim, 0)
Called set_value(0, array([[0, 0, 0, 0, 3]]), [3], array([ 1.]), 'sum')
```

Execute a model with one operand and no weights

```
>>> params.operands = [opl]
>>> ac._construct_call()
True
>>> ac.arg.target_index
array([[0, 0, 0, 0, 3],
       [0, 0, 1, 3, 5]])
>>> ac.arg.oper_target_index # doctest: +NORMALIZE_WHITESPACE
deque([array([0, 0, 0, 0, 1],
              [0, 0, 1, 1, 2],
              [0, 0, 2, 2, 3],
              [0, 0, 3, 3, 4],
              [0, 0, 4, 4, 5]]),
       array([0, 0, 0, 0, 3],
              [0, 0, 1, 3, 5]))])
>>> ac.arg.oper_data_level
deque([1])
>>> ac.arg.values # doctest: +NORMALIZE_WHITESPACE
deque([array([ 2.1,  2.2,  2.3,  5.1,  5.2])])
```

Execute a model with an aggregation condition

```
>>> params.condition = [('variable', (1, 0, True), (None, None)),
...                     ('value', 2.3),
...                     ('eq', (lambda a,b: a==b))]
>>> params.lowest_operand_level = 1
>>> ac._construct_call()
True
>>> ac.arg.target_index
array([[0, 0, 0, 0, 3],
       [0, 0, 1, 3, 5]])
>>> ac.arg.oper_target_index # doctest: +NORMALIZE_WHITESPACE
deque([array([[0, 0, 0, 0, 1],
              [0, 0, 1, 1, 2],
              [0, 0, 2, 2, 3],
              [0, 0, 3, 3, 4],
              [0, 0, 4, 4, 5]])])
>>> ac.arg.oper_data_level
deque([1])
>>> ac.arg.values
deque([array([ NaN,  NaN,  2.3,  NaN,  NaN])])
```

Execute a model with an aggregation condition for a different level from the operand:

```
>>> op3 = Mock('Operand')
>>> op3.type = 'variable'
>>> op3.variable = (2, 2)
>>> op3.weight = None
>>> op3.default_value = None
>>> op3.default_weight = None
>>> params.operands = [op3]

>>> params.condition = [('variable', (1, 0, True), (None, None)),
...                     ('value', 2.3),
...                     ('eq', (lambda a,b: a==b))]
>>> params.lowest_operand_level = 2
```

**NB! The next one is failing although it shouldn't. Make it work...**

```
ac._construct_call() True
ac.arg.target_index
array([[0, 0, 0, 0, 3], [0, 0, 1, 3, 5]])
ac.arg.oper_target_index # doctest: +NORMALIZE_WHITESPACE
deque([array([[0, 0, 0, 0, 1], [0, 0, 1, 1, 2], [0, 0, 2, 2, 3], [0, 0, 3, 3, 4]])])
ac.arg.oper_data_level deque([2])
ac.arg.values deque([array([ NaN, NaN, 4.3, NaN, 7.2])])
```

### 59.1.3 def \_construct\_call(self):

Construct an aggregation model call.

### 59.1.4 def \_add\_operand(self, targetlev, throughlev, tind, oper):

Add a single operand to aggregation arguments.

Parameters:

- targetlev – target level indice

- throughlev – through level indice
- tind – target index, 2d numpy array
- oper – aggregation operand object

#### **59.1.5 def \_call\_model(self):**

Call aggregation model and extract results.





# CALLER.PY

```
>>> from minimock import Mock
```

## 60.1 class Caller(object):

Super class of all model and table caller classes

Attributes:

- `_type`: caller type as string
- `_model`: model / table instance
- `_params`: parameter instance
- `_sim`: simulator instance
- `_depth`: current task depth as integer

### 60.1.1 def \_\_init\_\_(self, callertype):

Initialize caller:

```
>>> from simo.simulation.caller.caller import Caller
>>> caller = Caller('test-type')
>>> caller._sim = Mock('Simulator')
>>> caller._type
'test-type'
```

### 60.1.2 def execute(self, model, params, sim, depth):

Placeholder for subclass implementations.

### 60.1.3 def add\_error(self, errorstr):

Log error message:

```
>>> errorstr = "test error message"
>>> caller.add_error(errorstr, None, None) # doctest: +NORMALIZE_WHITESPACE
Called Simulator.add_error('test-type caller: test error message', None,
                           None)
```

#### **60.1.4 def add\_warning(self, warnstr):**

Log error message:

```
>>> warnstr = "test warning message"
>>> caller.add_warning(warnstr, None, None)
Called Simulator.add_warning(
    'test-type caller: test warning message',
    None,
    None)
```

# GEOTABLECALLER.PY

```
>>> from simo.builder.modelbase.modelbase import ModelbaseDef
>>> tdf = open('../../simulator/xml/schemas/Typedefs_SIMO.xsd')
>>> typedef = tdf.read()
>>> tdf.close()
>>> sf = open('../../simulator/xml/schemas/geo_table.xsd')
>>> schema = sf.read()
>>> sf.close()
>>> xml = u'''<geo_tables xmlns="http://www.simo-project.org/simo"
...     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...     <geo_table name="dem" desc="Digital elevation model">
...         <filename>test.latlonarray</filename>
...         <ulx>21.0</ulx>
...         <uly>61.0</uly>
...         <nrow>3</nrow>
...         <ncol>3</ncol>
...         <xdim>0.5</xdim>
...         <ydim>0.5</ydim>
...         <nvar>1</nvar>
...         <coordinate_variables>
...             <longitude>
...                 <variable>GEO_X</variable>
...                 <level>comp_unit</level>
...             </longitude>
...             <latitude>
...                 <variable>GEO_Y</variable>
...                 <level>comp_unit</level>
...             </latitude>
...         </coordinate_variables>
...         <targets>
...             <target>
...                 <variable>ALT</variable>
...                 <level>comp_unit</level>
...             </target>
...         </targets>
...     </geo_table>
... </geo_tables>'''

>>> class Validator(object):
...     def add_error(msg):
...         print msg

>>> class Lexicon(object):
...     def __init__(self):
...         self.models = {}
...     def get_variable_ind(self, level, variable, active=False):
...         if variable == 'GEO_X':
...             return (1,1)
...         elif variable == 'GEO_Y':
```

```
...         return (1,2)
...     elif variable == 'ALT':
...         return (1,3)
...     elif variable == 'TS':
...         return (1,4)
...     def get_level_ind(self, level):
...         return 1
...     def add_model(self, mtype, mname):
...         if mtype not in self.models:
...             self.models[mtype] = set()
...         self.models[mtype].add(mname)

>>> mb = ModelbaseDef(typedef)
>>> mb.schema = ('geo_table', schema)
>>> mb.xml = ('testxml', xml, Lexicon(),
...         ['./builder/modelbase/test/data'], 'geo_table')
>>> mb.xml['geo_table']['testxml'][:11]
u'<geo_tables'
```

## 61.1 class GeoTableCaller(Caller):

Construct geo table model calls and process results.

### 61.1.1 def \_\_init\_\_(self):

Construct geo table from the parsed geo table xml:

```
>>> from simo.simulation.caller.geotablecaller import GeoTableCaller
>>> gt = mb.obj['geo_table']['testxml']['dem']
>>> gt.load_function(Validator())
>>> gtc = GeoTableCaller()
```

### 61.1.2 def \_\_find\_geo\_indices(self, tind, coord, ulaxis, gridstep, maxsteps, axis):

Find indice from the geo table by coordinate variable values based on the upper left corner and spacing of the grid.

### 61.1.3 def \_\_which\_values\_to\_extract(self, gtable, params):

Which values to return from the geo table

### 61.1.4 def execute(self, gtable, params, sim, depth):

Execute a geo\_table model for the data objects (objs) using the settings from the modelchain (param) for the simulator instance (sim):

```
>>> import numpy
>>> from lxml import etree
>>> from simo.builder.modelbase.geotable import GeoTableParam

>>> etind = numpy.array([[0,0,0,0,1], [0,0,1,1,2], [0,0,2,2,3], [0,0,3,3,4],
...                     [0,0,4,4,5]], dtype=int)
>>> class Data(object):
```

```

...     var_name = {1:{1:'GEO_X', 2:'GEO_Y', 3:'ALT', 4:'TS'}}
...     level_name = {1:'comp_unit', 2:'stratum'}
...     obj_id = numpy.array(['o1'], ['o2'], ['o3'], ['o4'], ['o5']))
...     def get_value(self, tind, attr):
...         size = len(tind[:,0])
...         if attr == 1:
...             if size == 3:
...                 return numpy.array((21.0, 21.0, 22.0)), None
...             elif size == 4:
...                 return numpy.array((21.0, 21.0, 23.0, 21.0)), None
...             elif size == 5:
...                 val = numpy.array((21.0, None, None, None, None),
...                                     dtype=float)
...                 err = ([('missing value!', [1,2,3,4])], etind,
...                         set([1,2,3,4]))
...                 return val, err
...         elif attr == 2:
...             if size == 3:
...                 return numpy.array((60.0, 60.5, 61.0)), None
...             elif size == 4:
...                 return numpy.array((60.0, 64.0, 61.0, 60.5)), None
...             elif size == 5:
...                 val = numpy.array((60.0, None, None, None, None),
...                                     dtype=float)
...                 err = ([('missing value!', [1,2,3,4])], etind,
...                         set([1,2,3,4]))
...                 return val, err
...     def get_tind(self, level, depth):
...         tind = numpy.zeros((depth, 5), dtype=int)
...         tind[:,2] = range(depth)
...         return tind, set([])

>>> from minimock import Mock
>>> sim = Mock('Simulator')
>>> sim.level = 1
>>> sim.value_fixing = False
>>> sim.data = Data()
>>> sim.data.set_value = Mock('Simulator.data.set_value')

>>> pelem = etree.XML('''<geo_table><only_vars><var><name>ALT</name>
...                     <level>comp_unit</level></var></only_vars>
...                     </geo_table>''')
>>> execfile('simulation/caller/test/mocktask.py')
>>> param = GeoTableParam('', pelem, task, gt)

```

Calling with valid data:

```

>>> gtc.execute(gt, param, sim, 3)
Called Simulator.data.set_value(
    1,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]),
    3,
    array([ 100., 103., 108.], dtype=float32),
    'dem',
    False)

```

Calling with incomplete coordinate data:

```

>>> gtc.execute(gt, param, sim, 5)
Called Simulator.add_error(

```

```
'geo table caller: missing value!',
None,
array([[0, 0, 1, 1, 2],
       [0, 0, 2, 2, 3],
       [0, 0, 3, 3, 4],
       [0, 0, 4, 4, 5]]))
Called Simulator.add_error(
    'geo table caller: outside bound X coordinates',
    None,
    array([[0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0],
           [0, 0, 3, 0, 0],
           [0, 0, 4, 0, 0]]))
Called Simulator.add_error(
    'geo table caller: missing value!',
    None,
    array([[0, 0, 1, 1, 2],
           [0, 0, 2, 2, 3],
           [0, 0, 3, 3, 4],
           [0, 0, 4, 4, 5]]))
Called Simulator.add_error(
    'geo table caller: outside bound Y coordinates',
    None,
    array([[0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0],
           [0, 0, 3, 0, 0],
           [0, 0, 4, 0, 0]]))
Called Simulator.data.set_value(
    1,
    array([[0, 0, 0, 0, 0]]),
    3,
    array([ 100.], dtype=float32),
    'dem',
    False)
```

Calling “with out of bounds” values:

```
>>> gtc.execute(gt, param, sim, 4)
Called Simulator.add_error(
    'geo table caller: outside bound X coordinates',
    None,
    array([[0, 0, 2, 0, 0]]))
Called Simulator.add_error(
    'geo table caller: outside bound Y coordinates',
    None,
    array([[0, 0, 1, 0, 0]]))
Called Simulator.data.set_value(
    1,
    array([[0, 0, 0, 0, 0],
           [0, 0, 3, 0, 0]]),
    3,
    array([ 100.,  103.], dtype=float32),
    'dem',
    False)
```

Calling with invalid evaluation level:

```
>>> sim.data.values = None
>>> sim.level = 2
>>> gtc.execute(gt, param, sim, 3) # doctest: +NORMALIZE_WHITESPACE
Called Simulator.lexicon.get_level_name(2)
Called Simulator.lexicon.get_level_name(1)
```

```
Called Simulator.lexicon.get_variable_name(1, 3)
Called Simulator.add_error(
    "geo table caller: Level of (None) target variable 'None' is different
    from model chain evaluation level (None)",
    None,
    None)
```





# MANAGEMENTCALLER.PY

## 62.1 class ManagementModelCaller(Caller):

Class for executing management model calls

### 62.1.1 def \_\_init\_\_(self):

Initialize model caller:

```
>>> from simo.simulation.caller.managementcaller import ManagementModelCaller
>>> mc = ManagementModelCaller()
```

### 62.1.2 def execute(self, model, params, sim, depth):

Execute management model



# OPERATIONCALLER.PY

Operation caller - constructs operation model calls, executes the model calls and extracts model results.

## 63.1 class OperationModelCaller(Caller):

Class for executing operation models

Attributes:

- arg: PredictionArg object:

```
>>> execfile('simulation/caller/test/mocks4operationcaller.py')
Called add_model('parameter_table', 'thinning_preference_order')
Called add_model('cash_flow_table', 'timber_prices')
Called add_model('operation', 'thinning')
```

### 63.1.1 def \_\_init\_\_(self, warnings):

Initialize model caller. Warnings attribute is boolean defining whether warnings will be processed:

```
>>> from minimock import Mock
>>> from simo.simulation.caller.operationcaller import OperationModelCaller
>>> oc = OperationModelCaller(True)
>>> oc.arg
>>> oc.add_error = Mock('add_error')
```

### 63.1.2 def execute(self, model, params, sim, depthind):

Construct and execute operation model call:

```
>>> oc.arg = dummyarg
>>> oc.execute(dummymodel, dummyparams,
...           dummysim, 0) # doctest: +NORMALIZE_WHITESPACE
Called Lexicon.get_level_name(1)
Called get_id(
    1,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
Called Simulator.queue_opres(
    array([[0, 0, 0, 0, 0]]),
    (0, 0, 'comp_unit', 'UID0000'),
    [{'cash_flow': 0.0}],
```

```
        'thinning',
        'thinning')
Called Simulator.queue_opres(
    array([[0, 0, 1, 0, 0]]),
    (0, 0, 'comp_unit', 'UID0001'),
    [{'cash_flow': 0.0}],
    'thinning',
    'thinning')
Called Simulator.queue_opres(
    array([[0, 0, 2, 0, 0]]),
    (0, 0, 'comp_unit', 'UID0002'),
    [{'cash_flow': 0.0}],
    'thinning',
    'thinning')
Called Lexicon.get_level_name(1)
Called get_id(
    1,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
Called Simulator.queue_opres(
    array([[0, 0, 0, 0, 0]]),
    (0, 0, 'comp_unit', 'UID0000'),
    [{'Volume': 0.0, 'assortment': 2, 'SP': 1.0},
     {'Volume': 0.0, 'assortment': 2, 'SP': 3.0},
     {'Volume': 0.0, 'assortment': 1, 'SP': 3.0},
     {'Volume': 0.0, 'assortment': 1, 'SP': 2.0},
     {'Volume': 0.0, 'assortment': 2, 'SP': 2.0},
     {'Volume': 0.0, 'assortment': 1, 'SP': 1.0}],
    'thinning',
    'thinning')
Called Simulator.queue_opres(
    array([[0, 0, 1, 0, 0]]),
    (0, 0, 'comp_unit', 'UID0001'),
    [{'Volume': 0.0, 'assortment': 2, 'SP': 4.0},
     {'Volume': 0.0, 'assortment': 1, 'SP': 4.0},
     {'Volume': 0.0, 'assortment': 2, 'SP': 5.0},
     {'Volume': 0.0, 'assortment': 1, 'SP': 5.0}],
    'thinning',
    'thinning')
Called Simulator.queue_opres(
    array([[0, 0, 2, 0, 0]]),
    (0, 0, 'comp_unit', 'UID0002'),
    [{'Volume': 0.0, 'assortment': 2, 'SP': 8.0},
     {'Volume': 0.0, 'assortment': 1, 'SP': 7.0},
     {'Volume': 0.0, 'assortment': 1, 'SP': 8.0},
     {'Volume': 0.0, 'assortment': 1, 'SP': 6.0},
     {'Volume': 0.0, 'assortment': 2, 'SP': 9.0},
     {'Volume': 0.0, 'assortment': 2, 'SP': 6.0},
     {'Volume': 0.0, 'assortment': 2, 'SP': 7.0},
     {'Volume': 0.0, 'assortment': 1, 'SP': 9.0}],
    'thinning',
    'thinning')
```

### 63.1.3 def \_construct\_call(self, model, params, sim):

Construct operation model call

**63.1.4 def \_assign\_cash\_flow\_table(self, param, sim):**

Assign cash flow table to operation arguments

**63.1.5 def \_assign\_cash\_flow\_model(self, param, sim):**

Assign cash flow model to operation arguments

**63.1.6 def \_assign\_result\_weights(self, weight):**

Assign operation result weight variable values to arguments

**63.1.7 def \_construct\_variables(self, model, sim):**

Get operation variable values from data and add to arguments

**63.1.8 def \_construct\_parameters(self, model, sim):**

Get operation parameter values and add to arguments

**63.1.9 def \_construct\_result\_structure(self, model, sim):**

Construct operation result structure, which depends on the number of target objects and result variables

**63.1.10 def \_get\_number\_of\_targets(self, results, sim):**

Get number of target objects for current operation

**63.1.11 def \_define\_result\_variables(self, results, sim):**

Define number of operation result variables and construct result labels

**63.1.12 def \_classify\_results(self, key, cfiers, sim, depth):**

Generate operation result variable names by combining the variable name and the classifier name from the cash\_flow table

**63.1.13 def \_get\_classifiers(self, classifiers, table, sim, depthind):**

Get values for the operation result classifiers of a single result variable from data or cash flow table classifiers. Return number of result classes for each object, classifier values, classifier names and classifier labels:

```
>>> dummyarg.target_index, toremove = dummysim.data.get_tind(1,0,1)
>>> dummyarg.num_of_targets = len(dummyarg.target_index)
>>> oc.arg = dummyarg
>>> ret = oc._get_classifiers(dummyclassifiers, dummysim, 0)
>>> nclasses, values, names, labels, torem = ret
>>> nclasses
array([6, 4, 8])
>>> values # doctest: +NORMALIZE_WHITESPACE
[[[1, 2], True),
```

```
([array([ 1.,  2.,  3.]), array([ 4.,  5.]),
 array([ 6.,  7.,  8.,  9.])), False)]
>>> names
['assortment', 'SP']
>>> labels
[{1: 'log', 2: 'pulp'}, None]
```

#### 63.1.14 `def _set_labels(self, nc, values, names, labels, rlabs, i=0, row=[], obj=None):`

Set result label keys for each target object. Classifier value can be either from data or from cash flow table classifiers. When some or all of the classifiers are from data, the number of classes between different target objects can be different. If classifiers are only cash flow classifiers, the number of classes between target objects is always the same. Result label data structure works also as operation result structure that is used to pass operation results from operation model to operation caller.

- `nc`: number of classifiers
- `values`: classifier values
- `names`: classifier names in a list
- `labels`: classifier value to label mappings in a dictionary
- `rlabs`: list of lists for storing the classifier tuples for each object and result
- `i`: classifier indice
- `row`: label values from the previous classifiers
- `obj`: 'locked' target object indice

One target object and three cash flow classifiers:

```
>>> values = [( [1,2], True), ([1,2], True), ([1,2], True)]
>>> names = ['assortment', 'quality', 'size']
>>> labels = [{1: 'log', 2: 'pulp'},
...           {1: 'high', 2: 'low'},
...           {1: 'big', 2: 'small'}]
>>> rlabs = [{}, {}, {}]
>>> oc._set_labels(3, values, names, labels, rlabs)
>>> for i in rlabs[0]: print i
(1, 2, 1)
(2, 2, 2)
(1, 2, 2)
(2, 1, 1)
(1, 1, 2)
(2, 2, 1)
(2, 1, 2)
(1, 1, 1)
```

Two target objects and two cash flow classifiers:

```
>>> values = [( [1,2], True), ([1,2,3], True)]
>>> names = ['assortment', 'quality']
>>> labels = [{1: 'log', 2: 'pulp'}, {1: 'high', 2: 'medium', 3: 'low'}]
>>> rlabs = [{}, {}, []]
>>> oc.arg.num_of_targets = 2
>>> oc._set_labels(2, values, names, labels, rlabs)
>>> for i in rlabs[0]: print i
(1, 2)
(1, 3)
(2, 1)
```

```
(2, 3)
(2, 2)
(1, 1)
```

Two target objects, two cash flow classifiers, and one data classifier:

```
>>> import numpy
>>> values = [( [1, 2], True),
...           ([1, 2], True),
...           (numpy.array([1, 2, 3]), numpy.array([4, 5])), False)]
>>> names = ['assortment', 'quality', 'species']
>>> labels = [{1: 'log', 2: 'pulp'}, {1: 'high', 2: 'low'}, None]
>>> rlabs = [{}, {},]
>>> oc._set_labels(3, values, names, labels, rlabs)
>>> for i in rlabs[0]: print i
(1, 2, 1)
(2, 2, 2)
(1, 1, 3)
(2, 1, 1)
(1, 2, 3)
(1, 2, 2)
(2, 1, 2)
(2, 2, 3)
(1, 1, 1)
(1, 1, 2)
(2, 1, 3)
(2, 2, 1)
>>> for i in rlabs[1]: print i
(1, 2, 5)
(2, 2, 5)
(1, 1, 5)
(2, 1, 4)
(2, 2, 4)
(2, 1, 5)
(1, 2, 4)
(1, 1, 4)
```

Two target objects, two data classifiers, and one cash flow classifier:

```
>>> values = [( [1,2], True),
...           (numpy.array([1,2,3]), numpy.array([4,5])), False),
...           (numpy.array([7,8]), numpy.array([9,10])), False)]
>>> names = ['assortment', 'species', 'site']
>>> labels = [{1: 'log', 2: 'pulp'}, None, None]
>>> rlabs = [{}, {},]
>>> oc._set_labels(3, values, names, labels, rlabs)
>>> for i in rlabs[0]: print i
(2, 1, 7)
(1, 2, 8)
(1, 3, 8)
(1, 1, 7)
(1, 1, 8)
(1, 2, 7)
(2, 1, 8)
(1, 3, 7)
(2, 3, 7)
(2, 2, 7)
(2, 2, 8)
(2, 3, 8)
>>> for i in rlabs[1]: print i
(2, 4, 9)
(1, 4, 9)
```

```
(2, 5, 9)
(2, 5, 10)
(1, 5, 10)
(1, 4, 10)
(2, 4, 10)
(1, 5, 9)
```

### 63.1.15 def \_assign\_param\_table(self, sim, depth):

Assign parameter table values to arguments

### 63.1.16 def \_constr\_data(self, model, sim):

Construct operation input data structures and add to arguments. Go through each operation input data level and construct one level at a time for all target objects. When constructing data for operation top level (should be the same as current model chain evaluation level), only normal target index is needed. When the data level is different from top level, also a mapping from top level to the current level's parent level is needed. Also the indices of data level objects is needed. If the current level and top level have data levels in between (at least three data levels total), the values should be get from data with a targetindex that has a mapping from top level to the data level.

### 63.1.17 def \_constr\_data\_level(self, tind, values, child, oind):

Construct operation input data structure for all target object on given level. Number of rows is the number of target objects on the level. Number of columns is number of data attributes +3 if level is operation top level, and +4 otherwise.

Data columns are as follows: 1. iteration 2. branch 3. object indice 4. parent indice if level is not operation evaluation level, otherwise first attribute 4. or 5. to n - 1. attribute values n. operation treatment tag

Construct data level structure for a level without parent level:

```
>>> oc.arg.num_of_targets = 3
>>> tind = numpy.array([[0,0,1,0,4],
...                     [0,0,2,4,6],
...                     [0,1,5,6,9]], dtype=int)
>>> values = numpy.array([[1.1,1.2,1.3],
...                       [2.1,2.2,2.3]], dtype=float)
>>> data = oc._constr_data_level(tind, values, False, None)
>>> print data
[[ 0.  0.  1.  1.1 2.1  0. ]
 [ 0.  0.  2.  1.2 2.2  0. ]
 [ 0.  1.  5.  1.3 2.3  0. ]]
```

Construct data level structure for a level with operation top level as the parent level:

```
>>> oind = numpy.array([[0,0,5],
...                     [0,0,6],
...                     [0,0,7],
...                     [0,0,8],
...                     [0,0,9],
...                     [0,1,10],
...                     [0,1,15],
...                     [0,2,16],
...                     [0,3,17]], dtype=int)
>>> values = numpy.array([[1.11,1.12,1.13,1.14,1.25,1.26,1.37,1.38,1.39],
...                       [2.11,2.12,2.13,2.14,2.25,2.26,2.37,2.38,2.39]],
...                       dtype=float)
>>> data = oc._constr_data_level(tind, values, True, oind)
```



```
>>> print data
[[ 0.  0.  5.  1.  1.11 2.11 0. ]
 [ 0.  0.  6.  1.  1.12 2.12 0. ]
 [ 0.  0.  7.  1.  1.13 2.13 0. ]
 [ 0.  0.  8.  1.  1.14 2.14 0. ]
 [ 0.  0.  9.  2.  1.25 2.25 0. ]
 [ 0.  1. 10.  2.  1.26 2.26 0. ]
 [ 0.  1. 15.  5.  1.37 2.37 0. ]
 [ 0.  2. 16.  5.  1.38 2.38 0. ]
 [ 0.  3. 17.  5.  1.39 2.39 0. ]]
```

Construct data level structure when level's parent level is not the operation top level. Pass also pind (mapping to parent objects for the data level) to the method:

```
>>> tind = numpy.array([[0,0,5,0,3],
...                     [0,0,6,3,8],
...                     [0,0,7,8,10],
...                     [0,0,8,10,15],
...                     [0,0,9,15,16],
...                     [0,0,10,16,20],
...                     [0,0,15,20,22],
...                     [0,0,16,22,27],
...                     [0,0,17,27,29]], dtype=int)
>>> values = numpy.array([numpy.arange(29)], dtype=float)
>>> oind = numpy.transpose([numpy.zeros(29, dtype=int),
...                         numpy.zeros(29, dtype=int),
...                         numpy.arange(3,32)])
>>> data = oc._constr_data_level(tind, values, True, oind)
>>> print data
[[ 0.  0.  3.  5.  0.  0.]
 [ 0.  0.  4.  5.  1.  0.]
 [ 0.  0.  5.  5.  2.  0.]
 [ 0.  0.  6.  6.  3.  0.]
 [ 0.  0.  7.  6.  4.  0.]
 [ 0.  0.  8.  6.  5.  0.]
 [ 0.  0.  9.  6.  6.  0.]
 [ 0.  0. 10.  6.  7.  0.]
 [ 0.  0. 11.  7.  8.  0.]
 [ 0.  0. 12.  7.  9.  0.]
 [ 0.  0. 13.  8. 10.  0.]
 [ 0.  0. 14.  8. 11.  0.]
 [ 0.  0. 15.  8. 12.  0.]
 [ 0.  0. 16.  8. 13.  0.]
 [ 0.  0. 17.  8. 14.  0.]
 [ 0.  0. 18.  9. 15.  0.]
 [ 0.  0. 19. 10. 16.  0.]
 [ 0.  0. 20. 10. 17.  0.]
 [ 0.  0. 21. 10. 18.  0.]
 [ 0.  0. 22. 10. 19.  0.]
 [ 0.  0. 23. 15. 20.  0.]
 [ 0.  0. 24. 15. 21.  0.]
 [ 0.  0. 25. 16. 22.  0.]
 [ 0.  0. 26. 16. 23.  0.]
 [ 0.  0. 27. 16. 24.  0.]
 [ 0.  0. 28. 16. 25.  0.]
 [ 0.  0. 29. 16. 26.  0.]
 [ 0.  0. 30. 17. 27.  0.]
 [ 0.  0. 31. 17. 28.  0.]]
```

### 63.1.18 def \_call\_model(self, model, sim):

Call the operation operation model or the cash\_flow function

### 63.1.19 def \_evaluate\_cash\_flow\_operation(self, sim):

Evaluate cash flow operation

### 63.1.20 def \_extract\_results(self, objs, model, sim):

Extract operation model results

### 63.1.21 def \_process\_cash\_flow(self, sim):

Process and store operation cash flow:

```
>>> oc._process_cash_flow(dummysim)
Called Lexicon.get_level_name(1)
Called OperationArg.get_cash_flow()
Called OperationArg.get_weights()
Called get_id(
    1,
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0]]))
Called Simulator.queue_opres(
    array([[0, 0, 0, 0, 0]]),
    (0, 0, 'comp_unit', 'UID0000'),
    [{'cash_flow': 3.0}],
    'dummymodel',
    'dummygroup')
Called Simulator.queue_opres(
    array([[0, 0, 1, 0, 0]]),
    (0, 0, 'comp_unit', 'UID0001'),
    [{'cash_flow': 4.0}],
    'dummymodel',
    'dummygroup')
Called Simulator.queue_opres(
    array([[0, 0, 2, 0, 0]]),
    (0, 0, 'comp_unit', 'UID0002'),
    [{'cash_flow': 3.0}],
    'dummymodel',
    'dummygroup')
```

### 63.1.22 def \_process\_oper\_results(self, sim):

Process operation results and store them to operation result database:

```
>>> oc.arg.num_of_targets = 3
>>> oc.arg.num_of_results = numpy.array([[4,2,6], [4,2,6]], dtype=int)
>>> oc.arg.result_variables = [('volume', ('assortment', 'SP')),
...                           ('income', ('assortment', 'SP'))]
>>> oc._process_oper_results(dummysim) # doctest: +NORMALIZE_WHITESPACE
Called Lexicon.get_level_name(1)
Called OperationArg.get_weights()
Called OperationArg.get_results()
Called get_id(
```

```

1,
array([[0, 0, 0, 0, 0],
      [0, 0, 1, 0, 0],
      [0, 0, 2, 0, 0]])
Called Simulator.queue_opres(
    array([[0, 0, 0, 0, 0]]),
    (0, 0, 'comp_unit', 'UID0000'),
    [{'volume': 10.800000000000001, 'assortment': 1, 'SP': 2},
     {'volume': 9.8999999999999986, 'assortment': 1, 'SP': 1},
     {'volume': 11.699999999999999, 'assortment': 2, 'SP': 1},
     {'volume': 12.600000000000001, 'assortment': 2, 'SP': 2},
     {'assortment': 1, 'SP': 2, 'income': 108.0},
     {'assortment': 1, 'SP': 1, 'income': 99.0},
     {'assortment': 2, 'SP': 1, 'income': 117.0},
     {'assortment': 2, 'SP': 2, 'income': 126.0}],
    'dummymodel',
    'dummygroup')
Called Simulator.queue_opres(
    array([[0, 0, 1, 0, 0]]),
    (0, 0, 'comp_unit', 'UID0001'),
    [{'volume': 10.6, 'assortment': 1, 'SP': 3},
     {'volume': 11.199999999999999, 'assortment': 2, 'SP': 3},
     {'assortment': 1, 'SP': 3, 'income': 106.0},
     {'assortment': 2, 'SP': 3, 'income': 112.0}],
    'dummymodel',
    'dummygroup')
Called Simulator.queue_opres(
    array([[0, 0, 2, 0, 0]]),
    (0, 0, 'comp_unit', 'UID0002'),
    [{'volume': 6.5999999999999996, 'assortment': 1, 'SP': 5},
     {'volume': 7.2000000000000002, 'assortment': 2, 'SP': 5},
     {'volume': 6.9000000000000004, 'assortment': 2, 'SP': 4},
     {'volume': 6.2999999999999998, 'assortment': 1, 'SP': 4},
     {'assortment': 1, 'SP': 5, 'income': 66.0},
     {'assortment': 2, 'SP': 5, 'income': 72.0},
     {'assortment': 2, 'SP': 4, 'income': 69.0},
     {'assortment': 1, 'SP': 4, 'income': 63.0}],
    'dummymodel',
    'dummygroup')

```

### 63.1.23 def \_process\_data\_results(self, model, sim):

Process operation results that affect the simulation data

### 63.1.24 def \_process\_data(self, model, sim):

Process data objects that are affected by the operation model

### 63.1.25 def \_process\_simulation\_effects(self, params, sim):

Process simulation effects caused by the operation model



# OPERATIONMEMORY.PY

## 64.1 class OperationMemory(object):

Class for managing operation history

Attributes:

- **memory:** five dimensional numpy array with following dimensions:
  1. iteration
  2. branch
  3. object
  4. operation
  5. last year of operation, number of operations done

### 64.1.1 def \_\_init\_\_(self):

Initialize operation memory structure:

```
>>> from simo.simulation.caller.operationmemory import OperationMemory
>>> m = OperationMemory()
```

### 64.1.2 def \_size\_check(self, tind, opind):

Check that operation memory is large enough for given objects, resize memory if necessary:

```
>>> import numpy as np
>>> tind = np.array([[0,0,1,0,0],[0,0,2,0,0],[0,1,1,0,0],[0,2,3,0,0]], dtype=int)
>>> m._size_check(tind, 1)
>>> m._mem.shape
(1, 3, 4, 2, 2)
>>> m._size_check(tind, 4)
>>> m._mem.shape
(1, 3, 4, 5, 2)
```

### 64.1.3 def update(self, tind, opind, level, years):

Update operation memory by adding operation defined by operation indice (opind) for the objects defined by target index (tind):

```
>>> m.update(tind, 0, 1, 2005)
>>> m._mem.shape
(1, 3, 4, 5, 2)
>>> m._mem[0,0,:,0,:]
array([[ 0, 0],
       [2005, 1],
       [2005, 1],
       [ 0, 0]])
>>> tind = np.array([[0,0,2,0,0],[0,0,5,0,0],[0,4,7,0,0]], dtype=int)
>>> m.update(tind, 0, 1, 2015)
>>> m._mem.shape
(1, 5, 8, 5, 2)
>>> m._mem[0,0,:,0,:]
array([[ 0, 0],
       [2005, 1],
       [2015, 2],
       [ 0, 0],
       [ 0, 0],
       [2015, 1],
       [ 0, 0],
       [ 0, 0]])
>>> m._level_map[1]
set([0])
```

#### 64.1.4 def reset(self):

Reset operation memory. Sets all operation memory values to zero.

#### 64.1.5 def since(self, tind, opind, year):

Get number of years since operation was done last time relative to given year:

```
>>> tind = np.array([[0,0,1,0,0],[0,0,2,0,0],[0,0,3,0,0],[0,0,5,0,0]], dtype=int)
>>> m.since(tind, 0, 2020)
array([ 15, 5, 2020, 5])
```

#### 64.1.6 def times(self, tind, opind):

Get number of times a given operation has been done:

```
>>> m.times(tind, 0)
array([1, 2, 0, 1])
>>> m.times(tind, 1)
array([0, 0, 0, 0])
```

#### 64.1.7 def add\_branch(self, par\_tind, new\_tind):

Update operation memory after adding a new branch to simulation data matrix

```
>>> par_tind = np.array([[0,0,1,0,0]], dtype=int)
>>> new_tind = np.array([[0,1,1,0,0]], dtype=int)
>>> m.add_branch(par_tind, new_tind)
>>> m._mem[0,(0,1),1,:,:)
array([[2005, 1],
       [ 0, 0],
```

```
        [ 0, 0],
        [ 0, 0],
        [ 0, 0]],
<BLANKLINE>
        [[2005, 1],
        [ 0, 0],
        [ 0, 0],
        [ 0, 0],
        [ 0, 0]])
```

#### 64.1.8 def del\_objects(self, it, br, objs, level):

Delete objects from operation memory (reset given objects)

```
>>> m.del_objects(0, 0, [0,1,2], 1)
>>> m._mem[0,0,(0,1,2),0,:]
array([[0, 0],
       [0, 0],
       [0, 0]])
```





# PARAMTABLECALLER.PY

## 65.1 def get\_indices(tind, table, sim, depthind):

Find parameter table indices for target objects

## 65.2 def \_limit\_range(table, searchrange, values, i):

Limit parameter table search range for each object with the i'th classifier of the table:

```
>>> import numpy
>>> from simo.simulation.caller.paramtablecaller import _limit_range
>>> table = numpy.array([[1, 1, 1],
...                      [1, 1, 2],
...                      [1, 2, 1],
...                      [1, 2, 2],
...                      [1, 2, 3],
...                      [2, 1, 1],
...                      [2, 1, 2],
...                      [2, 2, 1],
...                      [2, 2, 2],
...                      [2, 2, 3],
...                      [2, 2, 4],
...                      [2, 3, 1],
...                      [3, 1, 1],
...                      [3, 2, 1],
...                      [3, 2, 2],
...                      [3, 3, 3]], dtype=float)
```

Find the ranges for each object where the data values are equal to first classifier values:

```
>>> values = numpy.array([[1, 2, 3, 4]], dtype=float)
>>> searchrange = numpy.array([[1, 0, -1],
...                             [1, 0, -1],
...                             [1, 0, -1],
...                             [1, 0, -1]], dtype=int)
>>> _limit_range(table, searchrange, values, 0)
[None, None, None]
>>> searchrange
array([[ 1,  0,  5],
       [ 1,  5, 12],
       [ 1, 12, 16],
       [ 0,  0, -1]])
```

Find the ranges for each object where the data values are equal to second classifier values:

```
>>> values = numpy.array([[2, 2, 2, 2]], dtype=float)
>>> _limit_range(table, searchrange, values, 1)
[None, None, None]
>>> searchrange
array([[ 1,  2,  5],
       [ 1,  7, 11],
       [ 1, 13, 15],
       [ 0,  0, -1]])
```

Find the ranges for each object where the data values are equal to third classifier values:

```
>>> values = numpy.array([[3, 2, 2, 5]], dtype=float)
>>> _limit_range(table, searchrange, values, 2)
[None, None, None]
>>> searchrange
array([[ 1,  4,  5],
       [ 1,  8,  9],
       [ 1, 14, 15],
       [ 0,  0, -1]])
```

If the last classifier is a float, the parameter table value is interpolated using a coefficient calculated from the closest table values to the float classifier value:

```
>>> from simo.simulation.caller.paramtablecaller import _interpolate_value
>>> table = numpy.array([[5.],
...                     [2.],
...                     [1.],
...                     [4.],
...                     [3.]], dtype=float)
>>> searchrange = numpy.array([[1,0,-1],
...                           [1,0,-1],
...                           [1,0,-1],
...                           [1,0,-1]], dtype=int)
>>> values = numpy.array([[0.5, 1, 2.5, 5.5]], dtype=float)
>>> coeff = _interpolate_value(table, searchrange, values, 0)
>>> coeff
[0.0, 0.5]
>>> searchrange
array([[ 0,  0, -1],
       [ 1,  2,  3],
       [ 1,  1,  2],
       [ 0,  0, -1]])
```

### 65.2.1 class ParamTableCaller(Caller):

Parameter table model caller

Attributes:

- ...

### 65.3 def \_\_init\_\_(self, logger, logname):

# PREDICTIONCALLER.PY

```
>>> from simo.simulation.caller.predictioncaller import PredictionModelCaller
>>> execfile('simulation/caller/test/mocks4predictioncaller.py')
```

## 66.1 class PredictionModelCaller(Caller):

Class for executing prediction models

Attributes:

- memory: PredictionModelMemory object
- arg: PredictionArg object

### 66.1.1 def \_\_init\_\_(self):

Initialize model caller. Warnings attribute is boolean defining whether warnings will be processed:

```
>>> pc = PredictionModelCaller(True)
```

### 66.1.2 def execute(self, model, params, sim, depthind):

Construct and execute prediction model call for a model where the result level is equal to model chain evaluation level (results are stored for existing objects):

```
>>> pc.execute(model, params, sim, 0)
Called Data.get_tind(1, 0, None, None, True, None, None)
Called Data.get_active()
Called Data.get_tind(1, 0, 0, None, True, None, None)
Called Data.get_active()
Called Data.get_value(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 1, 0, 0, 0]]),
    [0, 1])
Called Data.set_active(None)
Called Data.get_value(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 1, 0, 0, 0]]),
    [2, 3])
Called Data.set_value(
    1,
    array([[0, 0, 0, 0, 0],
```

```
[0, 0, 1, 0, 0],
[0, 1, 0, 0, 0]]),
[1, 1, 1, 1],
array([[ 0. ,  1. ,  2. ],
       [ 0.1,  1.1,  2.1],
       [ 0.2,  1.2,  2.2],
       [ 0.3,  1.3,  2.3]]),
'dummymodel',
True)
```

Construct and execute prediction model call for a model where the result level is a child level of model chain evaluation level (new objects are constructed and the results are calculated for the new objects):

```
>>> sim.level = 0
>>> model.result_level = 1
>>> del model.vars[1]
>>> pc.execute(model, params, sim, 0)
Called Data.get_tind(0, 0, None, None, True, None, None)
Called Data.get_active()
Called Data.set_active(None)
Called Data.get_value(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 1, 0, 0, 0]]),
    [0, 1])
Called Data.add_objects(0, 0, 1, 4, 0, 0)
Called Data.set_value(
    1,
    None,
    [1, 1, 1, 1],
    array([[ 0. ,  0.1 ,  0.2 ,  0.3 ],
           [ 0.01,  0.11,  0.21,  0.31],
           [ 0.02,  0.12,  0.22,  0.32],
           [ 0.03,  0.13,  0.23,  0.33]]),
    'dummymodel',
    True)
Called Data.add_objects(0, 0, 1, 5, 0, 1)
Called Data.set_value(
    1,
    None,
    [1, 1, 1, 1],
    array([[ 1. ,  1.04,  1.13,  1.22,  1.31],
           [ 1.01,  1.1 ,  1.14,  1.23,  1.32],
           [ 1.02,  1.11,  1.2 ,  1.24,  1.33],
           [ 1.03,  1.12,  1.21,  1.3 ,  1.34]]),
    'dummymodel',
    True)
Called Data.add_objects(0, 1, 1, 6, 0, 0)
Called Data.set_value(
    1,
    None,
    [1, 1, 1, 1],
    array([[ 2. ,  2.04,  2.12,  2.2 ,  2.24,  2.32],
           [ 2.01,  2.05,  2.13,  2.21,  2.25,  2.33],
           [ 2.02,  2.1 ,  2.14,  2.22,  2.3 ,  2.34],
           [ 2.03,  2.11,  2.15,  2.23,  2.31,  2.35]]),
    'dummymodel',
    True)
```

Execute a prediction model so that data handler returns errors due to missing values

```

>>> values = numpy.array([[None, 2.1, 3.1],
...                        [1.2, None, 3.2]], dtype=float)
>>> err = (('value missing', [0,1]), 'error_tind', set([0,1]))
>>> sim.data.get_value.mock_returns = values, err
>>> sim.level = 1
>>> model.result_level = 1
>>> pc.execute(model, params, sim, 0)
Called Data.get_tind(1, 0, None, None, True, None, None)
Called Data.get_active()
Called Data.get_tind(1, 0, 0, None, True, None, None)
Called Data.get_active()
Called Data.get_value(
    array([[0, 0, 0, 0, 0],
          [0, 0, 1, 0, 0],
          [0, 1, 0, 0, 0]]),
    [0, 1])
Called Simulator.add_error(
    "prediction model caller: value missing when calling model 'dummymodel'",
    1,
    array([[0, 0, 0, 0, 0],
          [0, 0, 1, 0, 0]]))
Called Data.set_value(
    1,
    array([[0, 1, 0, 0, 0]],
          [1, 1, 1, 1],
          array([[ 0. ],
                [ 0.1],
                [ 0.2],
                [ 0.3]]),
    'dummymodel',
    True)

```

### 66.1.3 def \_construct\_call(self, model, params, sim, depthind, tind):

Construct prediction model call

### 66.1.4 def \_construct\_variables(self, model, sim, depthind, tind):

Get model input variable values and add the values to arguments

### 66.1.5 def \_construct\_parameters(self, model, params, sim):

Get model input parameter values and add the values to arguments

### 66.1.6 def \_call\_model(self, model, sim):

Call prediction model function

### 66.1.7 def \_process\_results(self, sim):

Process and store prediction model results

#### **66.1.8 def \_process\_eval\_level\_results(self, level, attrs, sim):**

Store results for existing objects

#### **66.1.9 def \_process\_child\_level\_results(self, level, attrs, sim):**

Create new objects and store model results to the new objects

# PREDICTIONMEMORY.PY

```
>>> import numpy
>>> from datetime import date
>>> from minimock import Mock

>>> model = Mock('Model')
>>> model.name = 'dummymodel'
>>> data = Mock('Data')
>>> data.get_tind.mock_returns = numpy.array([[0,0,1,0,0],
...                                           [0,0,2,0,0]], dtype=int)
>>> data.get_date.mock_returns = numpy.array(\
...     [date(2005,1,1) for i in range(4)])
>>> timespan = Mock('Timespan')
>>> timespan.time_step = 5
>>> sim = Mock('Simulator')
>>> sim.chain_index = 5
>>> sim.timespan = timespan
>>> sim.level = 0
>>> sim.data = data
>>> arg = Mock('Arg')
```

## 67.1 class PredictionModelMemory(object):

Class for handling prediction model memory

Attributes:

- `_mem`: three dimensional numpy array with following dimensions (branches, objects, (modelname,chain) indices)
- `_map`: dictionary mapping (modelname,chain) keys to memory array indices
- `_shp`: memory array shape
- `_blocked`: currently blocked objects

### 67.1.1 def \_\_init\_\_(self):

Initialize model memory

```
>>> from simo.simulation.caller.predictionmemory import PredictionModelMemory
>>> m = PredictionModelMemory()
```

### 67.1.2 def \_key\_check(self, key, level):

Check the given is mapped to memory indice, create mapping if not and return memory indice:

```
>>> m._key_check(('mod1',0), 1)
0
>>> m._key_check(('mod1',0), 1)
0
>>> m._key_check(('mod1',5), 2)
1
>>> m._key_check(('mod2',0), 1)
2
>>> m._key_check(('mod2',5), 2)
3
>>> m._key_check(('mod3',10), 3)
4
>>> keys = m._map.keys()
>>> keys.sort()
>>> print keys
[('mod1', 0), ('mod1', 5), ('mod2', 0), ('mod2', 5), ('mod3', 10)]
>>> vals = m._map.values()
>>> vals.sort()
>>> print vals
[0, 1, 2, 3, 4]
>>> m._level_map[1]
set([0, 2])
>>> m._level_map[2]
set([1, 3])
>>> m._level_map[3]
set([4])
```

### 67.1.3 def \_size\_check(self, tind, ind):

Check that model memory structure is large enough in all dimensions, resize memory if not:

```
>>> m._shp
(1, 1, 0, 0)
>>> tind = numpy.array([[0,2,2,0,0]], dtype=int)
>>> m._size_check(tind, 2)
>>> m._shp
(1, 3, 3, 3)
>>> tind = numpy.array([[0,4,2,0,0]], dtype=int)
>>> m._size_check(tind, 2)
>>> m._shp
(1, 5, 3, 3)
>>> tind = numpy.array([[0,4,5,0,0]], dtype=int)
>>> m._size_check(tind, 2)
>>> m._shp
(1, 5, 6, 3)
>>> m._size_check(tind, 6)
>>> m._shp
(1, 5, 6, 7)
```

### 67.1.4 def use(self, model, sim, depthind, tind):

Check if model memory should be used:



```
>>> m = PredictionModelMemory()
>>> m.use(model, sim, 0, None)
(False, None)
```

### 67.1.5 def update(self, arg, model, sim):

Update prediction model memory. Divide model results for the defined time period and set the ‘last usage’ for the model result values for given objects.

```
>>> arg.result_vars = [Mock('ResultVariable'),
...                   Mock('ResultVariable'),
...                   Mock('ResultVariable'),
...                   Mock('ResultVariable')]
>>> arg.result_vars[0].cumulation = 'once'
>>> arg.result_vars[1].cumulation = 'total'
>>> arg.result_vars[2].cumulation = 'annual'
>>> arg.result_vars[3].cumulation = 'annual%'
>>> arg.result_vars[0].time_span = 10
>>> arg.result_vars[1].time_span = 10
>>> arg.result_vars[2].time_span = 10
>>> arg.result_vars[3].time_span = 10
>>> arg.result_vars[0].time_unit = 'year'
>>> arg.result_vars[1].time_unit = 'year'
>>> arg.result_vars[2].time_unit = 'year'
>>> arg.result_vars[3].time_unit = 'year'
>>> arg.targetindex = numpy.array([[0,0,1,0,0,],
...                               [0,0,2,0,0,],
...                               [0,1,0,0,0,],
...                               [0,1,1,0,0,]], dtype=int)
>>> arg.mem = numpy.ones((4, 4), dtype=float) * 5.0
>>> arg.num_of_res_vars = 4
>>> arg.num_of_res_objs = numpy.ones(4, dtype=int)
```

Update model memory for the first time:

```
>>> arg.target_index = numpy.array([[0,0,0,0,0,],
...                               [0,0,3,0,0,],
...                               [0,2,0,0,0,],
...                               [0,2,3,0,0,]], dtype=int)
>>> m.update(arg, model, sim, 0)
Called Data.get_date(
    array([[0, 0, 0, 0, 0],
          [0, 0, 3, 0, 0],
          [0, 2, 0, 0, 0],
          [0, 2, 3, 0, 0]]))
True
>>> arg.mem
array([[ 5., 5., 5., 5.],
       [ 2.5, 2.5, 2.5, 2.5],
       [25., 25., 25., 25.],
       [27.62815625, 27.62815625, 27.62815625, 27.62815625]])
>>> m._map
{'(dummymodel', 5): 0}
>>> m._shp
(1, 3, 4, 1)
>>> m._mem[0,0,:,0]
array([2014-01-01, 0001-01-01, 0001-01-01, 2014-01-01], dtype=object)
```

Update already existing model memory:

```
>>> m.update(arg, model, sim, 0)
Called Data.get_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 3, 0, 0],
           [0, 2, 0, 0, 0],
           [0, 2, 3, 0, 0]]))
True
>>> m._map
{'dummymodel', 5): 0}
>>> m._mem[0,0,:,0]
array([2014-01-01, 0001-01-01, 0001-01-01, 2014-01-01], dtype=object)
```

Update model memory with new chain index:

```
>>> sim.chain_index = 0
>>> m.update(arg, model, sim, 0)
Called Data.get_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 3, 0, 0],
           [0, 2, 0, 0, 0],
           [0, 2, 3, 0, 0]]))
True
>>> m._map
{'dummymodel', 5): 0, ('dummymodel', 0): 1}
>>> m._mem[0,0,:,1]
array([2014-01-01, 0001-01-01, 0001-01-01, 2014-01-01], dtype=object)
```

Test use of existing model memory:

```
>>> sim.chain_index = 5
>>> tind = numpy.array([[0,0,0,0,0],
...                     [0,0,1,0,0],
...                     [0,0,2,0,0],
...                     [0,0,3,0,0]], dtype=int)
>>> blocks = []
>>> use, tind = m.use(model, sim, 0, tind)
Called Data.get_date(
    array([[0, 0, 0, 0, 0],
           [0, 0, 1, 0, 0],
           [0, 0, 2, 0, 0],
           [0, 0, 3, 0, 0]]))
Called Data.block(0, 0, array([[0, 0, 0, 0, 0],
                               [0, 0, 3, 0, 0]]))
Called Data.get_tind(0, 0)
>>> use
False
>>> tind
array([0, 0, 1, 0, 0])
```

### 67.1.6 def release\_blocked\_objects(self, sim, depthind):

Release objects that were blocked from evaluation because the model had active results stored in the memory:

```
>>> m.release_blocked_objects(sim, 2)
Called Data.release(0, 2, array([[0, 0, 0, 0, 0],
                               [0, 0, 3, 0, 0]]))
```

### 67.1.7 def del\_objects(self, it, br, objs, level):

Delete objects from operation memory (reset objects on given level)

```
>>> m.del_objects(0, 0, [0,1,2], 0)
>>> m._mem[0,0,:,:]
array([[0001-01-01, 0001-01-01],
       [0001-01-01, 0001-01-01],
       [0001-01-01, 0001-01-01],
       [2014-01-01, 2014-01-01]], dtype=object)
```

### 67.1.8 def add\_branch(self, par\_tind, new\_tind):

Update prediction memory after adding a new branch to simulation data matrix

```
>>> par_tind = numpy.array([[0,0,3,0,0]], dtype=int)
>>> new_tind = numpy.array([[0,1,3,0,0]], dtype=int)
>>> m.add_branch(par_tind, new_tind)
>>> m._mem[0,(0,1),3,:]
array([[2014-01-01, 2014-01-01],
       [2014-01-01, 2014-01-01]], dtype=object)
```

### 67.1.9 def reset(self):

Reset prediction memory

```
>>> m.reset()
>>> m._mem[0,0,:,:]
array([[0001-01-01, 0001-01-01],
       [0001-01-01, 0001-01-01],
       [0001-01-01, 0001-01-01],
       [0001-01-01, 0001-01-01]], dtype=object)
```

### 67.1.10 def \_total(self, value, preddiv):

Divide value with prediction divider:

```
>>> m._total(5.0, 5.0)
1.0
```

### 67.1.11 def \_totalpercentage(self, value, preddiv, sim):

Calculate total percentage value:

```
>>> m._totalpercentage(5.0, 5.0, sim)
0.9805797673485328
```

### 67.1.12 def \_totalprob(self, value, preddiv):

Calculate total removal probability over a period of time:

```
>>> m._totalprob(5.0, 5.0)
1.3797296614612149
```

### 67.1.13 def \_divide\_results(self, arg, model, sim, prediv):

Divide the result values between timesteps in the timespan. Results can be cumulated over multiple years in alternative ways which is controlled with cumulation type attribute. The cumulation types are: - once: the result is calculated once, do nothing - total: the value is divided with result timespan / timestep - total%: see \_totalpercentage method - annual: the result is multiplied with simulation timestep - annual%: the same as above, but for percentages - totalprob: ...

Divide the results with alternative cumulation types

```
>>> arg = Mock('Arg')
>>> arg.num_of_res_vars = 5
>>> arg.result_vars = [Mock('RV'), Mock('RV'), Mock('RV'), Mock('RV'),
...                    Mock('RV')]
>>> arg.result_vars[0].cumulation = 'total'
>>> arg.result_vars[1].cumulation = 'total%'
>>> arg.result_vars[2].cumulation = 'annual'
>>> arg.result_vars[3].cumulation = 'annual%'
>>> arg.result_vars[4].cumulation = 'totalprob'
>>> arg.num_of_res_objs = numpy.array([1, 1, 1], dtype=int)
>>> arg.mem = numpy.ones((5, 3), dtype=float)
>>> m._divide_results(arg, model, sim, 5.0)
True
>>> arg.mem
array([[ 0.2          ,  0.2          ,  0.2          ],
       [ 0.19920477,  0.19920477,  0.19920477],
       [ 5.          ,  5.          ,  5.          ],
       [ 5.10100501,  5.10100501,  5.10100501],
       [ 1.          ,  1.          ,  1.          ]])
```

# AGGREGATIONARG.PY

## 68.1 class AggregationArg(object):

Aggregation model argument container

Attributes:

- operands: number of operands as integer
- target\_variable: target indice tuple (level, variable)
- target\_index: branch-object index that identifies the target objects
- remove\_targets: objects that were missing and should be removed
- use\_nan\_funcs: True if operand values contain NaNs due to aggregation condition
- values: operand values as list of numpy arrays
- weights: weight values as list of numpy arrays
- oper\_target\_index: operand target indices as a deque of numpy arrays
- oper\_data\_level: operand data level indices in deque
- success: model evaluation success as boolean
- errors: list of error messages
- results: results as numpy array

### 68.1.1 def \_\_init\_\_(self, ind, tind, torem, scalar, deterministic=False):

```
>>> from simo.simulation.model.aggregationarg import AggregationArg
>>> import numpy
>>> ind = (1,1)
>>> tind = numpy.array([[0,0,0,0,0],
...                     [0,0,1,0,0],
...                     [0,0,2,0,0],
...                     [0,0,3,0,0],
...                     [0,0,4,0,0]], dtype=int)
>>> torem = set([1])
>>> arg = AggregationArg(ind, tind, torem, True)
>>> arg.target_variable
(1, 1)
```

### 68.1.2 def add\_operand(self, datalevel, tind, rm\_targets, values, weights, failed):

Add operand values and weights to argument container

Add a single operand with weight:

```
>>> arg.add_operand(1, tind, set([]), [1,1,1,1,1], [2,2,2,2,2], [])
>>> arg.operands
1
```

```
>>> arg.values
deque([[1, 1, 1, 1, 1]])
>>> arg.weights
deque([[2, 2, 2, 2, 2]])
```

Add second operand with weight:

```
>>> arg.add_operand(1, tind, set([3]), [1,1,1,1,1], [2,2,2,2,2], [])
>>> arg.operands
2
>>> arg.values
deque([[1, 1, 1, 1, 1], [1, 1, 1, 1, 1]])
>>> arg.weights
deque([[2, 2, 2, 2, 2], [2, 2, 2, 2, 2]])
>>> arg.oper_target_index
deque([array([[0, 0, 0, 0, 0],
              [0, 0, 1, 0, 0],
              [0, 0, 2, 0, 0],
              [0, 0, 3, 0, 0],
              [0, 0, 4, 0, 0]]), array([[0, 0, 0, 0, 0],
              [0, 0, 1, 0, 0],
              [0, 0, 2, 0, 0],
              [0, 0, 3, 0, 0],
              [0, 0, 4, 0, 0]])])
```

# OPERATIONARG.PY

```
>>> from simo.simulation.model.operationarg import OperationArg
>>> import numpy
>>> from minimock import Mock
```

## 69.1 class OperationArg(object):

Container class for passing the operation model arguments to operation models

Attributes:

- name: model name as string
- type: model type as string
- language: model implementation language as string
- target\_index: two-dimensional numpy array containing target object indices
- drop\_target: boolean vector for indicating whether the object at the same place in target\_index
- num\_of\_targets
- num\_of\_res\_objs
- num\_of\_res\_vars
- \_cash\_flow
- \_errors
- \_data
- \_variables
- \_params
- \_param\_table\_vals
- \_results
- \_weights
- cash\_flow\_handler
- cash\_flow\_model
- cash\_flow\_table
- cash\_flow\_table\_name
- result\_labels
- result\_type
- result\_level
- sim\_effects
- aborted
- failed

### 69.1.1 def \_\_init\_\_(self, name, group, type, lang):

Initialize operation model argument container

```
>>> sim = Mock('simulator')
>>> arg = OperationArg('dummymodel', 'dummygroup', 'operation', 'Python',
...                   sim)
```

```
>>> arg.name
'dummymodel'
>>> arg.type
'operation'
>>> arg.language
'Python'
```

### 69.1.2 def set\_targets(self, tind, torem):

Set the operation model target index:

```
>>> tind = numpy.array([[0,0,4,0,0],
...                     [0,0,5,0,0],
...                     [0,0,6,0,0],
...                     [0,0,12,0,0],
...                     [0,0,15,0,0]], dtype=int)
>>> arg.set_targets(tind, set([2]))
>>> arg.target_index
array([[ 0,  0,  4,  0,  0],
       [ 0,  0,  5,  0,  0],
       [ 0,  0,  6,  0,  0],
       [ 0,  0, 12,  0,  0],
       [ 0,  0, 15,  0,  0]])
>>> arg.num_of_targets
5
```

### 69.1.3 def set\_cash\_flow(self, cashflow, i):

Set cash flow values either for single target or for all targets:

```
>>> arg.set_cash_flow(5.0, 0)
>>> arg.set_cash_flow(3.0, 1)
>>> arg.set_cash_flow(10.1, 2)
>>> arg.set_cash_flow(1.0, 4)
```

### 69.1.4 def get\_cash\_flow(self, i=None):

Get cash flow container for target object i or for all target objects if i is None:

```
>>> arg.get_cash_flow(0)
array([ 5.])
>>> arg.get_cash_flow(2)
array([ 10.1])
>>> arg.get_cash_flow()
array([ 5. ,  3. , 10.1,  0. ,  1. ])
```

### 69.1.5 def get\_errors(self, i):

Get error container for target object i:

```
>>> arg.get_errors(0)
[]
```



### 69.1.6 def set\_vars(self, nvar, vals, loc, torem):

Set operation input variable values. Create container for the variables if not created yet:

```
>>> vals = numpy.ones(5, dtype=float)
>>> arg.set_vars(3, vals, 2, set([]))
>>> arg._variables
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.,  1.]])
```

### 69.1.7 def get\_vars(self, i):

Get input variable values for target object i:

```
>>> arg.get_vars(0)
array([ 0.,  0.,  1.]])
```

### 69.1.8 def set\_params(self, param):

Set operation input parameter values. Create container for the parameters if not created yet:

```
>>> arg.set_params(1.0)
>>> arg.set_params(2.0)
```

### 69.1.9 def get\_params(self):

Get input parameter values:

```
>>> arg.get_params()
deque([1.0, 2.0])
```

### 69.1.10 def set\_cash\_flow\_table(self, table, target=None):

Add a cash flow table to operation arguments:

```
>>> cashflowtable = Mock('CashFlowTable')
>>> cashflowtable.name = 'tablename'
>>> arg.set_cash_flow_table(cashflowtable) # doctest: +ELLIPSIS
Called simulator._data_db.store_timber_prices(
    <Mock ... simulator>,
    None,
    <Mock ... CashFlowTable>)
>>> arg.cash_flow_table_name
'tablename'
>>> arg.common_cash_flow_table # doctest: +ELLIPSIS
<Mock ... CashFlowTable>
>>> arg.set_cash_flow_table(cashflowtable, 2) # doctest: +ELLIPSIS
Called simulator._data_db.store_timber_prices(
    <Mock ... simulator>,
    2,
    <Mock ... CashFlowTable>)
>>> arg.cash_flow_tables # doctest: +ELLIPSIS
{2: <Mock ... CashFlowTable>}
```

### 69.1.11 def set\_weights(self, values, torem):

Set operation result weight value(s):

```
>>> arg.set_weights(numpy.ones(5, dtype=float), set([]))
```

### 69.1.12 def get\_weights(self, i=None):

Get operation weights for target object i or all targets

```
>>> arg.get_weights(1)
1.0
>>> arg.get_weights()
array([ 1.,  1.,  1.,  1.,  1.])
```

### 69.1.13 def set\_result\_structure(self, self, restype, nres, resvars, results, torem):

Create operation result structure with given number of rows and columns and the result labels:

```
>>> nres = numpy.ones((4, 5), dtype=int)
>>> resvars = [('volume', ('assortment', 'SP'))]
>>> results = numpy.zeros((5, 4), dtype=float)
>>> arg.set_result_structure('data', nres, resvars, results, set([]))
>>> arg.num_of_results
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])
```

### 69.1.14 def get\_results(self, i=None):

Get result structure for target object i:

```
>>> arg.get_results(0)
array([ 0.,  0.,  0.,  0.])
>>> arg.get_results()
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

### 69.1.15 def set\_param\_table\_values(self, values, obj, table, ntables):

Set parameter table values for a single object and target attribute:

```
>>> arg.set_param_table_values([1,2,3], 2, 1, 3)
>>> arg._param_table_vals
[[[]], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], []]
```

### 69.1.16 def get\_param\_table\_vals(self, i):

Get parameter table values for object i:

```
>>> arg.get_param_table_vals(0)
[[], [], []]
>>> arg.get_param_table_vals(2)
[[], [[1, 2, 3]], []]
```

### 69.1.17 def set\_data(self, data, index, level, attrs):

Store input data structure from single level to operation arguments:

```
>>> data = numpy.zeros([5,5], dtype=float)
>>> data[:,2] = range(5)
>>> data[:,3] = 1.1
>>> index = numpy.array([[0,1],[1,2],[2,3],[3,4],[4,5]], dtype=int)
>>> arg.set_data(data, index, 1, [3])
>>> for item in arg._data[1]: print item
[[ 0.  0.  0.  1.1  0. ]
 [ 0.  0.  1.  1.1  0. ]
 [ 0.  0.  2.  1.1  0. ]
 [ 0.  0.  3.  1.1  0. ]
 [ 0.  0.  4.  1.1  0. ]]
[[0 1]
 [1 2]
 [2 3]
 [3 4]
 [4 5]]
[3]

>>> data = numpy.zeros([10,6], dtype=float)
>>> data[:,2] = range(10)
>>> data[:,3] = [0,0,1,1,1,2,2,3,4,4]
>>> data[:,4] = 2.2
>>> index = numpy.array([[0,2],[2,5],[5,7],[7,8],[8,10]], dtype=int)
>>> arg.set_data(data, index, 2, [2])
>>> for item in arg._data[2]: print item
[[ 0.  0.  0.  0.  2.2  0. ]
 [ 0.  0.  1.  0.  2.2  0. ]
 [ 0.  0.  2.  1.  2.2  0. ]
 [ 0.  0.  3.  1.  2.2  0. ]
 [ 0.  0.  4.  1.  2.2  0. ]
 [ 0.  0.  5.  2.  2.2  0. ]
 [ 0.  0.  6.  2.  2.2  0. ]
 [ 0.  0.  7.  3.  2.2  0. ]
 [ 0.  0.  8.  4.  2.2  0. ]
 [ 0.  0.  9.  4.  2.2  0. ]]
[[ 0  2]
 [ 2  5]
 [ 5  7]
 [ 7  8]
 [ 8 10]]
[2]
```

### 69.1.18 def get\_data(self, i):

Get data structure for target object i:

```
>>> data = arg.get_data(0)
>>> data[1]
array([[ 0. ,  0. ,  0. ,  1.1,  0. ]])
>>> data[2]
array([[ 0. ,  0. ,  0. ,  0. ,  2.2,  0. ],
       [ 0. ,  0. ,  1. ,  0. ,  2.2,  0. ]])

>>> data = arg.get_data(1)
>>> data[1]
array([[ 0. ,  0. ,  1. ,  1.1,  0. ]])
>>> data[2]
array([[ 0. ,  0. ,  2. ,  1. ,  2.2,  0. ],
       [ 0. ,  0. ,  3. ,  1. ,  2.2,  0. ],
       [ 0. ,  0. ,  4. ,  1. ,  2.2,  0. ]])

>>> data = arg.get_data(2)
>>> data[1]
array([[ 0. ,  0. ,  2. ,  1.1,  0. ]])
>>> data[2]
array([[ 0. ,  0. ,  5. ,  2. ,  2.2,  0. ],
       [ 0. ,  0. ,  6. ,  2. ,  2.2,  0. ]])

>>> data = arg.get_data(3)
>>> data[1]
array([[ 0. ,  0. ,  3. ,  1.1,  0. ]])
>>> data[2]
array([[ 0. ,  0. ,  7. ,  3. ,  2.2,  0. ]])

>>> data = arg.get_data(4)
>>> data[1]
array([[ 0. ,  0. ,  4. ,  1.1,  0. ]])
>>> data[2]
array([[ 0. ,  0. ,  8. ,  4. ,  2.2,  0. ],
       [ 0. ,  0. ,  9. ,  4. ,  2.2,  0. ]])
```

### 69.1.19 def check\_eval(self, evalres):

Check operation model evaluation result

Check the evaluation result for all result targets. 1 = model successfully evaluated, 0 = warning(s), model evaluated, -1 = error(s), model not evaluated, -2 = model aborted

evalres – evaluation result vector

```
>>> arg.check_eval(numpy.array([1, 1, 1, 1, 1]))
>>> arg._errors[2]=['err1', 'err2']
>>> arg.check_eval(numpy.array([1, 0, -1, -2, 1])) #doctest: +ELLIPSIS
Called simulator.add_warning(
    'Group dummygroup Model dummymodel evaluated with warning(s)!',
    <Mock ... simulator.level>,
    array([[0, 0, 5, 0, 0]]))
Called simulator.add_error(
    'Group dummygroup Model dummymodel failed to evaluate due to error(s)!',
    <Mock ... simulator.level>,
    array([[0, 0, 6, 0, 0]]))
Called simulator.add_error(
    'err1',
    <Mock ... simulator.level>,
    array([[0, 0, 6, 0, 0]]))
Called simulator.add_error(
    'err2',
```

```
<Mock ... simulator.level>,
array([[0, 0, 6, 0, 0]])
Called simulator.add_warning(
'Group dummygroup Model dummymodel aborted due to user set limits.',
<Mock ... simulator.level>,
array([[ 0,  0, 12,  0,  0]]))
```



# PREDICTIONARG.PY

## 70.1 class PredictionArg(object):

Container class for passing the prediction model arguments to prediction library

Attributes:

- variables
- parameters
- result\_vars
- error\_check\_mode
- allowed\_risk\_level
- rect\_factor
- model\_name
- model\_language
- warnings
- errors
- error\_objects
- output\_type
- num\_of\_res: integer
- mem: array
- result\_level: integer
- target\_index: numpy array from data Handler's get\_tind
- num\_of\_targets: integer
- num\_of\_res\_vars: integer
- num\_of\_res\_objs: integer
- remove\_targets: set
- warning\_flag: boolean
- error\_flag: boolean

### 70.1.1 def \_\_init\_\_(self, name, lang):

Initialize a prediction model argument container:

```
>>> from simo.simulation.model.predictionarg import PredictionArg
>>> import numpy
>>> arg = PredictionArg('modelname', 'C')
>>> arg.model_name
'modelname'
>>> arg.model_language
'C'
```

### 70.1.2 def set\_targets(self, tind, torem, nresobjs):

Store target object indices in the argument container:

```
>>> tind = numpy.array([[0,0,0,0,0],
...                    [0,0,1,0,0],
...                    [0,0,2,0,0],
...                    [0,0,3,0,0],
...                    [0,0,4,0,0],
...                    [0,0,5,0,0]], dtype=int)
>>> arg.set_targets(tind, set([3]), 1)
>>> arg.num_of_targets
6
>>> arg.target_index
array([[0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 3, 0, 0],
       [0, 0, 4, 0, 0],
       [0, 0, 5, 0, 0]])
>>> arg.remove_targets
set([3])
```

### 70.1.3 def set\_result\_level(self, result\_level, cur\_level):

Set model result level:

```
>>> arg.set_result_level(0, 'self')
>>> arg.result_level
0
```

### 70.1.4 def set\_input\_variables(self, nvar, values, loc, rem\_targets):

Create container for model input variable values if None and store input variable values into the container:

```
>>> values = numpy.ones(tind.shape[0], dtype=float)
>>> values[0] = numpy.NaN
>>> loc = 2
>>> arg.set_input_variables(3, values, loc, set([0]))
>>> arg.variables
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [NaN,  1.,  1.,  1.,  1.,  1.]])
>>> arg.remove_targets
set([0, 3])
```



### 70.1.5 def set\_result\_variables(self, result\_variables):

Store result variable indices:

```

>>> arg.set_result_variables([0,1,2])
>>> arg.result_vars
[0, 1, 2]
>>> arg.num_of_res_vars
3

```

### 70.1.6 def add\_parameter(self, param):

Add input parameter value to model arguments:

```

>>> arg.add_parameter(1.0)
>>> arg.add_parameter(2.0)
>>> arg.parameters
deque([1.0, 2.0])

```

### 70.1.7 def set\_containers(self):

Construct containers for passing results back from the model:

```

>>> arg.set_containers()
>>> arg.num_of_res_objs
array([0, 0, 0, 0, 0, 0])
>>> arg.mem
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])
>>> arg.errors
['', '', '', '', '', '']

```

### 70.1.8 def remove\_missing(self):

Remove objects with missing input variable values:

```

>>> arg.remove_missing()
>>> arg.variables
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.]])
>>> arg.num_of_res_objs
array([0, 0, 0, 0])
>>> arg.mem
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> arg.errors
['', '', '', '']
>>> arg.num_of_targets
4
>>> arg.target_index
array([[0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 4, 0, 0],
       [0, 0, 5, 0, 0]])

```

### 70.1.9 def python2ctypes(self):

Change argument types from python data types to c data types:

```
>>> arg.python2ctypes()
>>> arg.mem # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
[<simo.simulation.model.predictionarg.c_double_Array_3 object at ...>,
 <simo.simulation.model.predictionarg.c_double_Array_3 object at ...>,
 <simo.simulation.model.predictionarg.c_double_Array_3 object at ...>,
 <simo.simulation.model.predictionarg.c_double_Array_3 object at ...>]
>>> arg.mem[0][0] = 0.0
>>> arg.mem[0][1] = 1.0
>>> arg.mem[0][2] = 2.0
>>> arg.mem[1][0] = 0.1
>>> arg.mem[2][0] = 0.2
>>> arg.mem[3][0] = 0.3
```

### 70.1.10 def ctypes2python(self):

Change argument types from ctypes to python and remove objects with evaluation errors from target objects:

```
>>> arg.ctypes2python()
>>> arg.mem # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
array([[ 0. ,  0.1,  0.2,  0.3],
       [ 1. ,  0. ,  0. ,  0. ],
       [ 2. ,  0. ,  0. ,  0. ]])
```

### 70.1.11 def \_\_construct\_ctypes\_array(self,valuearray):

structure. These arrays are used for passing data to the DLL model libraries. Uses ctypes package for the variable types

### 70.1.12 def add\_error(self, i):

Add error to target object i:

```
>>> arg.add_error(2)
>>> arg.errors
array(['', '', '', ''],
      dtype='<S1')

```

### 70.1.13 def drop\_error\_objects(self):

Trim the prediction model result structures to only contain valid results; i.e., no error in model call. Also remove empty error strings.:

```
>>> arg.drop_error_objects()
>>> arg.target_index
array([[0, 0, 1, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 5, 0, 0]])
>>> arg.num_of_res_objs
array([0, 0, 0])
>>> arg.mem
array([[ 0. ,  0.1,  0.3],
```

```
[ 1. , 0. , 0. ],  
[ 2. , 0. , 0. ]])
```

Utility modules:



# RUNNERCONFIG.PY



# LOGGER.PY





# UTILS.PY

## 73.1 def secs2str(seconds):

Convert seconds into hh:mm:ss.ss string

```
>>> from simo.utils.utils import secs2str
>>> secs2str(1.5)
'00:00:1.500'
>>> secs2str(121.9)
'00:02:1.900'
>>> secs2str(5121.9)
'01:25:21.900'
```



## DEV NOTES

### 74.1 Testing

#### 74.1.1 Output comparison

Running output comparison is quite simple, you just run `buildout_utils/output_tester.py` from the simo root folder. On the parts that don't match, you'll get a listing of what was expected and what was gotten instead.

In case the results don't match (that is, you've changed something and they shouldn't match), you can fix it by first running the `output_tester.py` (if you didn't already) and copying `output.test` from `buildout_utils` to `buildout_utils/output_test_files`. Likewise copy `aggregation.txt`, `by_level.txt`, `expression.txt`, `inlined.txt` and `operation_result.txt` from your output folder there.