Friendly Sam Documentation

Release 0.2.0

Rasmus Einarsson

June 15, 2015

Contents

Friendly Sam is a toolbox developed to formulate and solve optimization-based models of energy systems, but it could be used for many other systems too. Friendly Sam is designed to produce readable and understandable model specifications. It is developed with the Python ecosystem of scientific tools in mind and can be used together with numpy, pandas, matplotlib and many of your other favorite tools.

Note: Friendly Sam is work in progress. Please post any questions or issues on the Issue Tracker.

CHAPTER 1

Friendly Sam is friendly in a number of ways:

Flows of resources The **frien** in friendly stands for flows of resources in energy system networks. With Friendly Sam, we model power plants, energy storages, consumers and other components as nodes in a network, interconnected by flows of "resources". Resources is a common name for all the different flows you could model: district heating and cooling, electric power, fuels, etc.

User-friendly Friendly Sam is user-friendly. Instead of a global namespace with variable names like `VHSTOLOADT`, we use object-oriented code with descriptive names like `model["Heat storage A"].accumulation(42)`. Your model becomes easier to write, understand and maintain.

Open source Friendly Sam is open source software, because we think it's friendly and smart to collaborate. Friendly Sam is released under LGPL v3 license. The source code is on GitHub.

CHAPTER 2

Contents:

2.1 How to install Friendly Sam

2.1.1 Get Python 3

Friendly Sam is developed in Python 3 (at the time of this writing, Python 3.4). Download and install it now, if you haven't already.

2.1.2 Use a virtual environment

It is highly recommended that you use a virtual environment. It's not strictly necessary, but if you choose not to, there is a risk that you will have conflicts between different versions of the packages that Friendly Sam and other Python packages depend on. Google for `python virtualenv` if you want to learn more. If not, you can also do it the way I do, using vex.

- **If you are on Windows**

  1. Open a command prompt.
  2. Make sure you have the latest `setuptools` by running `pip install setuptools --upgrade`
  3. Install vex by running `pip install --user vex`
  4. Create a virtual environment named `my_project_name` and enter it by running `vex -m --python C:\Python34\python.exe my_project_name cmd`

  Now, whenever you want to use your virtual environment, open a command prompt and run `vex my_project_name cmd`.

- **If you are on Linux**

  Basically, you follow the instructions for Windows above but exchange `C:\Python34\python.exe` for something more suitable, and then do `vex my_project_name bash` instead. Also see the docs for vex if you have problems.

2.1.3 Install Friendly Sam

Assuming you have entered/activated your Python virtual environment, or wherever you want to install it, open a command prompt/shell and run the command:

```
pip install friendlysam
```

Optional dependencies

If you want to add support for `pandas` related stuff, or for saving and loading models using `dill`, do one of:

```
pip install friendlysam[pandas]
pip install friendlysam[pickling]
pip install friendlysam[pandas,pickling]
```

2.2 For developers

2.2.1 Install in developer mode

If you are developing the source code of Friendly Sam, you probably want to install it in "develop" mode instead. This has two benefits. First, you get some extra dependencies such as `nose` (testing package), `sphinx` (documentation package) and `twine` and `wheel` (used for releasing), etc. Second, you won't have to reinstall the package into your Python site-packages directory every time you change something.

1. Get Python 3. (Note: If you are on Windows it might be convenient to use a ready-made distribution like WinPython and skip step 5 below, but we can't guarantee it will work.)
2. Download the source code
   - **Alternative 1:** Download a zip file: https://github.com/sp-etx/friendlysam/archive/master.zip
   - **Alternative 2:** If you know git, clone into the repository:

```
git clone https://github.com/sp-etx/friendlysam.git
```

3. You probably want to install Friendly Sam in a *virtual environment*. Create one and activate it before you take the next step.
4. Now, to install Friendly Sam in develop mode, do this:

```
pip install -r develop.txt
```

Note: If you are on Windows, `pip`-installation of some packages will fail if you don't have a compiler correctly configured. One such example is NumPy. A simple way around it is to install binaries from Christoph Gohlke's website for the packages that throw errors when you do `pip install -r develop.txt`.

Let's say you are on Windows and download an installer called something like `numpy-MKL-1.9.0.win-amd64-py3.4.exe`. Don't just run the file, because then it will be installed in your "main" Python installation (usually at `C:\Python34`). Instead, do this:

1. Open a command prompt.
2. Go into your virtual environment (e.g. `vex my_project_name cmd`).
3. (option a) Do this if you have an **.exe file**:

```
easy_install numpy-MKL-1.9.0.win-amd64-py3.4.exe
```

3. (option b) Or, if you have a **.whl file** file, e.g. `numpy-1.9.2+mkl-cp34-none-win_amd64.whl`, do this:

```
pip install numpy-1.9.2+mkl-cp34-none-win_amd64.whl
```

2.2.2 Make Sphinx documentation

The documentation for residues is made with Sphinx and hosted with Read the Docs. To parse nice, human-readable docstrings, we use Napoleon.

- If you want to make a very minor change to the documentation, you can actually just edit the source, push to the github repository and magically, the docs will update at readthedocs.org.
- However, if you want to edit the docs a lot, you probably want to make test builds on your own machine. In that case, you need to learn about Sphinx. To build the docs, open a command prompt, go to `friendlysam\docs` and run the command:

```
make html
```

The resulting HTML can be previewed under `friendlysam\docs\_build\index.html`.

2.2.3 Run tests

Please run the tests before pushing to the master branch.

To run all the tests, including doctests in the source code and doctests in this documentation, go to the project root directory and run:

```
nosetests --with-doctest --doctest-options=+ELLIPSIS
```

2.2.4 Releasing Friendly Sam

If Friendly Sam is installed in develop mode, you should already have twine (for secure communication with PyPI) and wheel (for building wheel distribution files).

1. To put things on PyPI, you have to register on PyPI, and you should register on the test PyPI too:

   https://pypi.python.org/pypi

   https://testpypi.python.org/pypi

2. Make sure that your account is activated. You should get an email from PyPI.
3. Make sure you are added as a maintainer of the friendlysam repository at PyPI/testPyPI.
4. Create yourself a file called `.pypirc` and put it in your home directory. If you are on Windows, the file path should be‘‘C:Usersyourusername.pypirc‘‘. Put the following content in it:

```
[distutils]
index-servers =
    pypi
    test

[pypi]
repository:https://pypi.python.org/pypi
username:your_pypi_username

[test]
```

```
repository:https://testpypi.python.org/pypi
username:your_testpypi_username
```

5. (Windows users) For Windows, there is a nice `pypi.bat` you can use.

   To register info about the package on PyPI, first push to the PyPI test site:

   ```
   pypi.bat register test
   ```

   You will be asked for your PyPI test password. Make sure it turned out as you wanted. Then do the real thing:

   ```
   pypi.bat register pypi
   ```

   To build and upload the distribution, do this:

   ```
   pypi.bat upload test
   ```

   Twine will upload to PyPI and ask you for username and password. Check on the test site that everything is OK. You can also run `pip install ...` from the test repo to be sure. Then upload the package to the real repo by running:

   ```
   pypi.bat upload pypi
   ```

5. (Linux/Mac users) You can easily translate `pypi.bat` into a bash script. Please do so and contribute it to the repository!

2.3 What Friendly Sam is for

2.3.1 Why build another tool?

There are a lot of different tools for optimization-based modeling. Why in the world do we need another one?

The short answer is this: Friendly Sam is a **domain specific toolbox**. For the type of models we work with, the model code is shorter, more readable and easier to debug than it would be with many other tools. Furthermore, Friendly Sam **makes data handling and analysis easier**. Because Friendly Sam is implemented in Python, we get access to all our favorite Python tools for scientific computing and visualization, including Pandas, NumPy, SciPy, matplotlib, etc. This is a strong advantage because the majority of our modeling work is preparing input data and analyzing results.

In the coming paragraphs we'll explain more about what Friendly Sam is. And at end we'll also say a few things Friendly Sam is not.

2.3.2 Data handling is easier with Python

Friendly Sam was designed to simplify our work with optimization-based models of energy systems, so-called *dispatch models*. This is a common type of model in research and in applied analysis of energy systems, based on the thought that the operator(s) of an energy system always act so as to minimize the cost of delivering energy to customers, or maybe (in a parallel universe) to minimize the carbon emissions, or some other objective function. A dispatch model is usually formulated as a minimization problem: *"Minimize the operation cost of this system in this time period, subject to the technical and legal constraints of the system."*

There are a zillion different variants of such models, but many of them have in common that there is a lot of data going in and out. Some examples of possible input data are prices for different forms of energy, demand profiles, technical

constraints, etc. The output data could be operation decisions, system costs, greenhouse gas emissions, and many other things. Therefore, a large part of our modeling work is data handling: Reading and wrangling data files, transforming and resampling input and output data, visualizing results, making statistical tests, etc.

Many optimization-based models are implemented using a generic optimization modeling language like GAMS, AMPL, AIMMS or CMPL. These languages can be wonderful to work with when formulating models because they are made specifically for optimization, and they are efficient in transforming your human-readable code into something that can be understood by almost any optimization solver. However, the infrastructure for handling input and output data in GAMS and AMPL is sub-optimal (pun intended). Anyone who implemented a large, complicated model in one of those languages knows it's not an easy ride to keep track of all the data going in and out, especially not if you want to make a lot of similar runs with different parameter sets. I know several people who wrote their own tools for getting inputs and outputs back and forth between GAMS and their favorite data crunching tool (Excel, Python, MATLAB, R, etc).

When we started writing what would later become Friendly Sam, we chose Python because of the great ecosystem of open source tools that come with it. We have paid specific attention to numpy, pandas, and matplotlib when developing Friendly Sam. It's not necessary to use these tools with Friendly Sam, but there is a great chance they will make your life easier. What about optimization then? To formulate and solve the actual optimization problems, we first used the Python API of the Gurobi optimizer. Gurobi's Python API exposes a Variable class with overloaded operators for addition, multiplication, etc, so you can make algebraic expressions for the optimization objective and all the constraints in Python code. The Gurobi backend then translates these expression objects into a well-formed optimization problem, solves the problem and delivers the solution back through the Python API so you never have to leave Python. In Friendly Sam 1.0 we have created an abstraction layer to reduce the dependence on a certain solver backend. We are now using PuLP to interact with the Gurobi and CBC solvers, but you never have to interact directly with the backend, and it is not too hard to switch to another backend if we want to.

2.3.3 Domain specific toolbox

Friendly Sam is a Python library for formulating, running, and analyzing optimization-based models of energy systems.

In fact, it's not only suitable for modeling energy systems, but also for other systems where you want to optimize flow networks of physical or abstract quantities, be it energy carriers, money, solid waste, cargo deliveries, virtual water or something else.

In principle, you are not even restricted to modeling systems with flow networks, because the optimization engine behind Friendly Sam is exposed so you can formulate a large class of optimization problems. But if you want a generic tool for formulating optimization problems you should probably check out other tools instead. In Python it's worth to look at CyLP, cvxpy, PuLP, and Pyomo. If you want a pure optimization language, look at GAMS, AMPL, AIMMS or CMPL.

So although Friendly Sam can be used as a rather generic optimization modeling tool, it is domain specific in the sense that it has vocabulary for energy systems and similar systems. We developed it specifically to help us formulate dispatch models. In our energy system models, there are almost always balance equations for energy or materials, so Friendly Sam contains definitions of things like `FlowNetwork`, `Node` and `Cluster` to simplify the formulation of such constraints. And the `Node` class is a perfect starting point for modeling things like power plants, energy storages, and other things you typically find in an energy system. Friendly Sam also has a simple formulation of a myopic dispatch model of the type we often encounter in the academic literature on energy system modeling. If you use these building blocks, you will have to think less about sign errors in balance equations and instead concentrate on what your model really means.

Friendly Sam code is meant to be readable. For example, in a district heating model we can have instances of `Node` subclasses, one named `LinearCHP`, another named `HeatPump`, etc. This makes perfect sense to us, because the code is naturally structured similar to how we think about the energy system we are modeling. When the underlying optimization problem is solved, we can query the state of the model objects with code like `heat_pump.consumption['power'](time)`.

The code can also be easier to debug. When you have a bewildering error somewhere, it can be helpful to just eyeball the constraints of your optimization problem, to see if you can spot the error. Friendly Sam makes this easier by automatically naming constraints after their "owner", for example the `HeatPump` instance we just mentioned. You can also name variables and add descriptions to constraints. These features help you understand where things come from when you are looking at a long list of constraints.

2.3.4 What Friendly Sam is not

First, we want to clarify that Friendly Sam is not "a model". It is a toolbox we use to build models.

Second, Friendly Sam is not fool proof. It is entirely possible to make models that are stupid or wrong with Friendly Sam. We have tried to design Friendly Sam to produce readable, understandable, debuggable models, and to make idioms and conventions that help to avoid common errors. But having this tool is not an alternative to knowing and understanding the optimization problems you are creating. Friendly Sam is a tool to help us focus on what is important, rather than chasing indexing errors and how to formulate piecewise affine functions using special ordered sets.

Third, Friendly Sam is not primarily optimized for speed. If you want to solve a really big model fast, you are probably better off with something like AMPL or GAMS, or maybe writing your own code in a compiled language. However, if your model is moderately big you might get the job done faster with Friendly Sam because debugging, data handling, analysis and visualization will be so much faster. In our experience, the development phase often consumes more time and money than the computation phase, so development convenience is often more important than execution speed.

2.3.5 OK, let's get started!

You are now ready to read about Variables and expressions.

Now that you know *What Friendly Sam is for*, let's get started!

2.4 Variables and expressions

In Friendly Sam, each variable is an instance of the `Variable` class. Let's create one:

```
>>> from friendlysam import Variable
>>> my_var = Variable('x')
>>> my_var
<friendlysam.opt.Variable at 0x...: x>
>>> print(my_var)
x
```

Variables can be added, multiplied, subtracted, and so on, to form expressions, including equalities and inequalities.

```
>>> expressions = [
...     my_var * 2 + 1,
...     (my_var + 1) * 2,
...     my_var * 2 <= 3
... ]
>>> for expr in expressions:
...     expr
<friendlysam.opt.Add at 0x...>
<friendlysam.opt.Mul at 0x...>
<friendlysam.opt.LessEqual at 0x...>
>>>
>>> for expr in expressions:
...     print(expr)
```

```
x * 2 + 1
(x + 1) * 2
x * 2 <= 3
```

Warning: The operator == is reserved for checking object similarity, just like we are used to in Python. To create the relation "x equals y", use `Eq`:

```
>>> from friendlysam import Eq
>>> my_var == 1
False
>>> print(Eq(my_var, 1))
x == 1
```

There is also a nice `Sum` operation you should use for large sums. Using the built-in `sum()` will create a deeply nested and very inefficient tree of `Add` objects.

```
>>> from friendlysam import Sum
>>> many_terms = [my_var * i for i in range(100)]
>>> Sum(many_terms)
<friendlysam.opt.Sum at 0x...>
>>> sum(many_terms)
<friendlysam.opt.Add at 0x...>
```

2.4.1 Names don't mean anything

In the example above, we named the `Variable` object `'x'`. This is nothing more than a string attached to the object, and it does not say anything about the identity of the variable. In principle you can have several `Variable` objects with the same name, but that's really confusing and should not be necessary.

```
>>> my_var = Variable('y')
>>> my_other_var = Variable('y')
>>> my_var == my_other_var
False
>>> print(my_var + my_other_var)
y + y
```

It is often a good idea to give your variables names you can recognize, because that simplifies debugging when you want to inspect the expressions you have made with the variables. But if you don't want to name variables you don't have to. The variables are then named automatically.

```
>>> Variable()
<friendlysam.opt.Variable at 0x...: x1>
>>> Variable()
<friendlysam.opt.Variable at 0x...: x2>
```

2.4.2 VariableCollection is like an indexed Variable

There is also a convenient class called `VariableCollection`. It is a sort of lazy dictionary, which creates variables when you ask for them:

```
>>> from friendlysam import VariableCollection
>>> z = VariableCollection('z')
>>> z
<friendlysam.opt.VariableCollection at 0x...: z>
```

```
>>> z(1)
<friendlysam.opt.Variable at 0x...: z(1)>
>>> z((1, 'a'))
<friendlysam.opt.Variable at 0x...: z((1, 'a'))>
>>> z(None)
<friendlysam.opt.Variable at 0x...: z(None)>
```

You can think of `VariableCollection` as an indexed variable, but all it really does is to create variables when you call it, and then remember them.

The index must be hashable. For example, tuples are valid indices, but not lists:

```
>>> z((3, 1, 4))
<friendlysam.opt.Variable at 0x...: z((3, 1, 4))>
>>> z([3, 1, 4])
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

Variables can be named in a namespace, like this:

```
>>> from friendlysam import namespace
>>> with namespace('cheese'):
...     cheese1 = Variable('gorgonzola')
...     cheese2 = VariableCollection('ricotta')
...
>>> cheese1
<friendlysam.opt.Variable at 0x...: cheese.gorgonzola>
>>> cheese2
<friendlysam.opt.VariableCollection at 0x...: cheese.ricotta>
```

The namespace doesn't affect the function of a variable in any way. It only prepends a string representation of whatever object to the variable name, so you can also do things like this:

```
>>> with namespace(dict()):
...     Variable('x')
...
<friendlysam.opt.Variable at 0x...: {}.x>
```

2.4.3 Variables can have values

You can assign a value to a variable. The variable will still work in expressions:

```
>>> x = Variable('x')
>>> x.value = 39
>>> expression = x + 3
>>> expression
<friendlysam.opt.Add at 0x...>
>>> print(expression)
x + 3
```

The difference is that you can now evaluate expressions. But note that the expression object is unchanged.

```
>>> float(expression)
42.0
>>> print(expression)
x + 3
```

You can change or delete the value:

```
>>> x.value = 0.5
>>> int(expression)
3
>>> float(expression)
3.5
>>> expression.value
3.5
>>> del x.value
>>> float(expression)
Traceback (most recent call last):
...
friendlysam.opt.NoValueError: cannot get a numeric value: x + 3 evaluates to x + 3
```

And it works for relations, too:

```
>>> x.value = 10
>>> (x <= 12).value
True
```

2.4.4 Expressions are immutable

Expressions are hashed by structure: If they do the same thing, they hash and compare equal. This also means they are considered equal e.g. as `dict` keys.

```
>>> expr1 = x * 2
>>> expr2 = x * 2
>>> expr1 is expr2 # Different objects!
False
>>> expr1 == expr2 # But similar
True
>>> d = dict()
>>> d[expr1] = 'some value'
>>> d[expr2]
'some value'
```

Expressions are immutable, meaning that their state can never be changed. In the example above, `expr1 == expr2` and that will always be true. Two expressions are interchangeable if (and only if) they compare equal. For any purpose, in any situation, `expr1` will always do the same thing as `expr2`.

However, as you saw above, the result of `float(expr1)` may vary depending on whether variables in the expression have values. Let's look a little bit closer:

```
>>> x.value = 3
>>> expression = x + 39
>>> float(expression)
42.0
>>> x.value = 100
>>> another_expression = x + 39
>>> expression == another_expression
True
>>> float(expression)
139.0
>>> float(another_expression)
139.0
```

This is pretty much analogous to a tuple of mutable objects. The tuple itself may never change, but its contents may:

```
>>> a = [1, 2, 3]
>>> my_tuple = (a, 'something')
>>> my_tuple
([1, 2, 3], 'something')
>>> a[:] = ['changed'] # Only changing the contents of the list
>>> another_tuple = (a, 'something')
>>> my_tuple == another_tuple
True
>>> my_tuple
(['changed'], 'something')
>>> another_tuple
(['changed'], 'something')
```

2.4.5 Behind `value` is `evaluate()`

You might want to know what is happening behind the scenes when you ask for `expression.value` or `float(expression)`. In that case, check out the method *`evaluate()`*.

2.5 Optimization problems

2.5.1 Creating a problem

We use Friendly Sam to formulate MILP problems. The optimization library could be extended to allow other types of problems, too, but this is what is supported today.

Now, let's begin with a full example of an optimization problem.

```
>>> import friendlysam as fs
>>>
>>> # Create the problem
>>> x = fs.VariableCollection('x')
>>> prob = fs.Problem()
>>> prob.objective = fs.Maximize(x(1) + x(2))
>>> prob.add(8 * x(1) + 4 * x(2) <= 11)
>>> prob.add(2 * x(1) + 4 * x(2) <= 5)
>>>
>>> # Get a solver and solve the problem
>>> solver = fs.get_solver()
>>> solution = solver.solve(prob)
>>> type(solution)
<class 'dict'>
>>> solution[x(1)]
1.0
>>> solution[x(2)]
0.75
```

The solver does not in any way affect the problem or the variables. It just reads the problem, solves it and handles back a `dict` with your *Variable* objects as keys and their solutions as values.

If you set the `value` of some variables, those will be inserted into the problem before solving it:

```
>>> x(1).value = 0
>>> solution = solver.solve(prob)
>>> solution
{<friendlysam.opt.Variable at 0x...: x(2)>: 1.25}
```

```
>>> x(1) in solution
False
```

`x(1)` is not in the solution, because you already set its value, so it was handled like a number by the solver.

2.5.2 Debugging constraints

Now let's add another constraint:

```
>>> x(1).value = 0
>>> prob.add(1 <= x(1))
>>> solver.solve(prob)
Traceback (most recent call last):
...
friendlysam.opt.ConstraintError: The expression in <Constraint: Ad hoc constraint> evaluates to False
```

In this case it's obvious why the problem could not be solved. But for argument's sake, let's say we didn't know which constraint was causing a problem. The error message was not too helpful, but the *ConstraintError* luckily also contains a reference to the constraint that failed, so we can pick it out like this:

```
>>> try:
...     solver.solve(prob)
... except fs.ConstraintError as e:
...     failed_constraint = e.constraint
...     print(repr(failed_constraint))
...     print(repr(failed_constraint.expr))
...     print(failed_constraint.expr)
...     print(failed_constraint.desc)
...     print(failed_constraint.origin)
...
<friendlysam.opt.Constraint at 0x...>
<friendlysam.opt.LessEqual at 0x...>
1 <= x(1)
Ad hoc constraint
None
```

OK, that's helpful! We got the problematic constraint out. And there are a few things you should note.

1. The type of the failed constraint is *friendlysam.opt.Constraint*. It was automatically created when we added a *friendlysam.opt.LessEqual* constraint to the problem, and its sole purpose is to wrap the inequality `1 <= x(1)` and to add some metadata.
2. The *Constraint* object contains the *LessEqual* object that we added to the problem.
3. The *Constraint* object contains also a description `desc` and a variable called `origin` which is supposed to say something about where the constraint comes from.

Note: There is a quicker way of printing out some info about a constraint: *long_description*:

```
>>> print(failed_constraint.long_description)
<friendlysam.opt.Constraint at 0x...>
Description: Ad hoc constraint
Origin: None
```

If you want to make your model easier to debug, you can use *Constraint* instances with custom description and/or origin, like in this stupid example:

```
>>> from friendlysam import Constraint
>>> def constr(var, parameter):
...     return var / 42 >= parameter
>>> for i in range(5):
...     expr = constr(x(i), i)
...     origin = (constr, x(i), i)
...     prob += Constraint(expr, desc='Some description', origin=origin)
...
```

2.5.3 Different ways to add constraints

Note: In the examples above, we added constraints like this:

```
>>> prob.add(8 * x(1) + 4 * x(2) <= 11)
>>> prob += Constraint(expr, desc='Some description', origin=origin)
```

These two methods are equivalent, so just choose the syntax you like best.

You can also send an iterable (even a generator), and the items in the iterable can also be iterables, e.g:

```
>>> prob += ([constr(x(i), i), constr(x(i+1), i)] for i in range(5))
```

See the documentation for *add()* for all the details.

2.5.4 Special ordered sets

Friendly Sam also supports special ordered sets. You specify them as a sort of constraint: Check out *SOS1* and *SOS2*.

2.6 Model basics: Parts and constraints

2.6.1 Interconnected parts

Friendly Sam is made for optimization-based modeling of interconnected parts, producing and consuming various resources. For example, an urban energy system may have grids for district heating, electric power, fossil gas, etc. Consumers, producers, storages and other parts are connected to each other through these grids. To describe these relations, Friendly Sam models make heavy use of the *Part* class and its subclasses like *Node*, *FlowNetwork*, *Cluster*, *Storage*, etc. We will introduce these in due time, but first a few general things about *Part*.

2.6.2 Parts have indexed constraints

A *Part* typically represents something physical, like a heat consumer or a power grid. You can attach **constraint functions** to parts. Constraint functions are probably easiest to explain with a concrete example:

```
>>> from friendlysam import Part, namespace, VariableCollection
>>> class ChocolateFactory(Part):
...     def __init__(self):
...         with namespace(self):
...             self.output = VariableCollection('output')
...         self.constraints += self.more_than_last
...
...     def more_than_last(self, time):
```

```
...         last = self.step_time(time, -1)
...         return self.output(last) <= self.output(time)
...
```

OK, what happens above is the following: We define `ChocolateFactory` as a subclass of *Part*. Upon setup, in `__init__()`, we add a constraint function called `more_than_last`, which defines the (admittedly bizarre) rule that the factory may never decrease its output from one time step to the next.

In Friendly Sam's vocabulary, the `time` argument in the example above is called an **index**. Our typical use case for indexing is a discrete time model, where each hour, day, year, or whatever time period, is an index of the model, and each constraint "belongs" to a time step just like in the silly example above.

Going back to the example, we can get the constraints out by making a `ChocolateFactory` instance and calling `constraints.make()` with an index:

```
>>> chocolate_factory = ChocolateFactory() # Create an instance
>>> constraints = chocolate_factory.constraints.make(47)
>>> constraints
{<friendlysam.opt.Constraint at 0x...>}
>>> for c in constraints:
...     print(c.expr)
...
ChocolateFactory0001.output(46) <= ChocolateFactory0001.output(47)
```

The result of `constraints.make(47)` is a set with one single constraint in it, saying that output at "time" 47 must be greater than or equal to output at "time" 46.

Note: In the example above, we wrote `last = self.step_time(time, -1)` instead of just `last = t-1`. This is because *Part* has a bunch of nice functions to help out with time indexing. Read the API documentation for *step_time()*. Also check out *times()* and *times_between()*. For example, because we used `step_time(...)`, we can easily change the time representation of our chocolate factory like this:

```
>>> from pandas import Timestamp, Timedelta
>>> chocolate_factory.time_unit = Timedelta('6h')
>>> constraints = chocolate_factory.constraints.make(Timestamp('2015-06-10 18:00'))
>>> for c in constraints:
...     print(c.expr)
...
ChocolateFactory0001.output(2015-06-10 12:00:00) <= ChocolateFactory0001.output(2015-06-10 18:00:00)
```

2.6.3 Advanced indexing

Indexing is really just a way to organize constraints in groups that belong together. When we have a whole bunch of parts, to get all the constraints that belong together, we can do things like this:

```
>>> from itertools import chain
>>> parts = Part(), Part(), Part() # Put something more useful here...
>>> some_index = 'could be anything'
>>> constraints = set.union(*(p.constraints.make(some_index) for p in parts))
```

Indexing is typically used to represent time, but it is really up to you to decide what an index means, and what to use as indices. We call it "index" rather than "time" because it is something more general than a representation of time. In fact, any hashable object can be used as an index, so you can do all sorts of complicated things. Examples of indexing can be found in the docs for *step_time()*. Also check out *times()* and *times_between()*.

Friendly Sam currently has no mechanism for using constraint functions without indices. If you want to make a static model and really don't need indexing, then just use some common index like `None` or `0` for everything. (Or come up

with a better solution and discuss it with us on GitHub.)

In the next section *Flow networks: Nodes and resources* you will also see how indexing is used to represent time in flow networks.

2.7 Flow networks: Nodes and resources

Note: This tutorial does not cover everything. To learn more, follow the links into the API reference for *Node*, *FlowNetwork*, *Cluster* etc.

Friendly Sam makes it easy to formulate optimization problems with flow networks. Let's begin with an example.

2.7.1 Nodes and balance constraints

An example

Custom types of nodes should typically be created by subclassing *Node*, like this:

```
>>> from friendlysam import Node, VariableCollection, namespace
>>> class PowerPlant(Node):
...     def __init__(self):
...         with namespace(self):
...             x = VariableCollection('output')
...         self.production['power'] = x
...
>>> class Consumer(Node):
...     def __init__(self, demand):
...         self.consumption['power'] = lambda time: demand[time]
...
```

We have now defined a `PowerPlant` class inheriting *Node*, and a `Consumer` class, also inheriting *Node*. The power plant has its `production['power']` equal to a *VariableCollection*, and the consumer has `consumption['power']` equal to the value found in the argument `demand`. Let's create instances and test them:

```
>>> power_plant = PowerPlant()
>>> power_plant.production['power'](3)
<friendlysam.opt.Variable at 0x...: PowerPlant0001.output(3)>
```

```
>>> power_demand = [25, 30, 33, 29, 27]
>>> consumer = Consumer(power_demand)
>>> consumer.consumption['power'](3)
29
```

Now connect the two nodes:

```
>>> from friendlysam import FlowNetwork
>>> power_grid = FlowNetwork('power', name='Power grid')
>>> power_grid.connect(power_plant, consumer)
>>> power_grid.children == {power_plant, consumer}
True
```

The `Consumer` instance and the `PowerPlant` instance were added to the power grid, and can now be found as *children* of the *FlowNetwork*.

Note: In this example, we use the key `'power'` in a few different places. Whatever we put as a key in a

production or *consumption* dictionary, or a similar place, is called a **resource**. You are not limited to strings like `'power'` but could use any hashable type: numbers, tuples, most other objects, etc.

Balance constraints

Each *Node* has a pre-defined constraint function for balance constraints, so calling `constraints.make()` on the nodes creates **balance constraints**. The dictionaries *production* and *consumption* are automatically included in these balance constraints. The *connect()* call creates a flow between two nodes, and it adds this flow to the appropriate *outflows* or *inflows* on those two nodes. Each *Node* can then formulate its own balance constraints:

```
>>> for part in [consumer, power_plant, power_grid]:
...     for constraint in part.constraints.make(3):
...         print(constraint.long_description)
...         print(constraint.expr)
...         print()
...
<friendlysam.opt.Constraint at 0x...>
Description: Balance constraint (resource=power)
Origin: CallTo(func=<bound method Consumer.balance_constraints of <Consumer at 0x...: Consumer0001>>,
Power grid.flow(PowerPlant0001-->Consumer0001)(3) == 29

<friendlysam.opt.Constraint at 0x...>
Description: Balance constraint (resource=power)
Origin: CallTo(func=<bound method PowerPlant.balance_constraints of ...>, index=3, owner=<PowerPlant
PowerPlant0001.output(3) == Power grid.flow(PowerPlant0001-->Consumer0001)(3)
```

How balance constraints are made

Here are a few simple rules for how balance constraints are made:

- Each *Node* has the five dictionaries *consumption*, *production*, *accumulation*, *inflows*, and *outflows*.
- Whatever you decide to put as a key in any of these dictionaries is called a **resource**.
- For each resource present in any of the dictionaries, the *Node* produces balance constraints like this:

  (sum of inflows) + production = consumption + accumulation + (sum of outflows)

- The constraints of the node are accessed by calling something like

```
>>> index = 3
>>> constraints = power_plant.constraints.make(index)
```

  The index is passed on to the functions: `production[resource](index)`, `consumption[resource](index)`, etc. In this way, indices always represent time when you are working with nodes and flow networks. You can use any function or object as `production[resource]`, `consumption[resource]`, etc, as long as it is callable.

Note: A *Node* instance will always produce balance constraints for each of its *resources*. Let's say we had not connected the `PowerPlant` instance to the consumer, then its balance constraint would be `PowerPlant0001.output(3) == 0`. (Try it yourself!) In other words, flows of resources must always be balanced in a Friendly Sam model. Noone may produce a resource like `'power'` if it has nowhere to go, and noone can consume it unless there is a source.

Custom names

Note: You can name your *Node* instances if you want something more personal than `PowerPlant0001`. Just set the property *name*, for example in the `__init__` function, like this:

```
>>> class CHPPlant(Node):
...     def __init__(self, name=None):
...         if name:
...             self.name = name
...         ...
>>> chp_plant = CHPPlant(name='Rya KVV')
>>> chp_plant.name == str(chp_plant) == 'Rya KVV'
True
```

2.7.2 FlowNetwork

A *FlowNetwork* essentially does two things: It creates the variable collections representing flows in the network, and it modifies the *inflows* and *outflows* of nodes when you call *connect()*.

Unidirectional by default

Connections are unidirectional, so when you `connect(node1, node2)` things can flow from `node1` to `node2`. Make the opposite connection if you want a bidirectional flow, or use this shorthand:

```
>>> power_grid.connect(power_plant, consumer, bidirectional=True)
```

Flow restrictions

To limit the flow between two nodes, get the flow *VariableCollection* and set its upper bound *ub*:

```
>>> flow = power_grid.get_flow(power_plant, consumer)
>>> flow
<friendlysam.opt.VariableCollection at 0x...: Power grid.flow(PowerPlant0001-->Consumer0001)>
>>> flow.ub = 40
```

2.7.3 Clusters and multi-area models

A cluster is fully connected

Sometimes we are not interested in making a full network model specifying all the flows between different nodes. The *Cluster* class is a handy type of *Node* for that. It is a type of node that can contain other nodes, and it essentially acts like a fully connected network, where all nodes are connected to all others.

When a *Node* is put in a *Cluster*, the child *Node* will no longer make balance constraints, and instead the *Cluster* creates an aggregated balance constraint, summing up the `production`, `consumption` and `accumulation` of its contained *children*.

```
>>> from friendlysam import Cluster
>>> power_plant = PowerPlant()
>>> consumer = Consumer(power_demand)
>>> power_cluster = Cluster(power_plant, consumer, resource='power', name='Power cluster')
>>> for part in power_cluster.descendants_and_self:
```

```
...     for constraint in part.constraints.make(2):
...         print(constraint.long_description)
...         print(constraint.expr)
...
<friendlysam.opt.Constraint at 0x...>
Description: Balance constraint (resource=power)
Origin: CallTo(func=<bound method Cluster.balance_constraints ...>, index=2, owner=<Cluster at 0x...
PowerPlant0002.output(2) == 33
```

Multi-area models

A *Cluster* instance can be used like any other *Node*, for example in a *FlowNetwork*. This is a simple way of making a multi-area model of, say, a district heating system. Let's say the system has a few areas with significant flow restrictions between them. Then create a flow network with interconnected clusters, something like this:

```
area_A = Cluster(*nodes_in_area_A, resource='heat')
area_B = Cluster(*nodes_in_area_B, resource='heat')
area_C = Cluster(*nodes_in_area_C, resource='heat')

heat_grid = FlowNetwork('heat')
heat_grid.connect(area_A, area_B, bidirectional=True, capacity=ab)
heat_grid.connect(area_A, area_C, bidirectional=True, capacity=ac)
heat_grid.connect(area_B, area_C, bidirectional=True, capacity=bc)
```

2.7.4 Time in flow networks

It is natural to think of indices like time periods: All the expressions for flows, production and consumption must add up, for each index (time period). As shown in the examples above, the balance constraints for an index is called by passing the index to *production*, *consumption*, *outflows* and *inflows*.

There is another dictionary which is always used in balance constraints: *accumulation*. It works just like the dictionaries *production* and *consumption*. To learn more, read the API docs for *Storage*, and look at this example:

```
>>> from friendlysam import Storage
>>> from pandas import Timestamp, Timedelta
>>> battery = Storage('power', name='Battery')
>>> battery.time_unit = Timedelta('3h')
>>> t = Timestamp('2015-06-10 18:00')
>>> print(battery.accumulation['power'](t))
Battery.volume(2015-06-10 21:00:00) - Battery.volume(2015-06-10 18:00:00)
```

2.8 Example model

To get a feeling for what's possible with Friendly Sam, have a look at this example model:

https://github.com/sp-etx/example-model

2.9 API reference

2.9.1 Variables

Variable([name, lb, ub, domain])	A variable to build expressions with.
VariableCollection([name])	A lazy collection of *Variable* instances.
Domain	Domain of a variable.
namespace(name)	Prefix variable names.

friendlysam.opt.Variable

class friendlysam.opt.**Variable**(*name=None*, *lb=None*, *ub=None*, *domain=<Domain.real: 0>*)

A variable to build expressions with.

Parameters

- **name** (*str, optional*) – A name of the variable. It has no relation to the identity of the variable. Just a name used in string representations.
- **lb** (*number, optional*) – If supplied, a lower bound on the variable in optimization problems. If not supplied, the variable is unbounded downwards.
- **ub** (*number, optional*) – If supplied, an upper bound on the variable in optimization problems. If not supplied, the variable is unbounded upwards.
- **domain** (any of the *Domain* values) – The domain of the variable, enforced in optimization problems.

Note: The name, lb, ub and domain can also be set as attributes after creation.

```
>>> a = Variable('a')
>>> a.lb = 10
>>> a.Domain = Domain.integer
```

is equivalent to

```
>>> a = Variable('a', lb=10, domain=Domain.integer)
```

Examples

The *namespace()* context manager can be used to conveniently name groups of variables.

```
>>> with namespace('dimensions'):
...     w = Variable('width')
...     h = Variable('height')
...
>>> w.name, h.name
('dimensions.width', 'dimensions.height')
```

Variable.evaluate([replace, evaluators])	Evaluate a variable.
Variable.take_value(solution)	Try setting the value of this variable from a dictionary.

friendlysam.opt.Variable.evaluate

Variable.**evaluate**(*replace=None*, *evaluators=None*)

Evaluate a variable.

See *Operation.evaluate()* for a general explanation of expression evaluation.

A *Variable* is evaluated with the following priority order:

1. If it has a *value*, that is returned.
2. Otherwise, if the variable is a key in the *replace* dictionary, the corresponding value is returned.
3. Otherwise, the variable itself is returned.

Parameters

- **replace** (*dict, optional*) – Replacements.
- **evaluators** (*dict, optional*) – Has no effect. Just included to be compatible with the signature of *Operation.evaluate()*.

Examples

```
>>> x = Variable('x')
>>> x.evaluate() == x
True
>>> x.evaluate({x: 5}) == 5
True
```

```
>>> x.value = -1
>>> x.evaluate() == -1
True
>>> x.evaluate({x: 5}) == -1 # .value goes first!
True
```

```
>>> del x.value
>>> x.value
Traceback (most recent call last):
...
friendlysam.opt.NoValueError
```

friendlysam.opt.Variable.take_value

Variable.**take_value**(*solution*)

Try setting the value of this variable from a dictionary.

Set `self.value = solution[self]` if possible.

Raises **KeyError if ‘‘solution[self]‘‘ is not available.** –

Variable.value	Value property.
Variable.variables	

friendlysam.opt.Variable.value

Variable.**value**
Value property.

> **Warning:** There is nothing stopping you from setting `value` to a value which is inconsistent with the bounds and the domain of the variable.

friendlysam.opt.Variable.variables

Variable.**variables**

friendlysam.opt.VariableCollection

class friendlysam.opt.**VariableCollection**(*name=None*, ***kwargs*)
A lazy collection of *Variable* instances.

Use this class to create a family of variables. *Variable* instances are created as needed, and then kept in the collection.

Parameters

- **name** (*str, optional*) – Name of the variable family.
- ****kwargs** (*optional*) – Passed on as keyword arguments to *Variable* constructor.

Examples

```
>>> x = VariableCollection('x')
>>> x
<friendlysam.opt.VariableCollection at 0x...: x>
>>> x(1)
<friendlysam.opt.Variable at 0x...: x(1)>
```

```
>>> x = VariableCollection('y', lb=0, domain=Domain.integer)
>>> x(1).lb
0
>>> x(1).domain
<Domain.integer: 1>
```

VariableCollection.__call__(index)	Get a variable from the collection.

friendlysam.opt.VariableCollection.__call__

VariableCollection.**__call__**(*index*)
Get a variable from the collection.

A *VariableCollection* is callable. You call the object to get a *Variable* from the collection.

Parameters **index** (*hashable object*) – The index of the requested variable.

Returns

class:*VariableCollection* has not been called earlier with this index, creates a new *Variable*

instance and returns it.

If the index has been used before, the same *Variable* instance will be returned.

Return type If the

Examples

```
>>> x = VariableCollection('x')
>>> x
<friendlysam.opt.VariableCollection at 0x...: x>
>>> x(1)
<friendlysam.opt.Variable at 0x...: x(1)>
```

VariableCollection.domain	Gets the domain of the contained variables.
VariableCollection.lb	Gets the lower bound of the contained variables.
VariableCollection.ub	Gets the upper bound of the contained variables.

friendlysam.opt.VariableCollection.domain

VariableCollection.**domain**

Gets the domain of the contained variables.

> **Warning:** Gets the domain stored on the VariableCollection. The value on individual variables may have been changed individually.

friendlysam.opt.VariableCollection.lb

VariableCollection.**lb**

Gets the lower bound of the contained variables.

> **Warning:** Gets the upper bound stored on the VariableCollection. The value on individual variables may have been changed individually.

friendlysam.opt.VariableCollection.ub

VariableCollection.**ub**

Gets the upper bound of the contained variables.

> **Warning:** Gets the upper bound stored on the VariableCollection. The value on individual variables may have been changed individually.

friendlysam.opt.Domain

class friendlysam.opt.**Domain**

Domain of a variable.

Variable and *VariableCollection* support these domains passed in with the `domain` keyword argument of the constructor.

Examples

```
>>> for d in Domain:
...     print(d)
...
Domain.real
Domain.integer
Domain.binary
```

```
>>> s = get_solver()
>>> prob = Problem()
>>> x = Variable('x', domain=Domain.integer)
>>> prob.objective = Minimize(x)
>>> prob += (x >= 41.5)
>>> solution = s.solve(prob)
>>> solution[x] == 42
True
```

Domain.binary
Domain.integer
Domain.real

friendlysam.opt.Domain.binary

Domain.**binary** = <Domain.binary: 2>

friendlysam.opt.Domain.integer

Domain.**integer** = <Domain.integer: 1>

friendlysam.opt.Domain.real

Domain.**real** = <Domain.real: 0>

friendlysam.opt.namespace

friendlysam.opt.**namespace**(*name*)

Prefix variable names.

Examples

```
>>> with namespace('dimensions'):
...     w = Variable('width')
...     h = VariableCollection('heights')
...
>>> w
<friendlysam.opt.Variable at 0x...: dimensions.width>
>>> h(3)
<friendlysam.opt.Variable at 0x...: dimensions.heights(3)>
```

2.9.2 Expressions

Operation	An operation on some arguments.
Add	Addition operator.
Sub	Subtraction operator.
Mul	Subtraction operator.
Sum	A sum of items.
dot(a, b)	Make expression for the scalar product of two vectors.
Relation	Base class for binary relations.
Less	The relation “less than”.
LessEqual	The relation “less than or equal to”.
Eq	The relation “equals”.

friendlysam.opt.Operation

class friendlysam.opt.**Operation**

An operation on some arguments.

This is a base class. Concrete examples:

Arithmetic operations:

- Addition: *Add*
- Subtraction: *Sub*
- Multiplication: *Mul*
- Summation: *Sum*

Relations:

- Less than (<): *Less*
- Less or equal (<=): *LessEqual*
- Equals: *Eq*

Note: The *Variable* class and the arithmetic operation classes have overloaded operators which create *Operation* instances. For example:

```
>>> x = Variable('x')
>>> isinstance(x * 2, Operation)
True
>>> x + 1
<friendlysam.opt.Add at 0x...>
```

Operation.create(*args)	Classmethod to create a new object.
Operation.evaluate([replace, evaluators])	Evaluate the expression recursively.

friendlysam.opt.Operation.create

classmethod Operation.**create**(**args*)

Classmethod to create a new object.

This method is the default evaluator function used in *evaluate()*. Usually you don’t want to use this function,

but instead the constructor.

Parameters ***args** – The arguments the operation operates on.

Examples

```
>>> x = Variable('x')
>>> args = (2, x)
>>> Add.create(*args) == 2 + x
True
>>> LessEqual.create(*args) == (2 <= x)
True
```

friendlysam.opt.Operation.evaluate

Operation.**evaluate**(*replace=None*, *evaluators=None*)

Evaluate the expression recursively.

Evaluating an expression:

1. Get an evaluating function. If the class of the present expression is in the `evaluators` dict, use that. Otherwise, take the *`create()`* classmethod of the present expression class.

2. Evaluate all the arguments. For each argument `arg`, first try to replace it by looking for `replace[arg]`. If it's not there, try to evaluate it by calling `arg.evaluate()` with the same arguments supplied to this call. If `arg.evaluate()` is not present, leave the argument unchanged.

3.Run the evaluating function `func(*evaluated_args)` and return the result.

Parameters

- **replace** (*dict, optional*) – Replacements for arguments. Arguments matching keys will be replaced by specified values.
- **evaluators** (*dict, optional*) – Evaluating functions to use instead of the default (which is the *`create()`* classmethod of the argument's class). An argument whose `__class__` equals a key will be evaluated with the specified function.

Examples

```
>>> x = VariableCollection('x')
>>> expr = x(1) + x(2)
>>> print(expr.evaluate())
x(1) + x(2)
>>> expr.evaluate(replace={x(1): 10, x(2): 20})
<friendlysam.opt.Add at 0x...>
>>> print(_)
10 + 20
>>> expr.evaluate(replace={x(1): 10, x(2): 20}, evaluators=fs.CONCRETE_EVALUATORS)
30
```

`Operation.args`	This property holds the arguments of the operation.
`Operation.leaves`	The leaves of the expression tree.
`Operation.value`	The concrete value of the expression, if possible.
`Operation.variables`	This property gives all `leaves` which are instances of *`Variable`*.

friendlysam.opt.Operation.args

Operation.**args**
This property holds the arguments of the operation.

See also *create()*.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = x + y
>>> expr
<friendlysam.opt.Add at 0x...>
>>> expr.args == (x, y)
True
```

```
>>> (x + y) * 2
<friendlysam.opt.Mul at 0x...>
>>> _.args
(<friendlysam.opt.Add at 0x...>, 2)
```

friendlysam.opt.Operation.leaves

Operation.**leaves**
The leaves of the expression tree.

The leaves of an *Operation* are all the *args* which do not themselves have a *leaves* property.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = (42 + x * y * 3.5) * 2
>>> expr.leaves == {42, x, y, 3.5, 2}
True
```

friendlysam.opt.Operation.value

Operation.**value**
The concrete value of the expression, if possible.

This property should only be used when you expect a concrete value. It is computed by calling *evaluate()* with the `evaluators` argument set to `CONCRETE_EVALUATORS`. If the returned value is a number or boolean, it is returned.

Raises : `exc` – *NoValueError* if the expression did not evaluate to a number or boolean.

friendlysam.opt.Operation.variables

Operation.**variables**
This property gives all *leaves* which are instances of *Variable*.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = (42 + x * y * 3.5) * 2
>>> expr.variables == {x, y}
True
```

friendlysam.opt.Add

class friendlysam.opt.**Add**

Addition operator.

See *Operation* for a general description of operations.

Parameters ***args** – Should be exactly two terms to add.

Examples

```
>>> x = VariableCollection('x')
>>> expr = x(1) + x(2)
>>> expr
<friendlysam.opt.Add at 0x...>
>>> expr == Add(x(1), x(2))
True
>>> x(1).value, x(2).value = 2, 3
>>> float(expr)
5.0
```

Add.create(*args)	Classmethod to create a new object.
Add.evaluate([replace, evaluators])	Evaluate the expression recursively.

friendlysam.opt.Add.create

Add.**create**(**args*)

Classmethod to create a new object.

This method is the default evaluator function used in *evaluate()*. Usually you don't want to use this function, but instead the constructor.

Parameters ***args** – The arguments the operation operates on.

Examples

```
>>> x = Variable('x')
>>> args = (2, x)
>>> Add.create(*args) == 2 + x
True
>>> LessEqual.create(*args) == (2 <= x)
True
```

friendlysam.opt.Add.evaluate

Add.**evaluate**(*replace=None*, *evaluators=None*)

Evaluate the expression recursively.

Evaluating an expression:

1. Get an evaluating function. If the class of the present expression is in the `evaluators` dict, use that. Otherwise, take the *`create()`* classmethod of the present expression class.

2. Evaluate all the arguments. For each argument `arg`, first try to replace it by looking for `replace[arg]`. If it's not there, try to evaluate it by calling `arg.evaluate()` with the same arguments supplied to this call. If `arg.evaluate()` is not present, leave the argument unchanged.

3.Run the evaluating function `func(*evaluated_args)` and return the result.

Parameters

- **replace** (*dict, optional*) – Replacements for arguments. Arguments matching keys will be replaced by specified values.
- **evaluators** (*dict, optional*) – Evaluating functions to use instead of the default (which is the *`create()`* classmethod of the argument's class). An argument whose `__class__` equals a key will be evaluated with the specified function.

Examples

```
>>> x = VariableCollection('x')
>>> expr = x(1) + x(2)
>>> print(expr.evaluate())
x(1) + x(2)
>>> expr.evaluate(replace={x(1): 10, x(2): 20})
<friendlysam.opt.Add at 0x...>
>>> print(_)
10 + 20
>>> expr.evaluate(replace={x(1): 10, x(2): 20}, evaluators=fs.CONCRETE_EVALUATORS)
30
```

`Add.args`	This property holds the arguments of the operation.
`Add.leaves`	The leaves of the expression tree.
`Add.value`	The concrete value of the expression, if possible.
`Add.variables`	This property gives all `leaves` which are instances of *`Variable`*.

friendlysam.opt.Add.args

Add.**args**

This property holds the arguments of the operation.

See also *`create()`*.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = x + y
>>> expr
```

```
    <friendlysam.opt.Add at 0x...>
    >>> expr.args == (x, y)
    True
```

```
    >>> (x + y) * 2
    <friendlysam.opt.Mul at 0x...>
    >>> _.args
    (<friendlysam.opt.Add at 0x...>, 2)
```

friendlysam.opt.Add.leaves

Add.**leaves**
The leaves of the expression tree.

The leaves of an *Operation* are all the *args* which do not themselves have a *leaves* property.

Examples

```
    >>> x, y = Variable('x'), Variable('y')
    >>> expr = (42 + x * y * 3.5) * 2
    >>> expr.leaves == {42, x, y, 3.5, 2}
    True
```

friendlysam.opt.Add.value

Add.**value**
The concrete value of the expression, if possible.

This property should only be used when you expect a concrete value. It is computed by calling *evaluate()* with the `evaluators` argument set to `CONCRETE_EVALUATORS`. If the returned value is a number or boolean, it is returned.

Raises : `exc` – *NoValueError* if the expression did not evaluate to a number or boolean.

friendlysam.opt.Add.variables

Add.**variables**
This property gives all *leaves* which are instances of *Variable*.

Examples

```
    >>> x, y = Variable('x'), Variable('y')
    >>> expr = (42 + x * y * 3.5) * 2
    >>> expr.variables == {x, y}
    True
```

friendlysam.opt.Sub

class friendlysam.opt.**Sub**
Subtraction operator.

See *Operation* for a general description of operations.

Parameters ***args** – Should be exactly two items to subtract.

Examples

```
>>> x = VariableCollection('x')
>>> expr = x(1) - x(2)
>>> expr
<friendlysam.opt.Sub at 0x...>
>>> expr == Sub(x(1), x(2))
True
>>> x(1).value, x(2).value = 2, 3
>>> float(expr)
-1.0
```

Sub.create(*args)	Classmethod to create a new object.
Sub.evaluate([replace, evaluators])	Evaluate the expression recursively.

friendlysam.opt.Sub.create

Sub.**create**(**args*)

Classmethod to create a new object.

This method is the default evaluator function used in *evaluate()*. Usually you don't want to use this function, but instead the constructor.

Parameters ***args** – The arguments the operation operates on.

Examples

```
>>> x = Variable('x')
>>> args = (2, x)
>>> Add.create(*args) == 2 + x
True
>>> LessEqual.create(*args) == (2 <= x)
True
```

friendlysam.opt.Sub.evaluate

Sub.**evaluate**(*replace=None*, *evaluators=None*)

Evaluate the expression recursively.

Evaluating an expression:

1. Get an evaluating function. If the class of the present expression is in the `evaluators` dict, use that. Otherwise, take the *create()* classmethod of the present expression class.

2. Evaluate all the arguments. For each argument `arg`, first try to replace it by looking for `replace[arg]`. If it's not there, try to evaluate it by calling `arg.evaluate()` with the same arguments supplied to this call. If `arg.evaluate()` is not present, leave the argument unchanged.

3.Run the evaluating function `func(*evaluated_args)` and return the result.

Parameters

- **replace** (*dict, optional*) – Replacements for arguments. Arguments matching keys will be replaced by specified values.
- **evaluators** (*dict, optional*) – Evaluating functions to use instead of the default (which is the *create()* classmethod of the argument's class). An argument whose `__class__` equals a key will be evaluated with the specified function.

Examples

```
>>> x = VariableCollection('x')
>>> expr = x(1) + x(2)
>>> print(expr.evaluate())
x(1) + x(2)
>>> expr.evaluate(replace={x(1): 10, x(2): 20})
<friendlysam.opt.Add at 0x...>
>>> print(_)
10 + 20
>>> expr.evaluate(replace={x(1): 10, x(2): 20}, evaluators=fs.CONCRETE_EVALUATORS)
30
```

Sub.args	This property holds the arguments of the operation.
Sub.leaves	The leaves of the expression tree.
Sub.value	The concrete value of the expression, if possible.
Sub.variables	This property gives all `leaves` which are instances of *Variable*.

friendlysam.opt.Sub.args

`Sub.`**`args`**

This property holds the arguments of the operation.

See also *create()*.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = x + y
>>> expr
<friendlysam.opt.Add at 0x...>
>>> expr.args == (x, y)
True
```

```
>>> (x + y) * 2
<friendlysam.opt.Mul at 0x...>
>>> _.args
(<friendlysam.opt.Add at 0x...>, 2)
```

friendlysam.opt.Sub.leaves

`Sub.`**`leaves`**

The leaves of the expression tree.

The leaves of an *Operation* are all the *args* which do not themselves have a *leaves* property.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = (42 + x * y * 3.5) * 2
>>> expr.leaves == {42, x, y, 3.5, 2}
True
```

friendlysam.opt.Sub.value

Sub.**value**

The concrete value of the expression, if possible.

This property should only be used when you expect a concrete value. It is computed by calling *evaluate()* with the `evaluators` argument set to `CONCRETE_EVALUATORS`. If the returned value is a number or boolean, it is returned.

Raises : `exc` – *NoValueError* if the expression did not evaluate to a number or boolean.

friendlysam.opt.Sub.variables

Sub.**variables**

This property gives all *leaves* which are instances of *Variable*.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = (42 + x * y * 3.5) * 2
>>> expr.variables == {x, y}
True
```

friendlysam.opt.Mul

class friendlysam.opt.**Mul**

Subtraction operator.

See *Operation* for a general description of operations.

Parameters ***args** – Should be exactly two terms to multiply.

Examples

```
>>> x = VariableCollection('x')
>>> expr = x(1) * x(2)
>>> expr
<friendlysam.opt.Mul at 0x...>
>>> expr == Mul(x(1), x(2))
True
>>> x(1).value, x(2).value = 2, 3
```

```
>>> float(expr)
6.0
```

Note: There is currently no division operator, but the operator `/` is overloaded such that `x = a / b` is equivalent to `x = a * (1/b)`. Hence, you can do simple things like

```
>>> print(x(1) / 4)
x(1) * 0.25
```

Mul.create(*args)	Classmethod to create a new object.
Mul.evaluate([replace, evaluators])	Evaluate the expression recursively.

friendlysam.opt.Mul.create

Mul.**create**(**args*)

Classmethod to create a new object.

This method is the default evaluator function used in *evaluate()*. Usually you don't want to use this function, but instead the constructor.

Parameters ***args** – The arguments the operation operates on.

Examples

```
>>> x = Variable('x')
>>> args = (2, x)
>>> Add.create(*args) == 2 + x
True
>>> LessEqual.create(*args) == (2 <= x)
True
```

friendlysam.opt.Mul.evaluate

Mul.**evaluate**(*replace=None*, *evaluators=None*)

Evaluate the expression recursively.

Evaluating an expression:

1. Get an evaluating function. If the class of the present expression is in the `evaluators` dict, use that. Otherwise, take the *create()* classmethod of the present expression class.

2. Evaluate all the arguments. For each argument `arg`, first try to replace it by looking for `replace[arg]`. If it's not there, try to evaluate it by calling `arg.evaluate()` with the same arguments supplied to this call. If `arg.evaluate()` is not present, leave the argument unchanged.

3.Run the evaluating function `func(*evaluated_args)` and return the result.

Parameters

- **replace** (*dict, optional*) – Replacements for arguments. Arguments matching keys will be replaced by specified values.

- **evaluators** (*dict, optional*) – Evaluating functions to use instead of the default (which is the *create()* classmethod of the argument's class). An argument whose __class__ equals a key will be evaluated with the specified function.

Examples

```
>>> x = VariableCollection('x')
>>> expr = x(1) + x(2)
>>> print(expr.evaluate())
x(1) + x(2)
>>> expr.evaluate(replace={x(1): 10, x(2): 20})
<friendlysam.opt.Add at 0x...>
>>> print(_)
10 + 20
>>> expr.evaluate(replace={x(1): 10, x(2): 20}, evaluators=fs.CONCRETE_EVALUATORS)
30
```

Mul.args	This property holds the arguments of the operation.
Mul.leaves	The leaves of the expression tree.
Mul.value	The concrete value of the expression, if possible.
Mul.variables	This property gives all leaves which are instances of *Variable*.

friendlysam.opt.Mul.args

Mul.**args**

This property holds the arguments of the operation.

See also *create()*.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = x + y
>>> expr
<friendlysam.opt.Add at 0x...>
>>> expr.args == (x, y)
True
```

```
>>> (x + y) * 2
<friendlysam.opt.Mul at 0x...>
>>> _.args
(<friendlysam.opt.Add at 0x...>, 2)
```

friendlysam.opt.Mul.leaves

Mul.**leaves**

The leaves of the expression tree.

The leaves of an *Operation* are all the *args* which do not themselves have a *leaves* property.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = (42 + x * y * 3.5) * 2
>>> expr.leaves == {42, x, y, 3.5, 2}
True
```

friendlysam.opt.Mul.value

Mul.**value**
The concrete value of the expression, if possible.

This property should only be used when you expect a concrete value. It is computed by calling *evaluate()* with the `evaluators` argument set to `CONCRETE_EVALUATORS`. If the returned value is a number or boolean, it is returned.

Raises : `exc` – *NoValueError* if the expression did not evaluate to a number or boolean.

friendlysam.opt.Mul.variables

Mul.**variables**
This property gives all *leaves* which are instances of *Variable*.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = (42 + x * y * 3.5) * 2
>>> expr.variables == {x, y}
True
```

friendlysam.opt.Sum

class friendlysam.opt.**Sum**
A sum of items.

See the base class *Operation* for a basic description of attributes and methods.

Examples

Note that the constructor takes an iterable of arguments, just like the built-in `sum()` function, but the class-method *create()* takes a list of arguments, as follows.

```
>>> x = VariableCollection('x')
>>> terms = [x(i) for i in range(4)]
>>> Sum(terms) == Sum.create(*terms)
True
```

```
>>> s = Sum(terms)
>>> s.evaluate(evaluators={Sum: sum})
Traceback (most recent call last):
...
TypeError: sum expected at most 2 arguments, got 4
```

```
>>> s.evaluate(evaluators={Sum: lambda *args: sum(args)})
<friendlysam.opt.Add at 0x...>
```

Sum.create(*args)	Classmethod to create a new Sum object.
Sum.evaluate([replace, evaluators])	Evaluate the expression recursively.

friendlysam.opt.Sum.create

classmethod Sum.**create**(**args*)

Classmethod to create a new Sum object.

Note that *create()* has a different signature than the constructor. The constructor takes an iterable as only argument, but *create()* takes a list of arguments.

Example

```
>>> x = VariableCollection('x')
>>> terms = [x(i) for i in range(4)]
>>> Sum(terms) == Sum.create(*terms)
True
```

friendlysam.opt.Sum.evaluate

Sum.**evaluate**(*replace=None*, *evaluators=None*)

Evaluate the expression recursively.

Evaluating an expression:

1. Get an evaluating function. If the class of the present expression is in the `evaluators` dict, use that. Otherwise, take the *create()* classmethod of the present expression class.

2. Evaluate all the arguments. For each argument `arg`, first try to replace it by looking for `replace[arg]`. If it's not there, try to evaluate it by calling `arg.evaluate()` with the same arguments supplied to this call. If `arg.evaluate()` is not present, leave the argument unchanged.

3.Run the evaluating function `func(*evaluated_args)` and return the result.

Parameters

- **replace** (*dict, optional*) – Replacements for arguments. Arguments matching keys will be replaced by specified values.
- **evaluators** (*dict, optional*) – Evaluating functions to use instead of the default (which is the *create()* classmethod of the argument's class). An argument whose `__class__` equals a key will be evaluated with the specified function.

Examples

```
>>> x = VariableCollection('x')
>>> expr = x(1) + x(2)
>>> print(expr.evaluate())
x(1) + x(2)
>>> expr.evaluate(replace={x(1): 10, x(2): 20})
```

```
    <friendlysam.opt.Add at 0x...>
    >>> print(_)
    10 + 20
    >>> expr.evaluate(replace={x(1): 10, x(2): 20}, evaluators=fs.CONCRETE_EVALUATORS)
    30
```

Sum.args	This property holds the arguments of the operation.
Sum.leaves	The leaves of the expression tree.
Sum.value	The concrete value of the expression, if possible.
Sum.variables	This property gives all `leaves` which are instances of *Variable*.

friendlysam.opt.Sum.args

Sum.**args**

This property holds the arguments of the operation.

See also *create()*.

Examples

```
    >>> x, y = Variable('x'), Variable('y')
    >>> expr = x + y
    >>> expr
    <friendlysam.opt.Add at 0x...>
    >>> expr.args == (x, y)
    True
```

```
    >>> (x + y) * 2
    <friendlysam.opt.Mul at 0x...>
    >>> _.args
    (<friendlysam.opt.Add at 0x...>, 2)
```

friendlysam.opt.Sum.leaves

Sum.**leaves**

The leaves of the expression tree.

The leaves of an *Operation* are all the *args* which do not themselves have a *leaves* property.

Examples

```
    >>> x, y = Variable('x'), Variable('y')
    >>> expr = (42 + x * y * 3.5) * 2
    >>> expr.leaves == {42, x, y, 3.5, 2}
    True
```

friendlysam.opt.Sum.value

Sum.**value**

The concrete value of the expression, if possible.

This property should only be used when you expect a concrete value. It is computed by calling *evaluate()* with the `evaluators` argument set to `CONCRETE_EVALUATORS`. If the returned value is a number or boolean, it is returned.

Raises : `exc` – *NoValueError* if the expression did not evaluate to a number or boolean.

friendlysam.opt.Sum.variables

`Sum.`**`variables`**

This property gives all *leaves* which are instances of *Variable*.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = (42 + x * y * 3.5) * 2
>>> expr.variables == {x, y}
True
```

friendlysam.opt.dot

`friendlysam.opt.`**`dot`**(*a*, *b*)

Make expression for the scalar product of two vectors.

`dot(a, b)` is equivalent to `Sum(ai * bi for ai, bi in zip(a, b))`.

Returns An expression.

Examples

```
>>> n = 10
>>> coefficients = (2 ** i for i in range(n))
>>> x = VariableCollection('x')
>>> vars = [x(i) for i in range(n)]
>>> dot(coefficients, vars)
<friendlysam.opt.Sum at 0x...>
```

friendlysam.opt.Relation

class `friendlysam.opt.`**`Relation`**

Base class for binary relations.

See child classes:

Less *LessEqual* *Eq*

Relation.create(*args)	Classmethod to create a new object.
Relation.evaluate([replace, evaluators])	Evaluate the expression recursively.

friendlysam.opt.Relation.create

Relation.**create**(**args*)

Classmethod to create a new object.

This method is the default evaluator function used in *evaluate()*. Usually you don't want to use this function, but instead the constructor.

Parameters ***args** – The arguments the operation operates on.

Examples

```
>>> x = Variable('x')
>>> args = (2, x)
>>> Add.create(*args) == 2 + x
True
>>> LessEqual.create(*args) == (2 <= x)
True
```

friendlysam.opt.Relation.evaluate

Relation.**evaluate**(*replace=None*, *evaluators=None*)

Evaluate the expression recursively.

Evaluating an expression:

1. Get an evaluating function. If the class of the present expression is in the `evaluators` dict, use that. Otherwise, take the *create()* classmethod of the present expression class.

2. Evaluate all the arguments. For each argument `arg`, first try to replace it by looking for `replace[arg]`. If it's not there, try to evaluate it by calling `arg.evaluate()` with the same arguments supplied to this call. If `arg.evaluate()` is not present, leave the argument unchanged.

3.Run the evaluating function `func(*evaluated_args)` and return the result.

Parameters

- **replace** (*dict, optional*) – Replacements for arguments. Arguments matching keys will be replaced by specified values.
- **evaluators** (*dict, optional*) – Evaluating functions to use instead of the default (which is the *create()* classmethod of the argument's class). An argument whose `__class__` equals a key will be evaluated with the specified function.

Examples

```
>>> x = VariableCollection('x')
>>> expr = x(1) + x(2)
>>> print(expr.evaluate())
x(1) + x(2)
>>> expr.evaluate(replace={x(1): 10, x(2): 20})
<friendlysam.opt.Add at 0x...>
>>> print(_)
10 + 20
>>> expr.evaluate(replace={x(1): 10, x(2): 20}, evaluators=fs.CONCRETE_EVALUATORS)
30
```

Relation.args	This property holds the arguments of the operation.
Relation.leaves	The leaves of the expression tree.
Relation.value	The concrete value of the expression, if possible.
Relation.variables	This property gives all `leaves` which are instances of *Variable*.

friendlysam.opt.Relation.args

`Relation.`**`args`**

This property holds the arguments of the operation.

See also *create()*.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = x + y
>>> expr
<friendlysam.opt.Add at 0x...>
>>> expr.args == (x, y)
True
```

```
>>> (x + y) * 2
<friendlysam.opt.Mul at 0x...>
>>> _.args
(<friendlysam.opt.Add at 0x...>, 2)
```

friendlysam.opt.Relation.leaves

`Relation.`**`leaves`**

The leaves of the expression tree.

The leaves of an *Operation* are all the *args* which do not themselves have a *leaves* property.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = (42 + x * y * 3.5) * 2
>>> expr.leaves == {42, x, y, 3.5, 2}
True
```

friendlysam.opt.Relation.value

`Relation.`**`value`**

The concrete value of the expression, if possible.

This property should only be used when you expect a concrete value. It is computed by calling *evaluate()* with the `evaluators` argument set to `CONCRETE_EVALUATORS`. If the returned value is a number or boolean, it is returned.

Raises : `exc` – *NoValueError* if the expression did not evaluate to a number or boolean.

friendlysam.opt.Relation.variables

Relation.**variables**
This property gives all *leaves* which are instances of *Variable*.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = (42 + x * y * 3.5) * 2
>>> expr.variables == {x, y}
True
```

friendlysam.opt.Less

class friendlysam.opt.**Less**
The relation "less than".

Examples

```
>>> x = Variable('x')
>>> expr = (x < 1)
>>> expr
<friendlysam.opt.Less at 0x...>
>>> expr == Less(x, 1)
True
>>> x.value = 1
>>> expr.value
False
```

Note: There is no Greater class, but you can use the overloaded operator >.

```
>>> x > 1
<friendlysam.opt.Less at 0x...>
>>> print(_)
1 < x
>>> (x > 1) == (1 < x)
True
```

Less.create(*args)	Classmethod to create a new object.
Less.evaluate([replace, evaluators])	Evaluate the expression recursively.

friendlysam.opt.Less.create

Less.**create**(**args*)
Classmethod to create a new object.

This method is the default evaluator function used in *evaluate()*. Usually you don't want to use this function, but instead the constructor.

Parameters ***args** – The arguments the operation operates on.

Examples

```
>>> x = Variable('x')
>>> args = (2, x)
>>> Add.create(*args) == 2 + x
True
>>> LessEqual.create(*args) == (2 <= x)
True
```

friendlysam.opt.Less.evaluate

`Less.`**`evaluate`**(*replace=None*, *evaluators=None*)

Evaluate the expression recursively.

Evaluating an expression:

1. Get an evaluating function. If the class of the present expression is in the `evaluators` dict, use that. Otherwise, take the *`create()`* classmethod of the present expression class.

2. Evaluate all the arguments. For each argument `arg`, first try to replace it by looking for `replace[arg]`. If it's not there, try to evaluate it by calling `arg.evaluate()` with the same arguments supplied to this call. If `arg.evaluate()` is not present, leave the argument unchanged.

3.Run the evaluating function `func(*evaluated_args)` and return the result.

Parameters

- **`replace`** (*dict, optional*) – Replacements for arguments. Arguments matching keys will be replaced by specified values.
- **`evaluators`** (*dict, optional*) – Evaluating functions to use instead of the default (which is the *`create()`* classmethod of the argument's class). An argument whose `__class__` equals a key will be evaluated with the specified function.

Examples

```
>>> x = VariableCollection('x')
>>> expr = x(1) + x(2)
>>> print(expr.evaluate())
x(1) + x(2)
>>> expr.evaluate(replace={x(1): 10, x(2): 20})
<friendlysam.opt.Add at 0x...>
>>> print(_)
10 + 20
>>> expr.evaluate(replace={x(1): 10, x(2): 20}, evaluators=fs.CONCRETE_EVALUATORS)
30
```

`Less.args`	This property holds the arguments of the operation.
`Less.leaves`	The leaves of the expression tree.
`Less.value`	The concrete value of the expression, if possible.
`Less.variables`	This property gives all `leaves` which are instances of *`Variable`*.

friendlysam.opt.Less.args

Less.**args**
This property holds the arguments of the operation.

See also *create()*.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = x + y
>>> expr
<friendlysam.opt.Add at 0x...>
>>> expr.args == (x, y)
True
```

```
>>> (x + y) * 2
<friendlysam.opt.Mul at 0x...>
>>> _.args
(<friendlysam.opt.Add at 0x...>, 2)
```

friendlysam.opt.Less.leaves

Less.**leaves**
The leaves of the expression tree.

The leaves of an *Operation* are all the *args* which do not themselves have a *leaves* property.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = (42 + x * y * 3.5) * 2
>>> expr.leaves == {42, x, y, 3.5, 2}
True
```

friendlysam.opt.Less.value

Less.**value**
The concrete value of the expression, if possible.

This property should only be used when you expect a concrete value. It is computed by calling *evaluate()* with the `evaluators` argument set to `CONCRETE_EVALUATORS`. If the returned value is a number or boolean, it is returned.

Raises : `exc` – *NoValueError* if the expression did not evaluate to a number or boolean.

friendlysam.opt.Less.variables

Less.**variables**
This property gives all *leaves* which are instances of *Variable*.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = (42 + x * y * 3.5) * 2
>>> expr.variables == {x, y}
True
```

friendlysam.opt.LessEqual

class friendlysam.opt.**LessEqual**
The relation "less than or equal to".

Examples

```
>>> x = Variable('x')
>>> expr = (x <= 1)
>>> expr
<friendlysam.opt.LessEqual at 0x...>
>>> expr == LessEqual(x, 1)
True
>>> x.value = 1
>>> expr.value
True
```

Note: There is no GreaterEqual class, but you can use the overloaded operator >=.

```
>>> x >= 1
<friendlysam.opt.LessEqual at 0x...>
>>> print(_)
1 <= x
>>> (x >= 1) == (1 <= x)
True
```

LessEqual.create(*args)	Classmethod to create a new object.
LessEqual.evaluate([replace, evaluators])	Evaluate the expression recursively.

friendlysam.opt.LessEqual.create

LessEqual.**create**(**args*)
Classmethod to create a new object.

This method is the default evaluator function used in *evaluate()*. Usually you don't want to use this function, but instead the constructor.

Parameters ***args** – The arguments the operation operates on.

Examples

```
>>> x = Variable('x')
>>> args = (2, x)
>>> Add.create(*args) == 2 + x
```

```
    True
    >>> LessEqual.create(*args) == (2 <= x)
    True
```

friendlysam.opt.LessEqual.evaluate

LessEqual.**evaluate**(*replace=None*, *evaluators=None*)

Evaluate the expression recursively.

Evaluating an expression:

1. Get an evaluating function. If the class of the present expression is in the `evaluators` dict, use that. Otherwise, take the *`create()`* classmethod of the present expression class.

2. Evaluate all the arguments. For each argument `arg`, first try to replace it by looking for `replace[arg]`. If it's not there, try to evaluate it by calling `arg.evaluate()` with the same arguments supplied to this call. If `arg.evaluate()` is not present, leave the argument unchanged.

3.Run the evaluating function `func(*evaluated_args)` and return the result.

Parameters

- **replace** (*dict, optional*) – Replacements for arguments. Arguments matching keys will be replaced by specified values.
- **evaluators** (*dict, optional*) – Evaluating functions to use instead of the default (which is the *`create()`* classmethod of the argument's class). An argument whose `__class__` equals a key will be evaluated with the specified function.

Examples

```
    >>> x = VariableCollection('x')
    >>> expr = x(1) + x(2)
    >>> print(expr.evaluate())
    x(1) + x(2)
    >>> expr.evaluate(replace={x(1): 10, x(2): 20})
    <friendlysam.opt.Add at 0x...>
    >>> print(_)
    10 + 20
    >>> expr.evaluate(replace={x(1): 10, x(2): 20}, evaluators=fs.CONCRETE_EVALUATORS)
    30
```

`LessEqual.args`	This property holds the arguments of the operation.
`LessEqual.leaves`	The leaves of the expression tree.
`LessEqual.value`	The concrete value of the expression, if possible.
`LessEqual.variables`	This property gives all `leaves` which are instances of *`Variable`*.

friendlysam.opt.LessEqual.args

LessEqual.**args**

This property holds the arguments of the operation.

See also *`create()`*.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = x + y
>>> expr
<friendlysam.opt.Add at 0x...>
>>> expr.args == (x, y)
True
```

```
>>> (x + y) * 2
<friendlysam.opt.Mul at 0x...>
>>> _.args
(<friendlysam.opt.Add at 0x...>, 2)
```

friendlysam.opt.LessEqual.leaves

LessEqual.**leaves**

The leaves of the expression tree.

The leaves of an *Operation* are all the *args* which do not themselves have a *leaves* property.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = (42 + x * y * 3.5) * 2
>>> expr.leaves == {42, x, y, 3.5, 2}
True
```

friendlysam.opt.LessEqual.value

LessEqual.**value**

The concrete value of the expression, if possible.

This property should only be used when you expect a concrete value. It is computed by calling *evaluate()* with the `evaluators` argument set to `CONCRETE_EVALUATORS`. If the returned value is a number or boolean, it is returned.

Raises : `exc` – *NoValueError* if the expression did not evaluate to a number or boolean.

friendlysam.opt.LessEqual.variables

LessEqual.**variables**

This property gives all *leaves* which are instances of *Variable*.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = (42 + x * y * 3.5) * 2
>>> expr.variables == {x, y}
True
```

friendlysam.opt.Eq

class `friendlysam.opt.`**`Eq`**

The relation "equals".

> **Warning:** The operator == is reserved for checking object similarity, just like we are used to in Python. To create the relation "x equals y", use `Eq`:
>
> ```
> >>> from friendlysam import Eq
> >>> my_var = Variable('x')
> >>> my_var == 1
> False
> >>> print(Eq(my_var, 1))
> x == 1
> ```

Examples

```
>>> x = Variable('x')
>>> x == 3 # Don't do this!
False
```

```
>>> equality = Eq(x, 3) # Do this instead.
>>> equality
<friendlysam.opt.Eq at 0x...>
>>> x.value = 3
>>> equality.value
True
>>> x.value = 4
>>> equality.value
False
```

Eq.create(*args)	Classmethod to create a new object.
Eq.evaluate([replace, evaluators])	Evaluate the expression recursively.

friendlysam.opt.Eq.create

`Eq.`**`create`**`(`**args*`)`

Classmethod to create a new object.

This method is the default evaluator function used in *evaluate()*. Usually you don't want to use this function, but instead the constructor.

Parameters **`*args`** – The arguments the operation operates on.

Examples

```
>>> x = Variable('x')
>>> args = (2, x)
>>> Add.create(*args) == 2 + x
True
>>> LessEqual.create(*args) == (2 <= x)
True
```

friendlysam.opt.Eq.evaluate

Eq.**evaluate**(*replace=None*, *evaluators=None*)

Evaluate the expression recursively.

Evaluating an expression:

1. Get an evaluating function. If the class of the present expression is in the `evaluators` dict, use that. Otherwise, take the *`create()`* classmethod of the present expression class.

2. Evaluate all the arguments. For each argument `arg`, first try to replace it by looking for `replace[arg]`. If it's not there, try to evaluate it by calling `arg.evaluate()` with the same arguments supplied to this call. If `arg.evaluate()` is not present, leave the argument unchanged.

3.Run the evaluating function `func(*evaluated_args)` and return the result.

Parameters

- **replace** (*dict, optional*) – Replacements for arguments. Arguments matching keys will be replaced by specified values.
- **evaluators** (*dict, optional*) – Evaluating functions to use instead of the default (which is the *`create()`* classmethod of the argument's class). An argument whose `__class__` equals a key will be evaluated with the specified function.

Examples

```
>>> x = VariableCollection('x')
>>> expr = x(1) + x(2)
>>> print(expr.evaluate())
x(1) + x(2)
>>> expr.evaluate(replace={x(1): 10, x(2): 20})
<friendlysam.opt.Add at 0x...>
>>> print(_)
10 + 20
>>> expr.evaluate(replace={x(1): 10, x(2): 20}, evaluators=fs.CONCRETE_EVALUATORS)
30
```

`Eq.args`	This property holds the arguments of the operation.
`Eq.leaves`	The leaves of the expression tree.
`Eq.value`	The concrete value of the expression, if possible.
`Eq.variables`	This property gives all `leaves` which are instances of *`Variable`*.

friendlysam.opt.Eq.args

Eq.**args**

This property holds the arguments of the operation.

See also *`create()`*.

Examples

```
>>> x, y = Variable('x'), Variable('y')
>>> expr = x + y
>>> expr
```

```
    <friendlysam.opt.Add at 0x...>
    >>> expr.args == (x, y)
    True
```

```
    >>> (x + y) * 2
    <friendlysam.opt.Mul at 0x...>
    >>> _.args
    (<friendlysam.opt.Add at 0x...>, 2)
```

friendlysam.opt.Eq.leaves

Eq.**leaves**

The leaves of the expression tree.

The leaves of an *Operation* are all the *args* which do not themselves have a *leaves* property.

Examples

```
    >>> x, y = Variable('x'), Variable('y')
    >>> expr = (42 + x * y * 3.5) * 2
    >>> expr.leaves == {42, x, y, 3.5, 2}
    True
```

friendlysam.opt.Eq.value

Eq.**value**

The concrete value of the expression, if possible.

This property should only be used when you expect a concrete value. It is computed by calling *evaluate()* with the `evaluators` argument set to `CONCRETE_EVALUATORS`. If the returned value is a number or boolean, it is returned.

Raises : `exc` – *NoValueError* if the expression did not evaluate to a number or boolean.

friendlysam.opt.Eq.variables

Eq.**variables**

This property gives all *leaves* which are instances of *Variable*.

Examples

```
    >>> x, y = Variable('x'), Variable('y')
    >>> expr = (42 + x * y * 3.5) * 2
    >>> expr.variables == {x, y}
    True
```

2.9.3 Constraints and optimization

get_solver([engine, options])	Get a solver object.
Problem()	An optimization problem.
Maximize(expr)	A maximization objective.
Minimize(expr)	A minimization objective.
Constraint(expr[, desc, origin])	An equality or inequality constraint.
SOS1(variables, **kwargs)	Special ordered set, type 1
SOS2(variables, **kwargs)	Special ordered set, type 2
piecewise_affine(points[, name])	Create a piecewise affine expression and constraints.
piecewise_affine_constraints(variables[, ...])	Constrains for a piecewise affine expression.

friendlysam.opt.get_solver

friendlysam.opt.**get_solver**(*engine='pulp'*, *options=None*)

Get a solver object.

Parameters

- **engine** (*str, optional*) – Which engine to use.
- **options** (*dict, optional*) – Parameters to the engine constructor.

  If `engine == 'pulp'`, the engine is created using `PulpSolver(options)`. See `PulpSolver` constructor for details.

friendlysam.opt.Problem

class friendlysam.opt.**Problem**

An optimization problem.

The problem class is essentially a container for an objective function and a set of constraints.

Examples

```
>>> x = VariableCollection('x')
>>> prob = Problem()
>>> prob.objective = Maximize(x(1) + x(2))
>>> prob.add(8 * x(1) + 4 * x(2) <= 11)
>>> prob.add(2 * x(1) + 4 * x(2) <= 5)
>>>
>>> # Get a solver and solve the problem
>>> solver = fs.get_solver()
>>> solution = solver.solve(prob)
>>> type(solution)
<class 'dict'>
>>> solution[x(1)]
1.0
>>> solution[x(2)]
0.75
```

Problem.add(*constraints)	Add zero or more constraints to the problem.
Problem.variables_without_value()	Get all *Variable* instances without value.

friendlysam.opt.Problem.add

Problem.**add**(**constraints*)

Add zero or more constraints to the problem.

Parameters ***constraints** – zero or more constraints or iterables of constraints. Each constraint should be an instance of *Relation*, *Constraint*, *SOS1* or *SOS2*.

Note: The syntax `problem += constraints` is equivalent to ‘‘problem.add(constraints).

Examples

```
>>> prob = Problem()
>>> x = VariableCollection('x')
```

```
>>> prob.add(8 * x(1) + 4 * x(2) <= 11)
```

```
>>> prob += Constraint(x(0) <= x(1), desc='Some description')
```

```
>>> prob += ([x(i) <= i, x(i+1) <= i] for i in range(5))
```

friendlysam.opt.Problem.variables_without_value

Problem.**variables_without_value**()

Get all *Variable* instances without value.

These are effectively the variables of the optimization problem.

Problem.constraints	A set of constraints.
Problem.objective	

friendlysam.opt.Problem.constraints

Problem.**constraints**

A set of constraints.

To add constraints, use *Problem.add()*.

friendlysam.opt.Problem.objective

Problem.**objective**

friendlysam.opt.Maximize

class friendlysam.opt.**Maximize**(*expr*)

A maximization objective.

Parameters **expr** (expression or *Variable* instance) – An expression to maximize.

Examples

```
>>> x = VariableCollection('x')
>>> prob = Problem()
>>> prob.objective = Maximize(Sum(x(i) for i in range(50)))
```

Maximize.expr	
Maximize.variables	Read only property, shorthand for `.expr.variables`

friendlysam.opt.Maximize.expr

`Maximize.`**`expr`** = None

friendlysam.opt.Maximize.variables

`Maximize.`**`variables`**
Read only property, shorthand for `.expr.variables`

friendlysam.opt.Minimize

class `friendlysam.opt.`**`Minimize`**(*expr*)
A minimization objective.

Parameters **`expr`** (expression or *`Variable`* instance) – An expression to minimize.

Examples

```
>>> x = VariableCollection('x')
>>> prob = Problem()
>>> prob.objective = Minimize(Sum(x(i) for i in range(50)))
```

Minimize.expr	
Minimize.variables	Read only property, shorthand for `.expr.variables`

friendlysam.opt.Minimize.expr

`Minimize.`**`expr`** = None

friendlysam.opt.Minimize.variables

`Minimize.`**`variables`**
Read only property, shorthand for `.expr.variables`

friendlysam.opt.Constraint

class `friendlysam.opt.`**`Constraint`**(*expr*, *desc=None*, *origin=None*)

An equality or inequality constraint.

This class is used to wrap a constraint expression and (optionally) add some metadata.

Parameters

- **`expr`** (*`Relation`* instance) – An equality or inequality.
- **`desc`** (*str, optional*) – A text describing the constraint.
- **`origin`** (*anything, optional*) – Some object describing where the constraint comes from.

Examples

```
>>> x = Variable('x')
>>> c = Constraint(x + 1 <= 2 * x, desc='Some text')
>>> print(c)
<Constraint: Some text>
>>> c.origin = 'randomly created'
>>> print(c)
<Constraint [randomly created]: Some text>
>>> print(c.expr)
x + 1 <= 2 * x
```

`Constraint.desc`	A description of the constraint, for debugging.
`Constraint.long_description`	A long, human-readable string representation of the constraint.
`Constraint.origin`	The origin of the description.
`Constraint.variables`	Read only property, shorthand for `.expr.variables`

friendlysam.opt.Constraint.desc

`Constraint.`**`desc`**

A description of the constraint, for debugging.

friendlysam.opt.Constraint.long_description

`Constraint.`**`long_description`**

A long, human-readable string representation of the constraint.

The description includes the `repr()` of the *`Constraint`*, the *`desc`*, and the *`origin`*.

It is broken into three lines.

friendlysam.opt.Constraint.origin

`Constraint.`**`origin`**

The origin of the description.

Can be any object. Supposed to indicate where the constraint comes from, for debugging.

friendlysam.opt.Constraint.variables

Constraint.**variables**
Read only property, shorthand for `.expr.variables`

friendlysam.opt.SOS1

class friendlysam.opt.**SOS1**(*variables*, ***kwargs*)
Special ordered set, type 1

An ordered set of variables, of which **at most one** may be nonzero.

Add a *SOS1* instance to an optimization problem just like a *Constraint* to enforce this condition.

Parameters

- **variables** (sequence of *Variable* instances) – The variables in the ordered set. Must be an ordered sequence (today `list` and `tuple` are allowed).
- **desc** (*str, optional*) – A text describing the constraint.
- **origin** (*anything, optional*) – Some object describing where the constraint comes from.

SOS1.desc	A description of the constraint, for debugging.
SOS1.level	A number indicating SOS1 (level=1) or SOS2 (level=2).
SOS1.origin	The origin of the description.
SOS1.variables	The ordered list of variables as a tuple.

friendlysam.opt.SOS1.desc

SOS1.**desc**
A description of the constraint, for debugging.

friendlysam.opt.SOS1.level

SOS1.**level**
A number indicating SOS1 (level=1) or SOS2 (level=2).

friendlysam.opt.SOS1.origin

SOS1.**origin**
The origin of the description.

Can be any object. Supposed to indicate where the constraint comes from, for debugging.

friendlysam.opt.SOS1.variables

SOS1.**variables**
The ordered list of variables as a tuple.

friendlysam.opt.SOS2

class `friendlysam.opt.`**`SOS2`**(*variables*, ***kwargs*)
Special ordered set, type 2

An ordered set of variables, of which **at most two** may be nonzero. Any nonzero variables must be adjacent in order.

Add a *SOS2* instance to an optimization problem just like a *Constraint* to enforce this condition.

Parameters

- **variables** (sequence of *Variable* instances) – The variables in the ordered set. Must be an ordered sequence (today `list` and `tuple` are allowed).
- **desc** (*str, optional*) – A text describing the constraint.
- **origin** (*anything, optional*) – Some object describing where the constraint comes from.

SOS2.desc	A description of the constraint, for debugging.
SOS2.level	A number indicating SOS1 (level=1) or SOS2 (level=2).
SOS2.origin	The origin of the description.
SOS2.variables	The ordered list of variables as a tuple.

friendlysam.opt.SOS2.desc

`SOS2.`**`desc`**
A description of the constraint, for debugging.

friendlysam.opt.SOS2.level

`SOS2.`**`level`**
A number indicating SOS1 (level=1) or SOS2 (level=2).

friendlysam.opt.SOS2.origin

`SOS2.`**`origin`**
The origin of the description.

Can be any object. Supposed to indicate where the constraint comes from, for debugging.

friendlysam.opt.SOS2.variables

`SOS2.`**`variables`**
The ordered list of variables as a tuple.

friendlysam.opt.piecewise_affine

`friendlysam.opt.`**`piecewise_affine`**(*points*, *name=None*)
Create a piecewise affine expression and constraints.

There are several ways to express piecewise affine functions in MILP problems. This function helps with one of them, using SOS2 variables.

Definition:

$f(x)$ is the linear interpolation of a data set $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$.

The x_i are ordered: $x_0 \leq x_1 \leq \ldots \leq x_n$.

See http://en.wikipedia.org/wiki/Linear_interpolation

Parameters

- **points** (*dict or sequence of pairs*) – The x_i, y_i pairs.

  Alternative 1: Provide a `dict`, e.g. `{x0: y0, x1: y1, ..., x_n: y_n}`.

  Alternative 2: Provide a sequence of pairs, e.g. `[(x0, y0), (x1, y1), ..., (xn, yn)]`

  The points are automatically sorted in increasing `x_i`.

- **name** (*str, optional*) – A name base for the variables.

Returns

`(x, y, constraints)`

`x` is an expression for the argument of the function.

`y` is an expression for the function value.

`constraints` is a set of *SOS2* and *Constraint* instances that must be added to an optimization problem to enforce the relation between `x` and `y`.

Examples

```
>>> points = {1: 30, 1.5: 20, 2: 40}
>>> x, y, constraints = fs.piecewise_affine(points, name='pwa_vars')
>>> prob = fs.Problem()
>>> prob.objective = fs.Minimize(y)
>>> prob.add(constraints)
>>> solution = get_solver().solve(prob)
>>> for var in x.variables:
...     var.take_value(solution)
...
>>> float(x) == 1.5
True
>>> float(y) == 20
True
```

friendlysam.opt.piecewise_affine_constraints

`friendlysam.opt.`**`piecewise_affine_constraints`**(*variables*, *include_lb=True*)

Constrains for a piecewise affine expression.

For some variables $x_0, x_1, \ldots, x_n$, this function creates

- A *SOS2* constraint for the variables.
- A constraint that $\sum_{i=0}^{n} x_i = 1$.

•For each variable x_i, a constraint that $x_i \geq 0$.

It is used by *piecewise_affine()*.

Parameters

- **(sequence of** (variables) – class:*Variable* instances)
- **include_lb** (*boolean, optional*) – If `True` (the default), lower bound constraints `x[i] >= 0` are created for the variables. Set to `False` if your variables already have lower bounds `>= 0` and you want to avoid a few redundant constraints.

Returns A set of *SOS2* and *Constraint* instances.

Return type set

2.9.4 Models

Part([name])	A part of a model.
Node([name])	A node with balance constraints.
FlowNetwork(resource[, name])	Manages flows between nodes.
Cluster(*parts[, resource, name])	A node containing other nodes, fully connected.
Storage(resource[, capacity, maxchange, name])	Simple storage model.
ConstraintCollection(owner)	Generates constraints from functions.

friendlysam.parts.Part

class `friendlysam.parts.`**Part**(*name=None*)

A part of a model.

End users probably primarily want to use the subclasses *Node*, *FlowNetwork*, etc.

The `Part` class serves several purposes:

1. `Part` has the attribute *constraints*.
2. A `Part` can contain other parts. Read more about
   - *add_part()*, *remove_part()*
   - *parts()*
   - *find()*
   - *children*, *children_and_self*, *descendants*, *descendants_and_self*
3. `Part` implements Friendly Sam's time model. Read more about
   - *step_time()*
   - *times()* and *iter_times()*
   - *times_between()* and *iter_times_between()*

Parameters **name** (*str, optional*) – The *name* of the part.

Examples

See *Node* for examples.

Part.add_part(part)	Add a part to this part.
Part.find(name)	Try to find a part by name.
Part.iter_times(start, *range_args)	A generator yielding a sequence of times.
Part.iter_times_between(start, end)	A generator yielding all times between two points.
Part.parts([depth, include_self])	Get contained parts, recursively.
Part.remove_part(part)	Remove a part from this part.
Part.state_variables(index)	The state variables of the part.
Part.step_time(index, num_steps)	A function for stepping forward or backward in time.
Part.times(start, *range_args)	Get a sequence of times.
Part.times_between(start, end)	Get a tuple of all times between two points.

friendlysam.parts.Part.add_part

Part.**add_part**(*part*)

Add a part to this part.

Parameters **part** (*Part* or subclass instance) – The part to add.

Raises InsanityError – If the calling part is a descendant of the part to add. (This would generate a cyclic relationship.)

friendlysam.parts.Part.find

Part.**find**(*name*)

Try to find a part by name.

Searches among *descendants_and_self*, comparing the *name*. If there is exactly one match, it is returned.

Parameters **name** – The name to search for.

Returns attr:*descendants_and_self*.

Return type A part named name, if one exists among

Raises ValueError – If there is no match or several matches.

friendlysam.parts.Part.iter_times

Part.**iter_times**(*start*, **range_args*)

A generator yielding a sequence of times.

See also: *times()*, *iter_times_between()*, *times_between()*.

Equivalent to:

```
for num_steps in range(*range_args):
    yield self.step_time(start, num_steps)
```

Parameters

- **start** (*any object*) – The index to start from.
- ***range_args** – Args exactly like for the built-in range().

Examples

```
>>> part = Part()
>>> for t in part.iter_times(3, 5):
...     print(t)
...
3
4
5
6
7
>>> for t in part.iter_times(0, -5, 1, 2):
...     print(t)
...
-5
-3
-1
```

friendlysam.parts.Part.iter_times_between

Part.**iter_times_between**(*start*, *end*)

A generator yielding all times between two points.

See also: *times_between()*, *iter_times()*, *times()*.

Takes one time step at a time from `start` while <= `end`.

Parameters

- **start** (*any object*) – The index to start from.
- **end** (*any object*) – The index to go to.

Note: This function only works if times are orderable, or specifically that the <= operator is implemented.

Examples

```
>>> from pandas import Timestamp, Timedelta
>>> part = Part()
>>> part.time_unit = Timedelta('7 days')
>>> start, end = Timestamp('2011-02-14'), Timestamp('2011-02-28')
>>> between = part.iter_times_between(start, end)
>>> next(between)
Timestamp('2011-02-14 00:00:00')
>>> next(between)
Timestamp('2011-02-21 00:00:00')
>>> next(between)
Timestamp('2011-02-28 00:00:00')
```

friendlysam.parts.Part.parts

Part.**parts**(*depth='inf'*, *include_self=True*)

Get contained parts, recursively.

See also properties *children*, *children_and_self*, *descendants*, *descendants_and_self*.

Parameters

- **depth** (*integer or 'inf', optional*) – The recursion depth to search with. depth=1 searches among the parts directly contained by this part. depth=2 among children and their children, etc.
- **include_self** (*boolean, optional*) – Include this part in the results?

Returns A set of parts.

Examples

```
>>> child = Part(name='Baby')
>>> parent = Part(name='Mommy')
>>> parent.add_part(child)
>>> grandparent = Part('Granny')
>>> grandparent.add_part(parent)
>>> grandparent.children == {parent}
True
>>> grandparent.descendants == {parent, child}
True
>>> grandparent.descendants_and_self == {grandparent, parent, child}
True
>>> grandparent.parts(depth=1, include_self=False) == grandparent.children
True
```

friendlysam.parts.Part.remove_part

`Part.`**`remove_part`**(*part*)

Remove a part from this part.

Parameters **p** (*Part* or subclass instance) – The part to remove.

Raises `KeyError` – If the part is not there.

friendlysam.parts.Part.state_variables

`Part.`**`state_variables`**(*index*)

The state variables of the part.

Each subclass may define *state_variables()*, returning an iterable of the `Varible` instances that define the state of the part at the specified index.

friendlysam.models.MyopicDispatchModel is an example of how it can be used.

Parameters **index** – The index of the state.

Examples

```
>>> from friendlysam import VariableCollection, Domain
>>> class ChocolateFactory(Node):
...     def __init__(self):
...         self.total = VariableCollection('total production')
...         self.mc = VariableCollection('milk chocolate production')
...         self.production['dark chocolate'] = lambda t: self.total(t) - self.mc(t)
```

```
...             self.production['milk chocolate'] = self.mc
...
...         def state_variables(self, t):
...             return (self.total, self.mc)
```

friendlysam.parts.Part.step_time

Part.**step_time**(*index*, *num_steps*)

A function for stepping forward or backward in time.

A *Part* (or subclass) instance may use any logic for stepping in time. To change time stepping, you may have to change time_step or override *step_time()*.

Parameters

- **index** (*any object*) – The index to step from.
- **num_steps** (*int*) – The number of steps to take.

Returns the new time, num_steps away from index.

Examples

If your model is indexed in evenly spaced integers, the default implementation is enough. A step is taken as follows:

```
def step_time(self, index, num_steps):
    return index + self.time_unit * num_steps
```

The default time_unit is 1, so time stepping is done by adding an integer.

```
>>> part = Part()
>>> part.step_time(3, -2)
1
>>> part.time_unit = 10
>>> part.step_time(3, 2)
23
```

Let's assume your model is indexed with pandas.Timestamp in 2-hour increments, then it's still sufficient to change the time unit:

```
>>> from pandas import Timestamp, Timedelta
>>> part = Part()
>>> part.time_unit = Timedelta('2h')
>>> part.step_time(Timestamp('2015-06-10 16:00'), 1)
Timestamp('2015-06-10 18:00:00')
>>> part.step_time(Timestamp('2015-06-10 16:00'), -3)
Timestamp('2015-06-10 10:00:00')
```

If your model is indexed with something more complicated, you may have to change the *step_time()* method. For example, assume a model is indexed with two-element tuples where the first element is an integer representing time. Then override the *step_time()* as follows:

```
>>> def my_step_time(index, step):
...     t, other = index
...     t += step
...     return (t, other)
...
```

```
>>> part = Part()
>>> part.step_time = my_step_time
>>> part.step_time((1, 'ABC'), 2)
(3, 'ABC')
```

friendlysam.parts.Part.times

Part.**times**(*start*, **range_args*)

Get a sequence of times.

See also: *iter_times()*, *iter_times_between()*, *times_between()*.

This works exactly like *iter_times()*, but returns a tuple.

Parameters

- **start** (*any object*) – The index to start from.
- ***range_args** – Args exactly like for the built-in `range()`.

Examples

```
>>> part = Part()
>>> part.times(3, 5)
(3, 4, 5, 6, 7)
>>> part.times(0, -5, 1, 2)
(-5, -3, -1)
```

friendlysam.parts.Part.times_between

Part.**times_between**(*start*, *end*)

Get a tuple of all times between two points.

See also: *times_between()*, *iter_times()*, *times()*.

Takes one time step at a time from `start` while `<= end`. This works exactly like *iter_times_between()*, but returns a tuple.

Parameters

- **start** (*any object*) – The index to start from.
- **end** (*any object*) – The index to go to.

Note: This function only works if times are orderable, or specifically that the <= operator is implemented.

Examples

```
>>> from pandas import Timestamp, Timedelta
>>> part = Part()
>>> part.time_unit = Timedelta('7 days')
>>> start, end = Timestamp('2011-02-14'), Timestamp('2011-02-28')
>>> part.times_between(start, end)
(Timestamp('2011-02-14 00:00:00'), Timestamp('2011-02-21 00:00:00'), Timestamp('2011-02-28 00:00
```

Part.children	Parts in this part, excluding `self`.
Part.children_and_self	Parts in this part, including `self`.
Part.constraints	For defining and generating constraints.
Part.descendants	All `children`, children of children, etc, excluding `self`.
Part.descendants_and_self	All `children`, children of children, etc, including `self`.
Part.name	A name for the object.
Part.time_unit	The time unit used in `step_time()`.

friendlysam.parts.Part.children

`Part.`**`children`**

Parts in this part, excluding `self`.

To add children, use *add_part()*.

friendlysam.parts.Part.children_and_self

`Part.`**`children_and_self`**

Parts in this part, including `self`.

To add children, use *add_part()*.

friendlysam.parts.Part.constraints

`Part.`**`constraints`**

For defining and generating constraints.

This is a *ConstraintCollection* instance. Add functions or iterables of functions to it. Each function should return a constraint or an iterable of constraints. (In this context, a constraint is *Constraint*, *Relation*, *SOS1* or *SOS2*.)

If a constraint function returns a *Relation*, it automatically packaged in a *Constraint* object and marked with *origin* after creation. A constraint function may also return an iterable of constraints, even a generator.

All the added constraint functions are called when *make()* is called.

Examples

There are many ways to formulate constraint functions. Here is a wonderfully contrived example:

```
>>> from friendlysam.opt import VariableCollection, Constraint, Eq
>>> class MyNode(Node):
...     def __init__(self, k):
...         self.k = k
...         self.var = VariableCollection('x')
...         self.production['foo'] = self.var
...         # += and .add() are just alternative syntaxes
...         self.constraints.add(lambda t: self.var(t) >= k[t] - 1)
...         self.constraints += self.constraint_func_1, self.constraint_func_2
...         self.constraints.add(self.constraint_func_3)
...
...     def constraint_func_1(self, t):
...         return Constraint(
```

```
...             self.var(t) <= self.k[t] * 2,
...             desc='Some description')
...
...     def constraint_func_2(self, t):
...         constraints = []
...         t_plus_1 = self.step_time(t, 1)
...         constraints.append(self.var(t) <= self.k[t] * self.var(t_plus_1))
...         constraints.append(self.var(t) >= self.k[t_plus_1] * self.var(t_plus_1))
...         return constraints
...
...     def constraint_func_3(self, t):
...         for prev in self.times_between(0, t):
...             expr = Eq(self.k[prev] * self.var(prev), self.k[t] * self.var(t))
...             desc = 'Why make such a constraint? (k(t)={})'.format(self.k[t])
...             yield Constraint(expr, desc=desc)
...
>>> my_k = {i: i ** 2.4 for i in range(100)}
>>> node = MyNode(my_k)
>>> constraints = node.constraints.make(20)
>>> len(constraints)
26
```

Constraints can also be added from "outside":

```
>>> node.constraints += lambda index: node.production['foo'](index) >= index
>>> constraints = node.constraints.make(20)
>>> len(constraints)
27
```

friendlysam.parts.Part.descendants

Part.**descendants**
All *children*, children of children, etc, excluding `self`.

friendlysam.parts.Part.descendants_and_self

Part.**descendants_and_self**
All *children*, children of children, etc, including `self`.

friendlysam.parts.Part.name

Part.**name**
A name for the object.

The name is for debugging purposes. It has nothing to do with the identity of the object, so does not have to be unique

friendlysam.parts.Part.time_unit

Part.**time_unit**
The time unit used in *step_time()*.

The default value is 1.

For more info, see *step_time()*.

friendlysam.parts.Node

class friendlysam.parts.**Node**(*name=None*)
A node with balance constraints.

Suitable for modeling nodes in a flow network. A *Node* instance produces balance constraints for all its *resources*. The dictionaries *consumption*, *production*, *accumulation*, *inflows*, and *outflows*, are the basis for the balance constraints.

Parameters **name** (*str, optional*) – A name for the node.

Examples

```
>>> class PowerPlant(Node):
...     def __init__(self, efficiency):
...         with namespace(self):
...             x = VariableCollection('output')
...         self.production['power'] = x
...         self.consumption['fuel'] = lambda t: x(t) / efficiency
...
>>> power_plant = PowerPlant(0.85)
>>> constraints = power_plant.constraints.make(42)
>>> constraints
{<friendlysam.opt.Constraint at 0x...>, <friendlysam.opt.Constraint at 0x...>}
>>> {c.desc for c in constraints} == {'Balance constraint (resource=power)',
...                                   'Balance constraint (resource=fuel)'}
True
```

Node.add_part(part)	Add a part to this part.
Node.balance_constraints(index)	Balance constraints for all resources.
Node.cluster(resource)	Get a *Cluster* this node is in.
Node.find(name)	Try to find a part by name.
Node.iter_times(start, *range_args)	A generator yielding a sequence of times.
Node.iter_times_between(start, end)	A generator yielding all times between two points.
Node.parts([depth, include_self])	Get contained parts, recursively.
Node.remove_part(part)	Remove a part from this part.
Node.set_cluster(cluster)	Add this node to a *Cluster*.
Node.state_variables(index)	The state variables of the part.
Node.step_time(index, num_steps)	A function for stepping forward or backward in time.
Node.times(start, *range_args)	Get a sequence of times.
Node.times_between(start, end)	Get a tuple of all times between two points.
Node.unset_cluster(cluster)	Remove from a *Cluster*.

friendlysam.parts.Node.add_part

Node.**add_part**(*part*)
Add a part to this part.

Parameters **part** (*Part* or subclass instance) – The part to add.

Raises InsanityError – If the calling part is a descendant of the part to add. (This would generate a cyclic relationship.)

friendlysam.parts.Node.balance_constraints

Node.**balance_constraints**(*index*)

Balance constraints for all resources.

Returns one constraint for each resource in *resources*, except for the resources for which this node is in a *Cluster*.

Parameters **index** – The index to get the resources for.

Returns The balance constraints.

Return type set

friendlysam.parts.Node.cluster

Node.**cluster**(*resource*)

Get a *Cluster* this node is in.

Parameters **resource** – The *Cluster.resource*.

Returns The Cluster if this node is in a *Cluster* with Cluster.resource`` == resource‘‘, None otherwise.

Return type *cluster*

friendlysam.parts.Node.find

Node.**find**(*name*)

Try to find a part by name.

Searches among *descendants_and_self*, comparing the *name*. If there is exactly one match, it is returned.

Parameters **name** – The name to search for.

Returns attr:*descendants_and_self*.

Return type A part named name, if one exists among

Raises ValueError – If there is no match or several matches.

friendlysam.parts.Node.iter_times

Node.**iter_times**(*start*, **range_args*)

A generator yielding a sequence of times.

See also: *times()*, *iter_times_between()*, *times_between()*.

Equivalent to:

```
for num_steps in range(*range_args):
    yield self.step_time(start, num_steps)
```

Parameters

- **start** (*any object*) – The index to start from.
- ***range_args** – Args exactly like for the built-in range().

Examples

```
>>> part = Part()
>>> for t in part.iter_times(3, 5):
...     print(t)
...
3
4
5
6
7
>>> for t in part.iter_times(0, -5, 1, 2):
...     print(t)
...
-5
-3
-1
```

friendlysam.parts.Node.iter_times_between

Node.**iter_times_between**(*start*, *end*)

A generator yielding all times between two points.

See also: *times_between()*, *iter_times()*, *times()*.

Takes one time step at a time from `start` while <= `end`.

Parameters

- **start** (*any object*) – The index to start from.
- **end** (*any object*) – The index to go to.

Note: This function only works if times are orderable, or specifically that the <= operator is implemented.

Examples

```
>>> from pandas import Timestamp, Timedelta
>>> part = Part()
>>> part.time_unit = Timedelta('7 days')
>>> start, end = Timestamp('2011-02-14'), Timestamp('2011-02-28')
>>> between = part.iter_times_between(start, end)
>>> next(between)
Timestamp('2011-02-14 00:00:00')
>>> next(between)
Timestamp('2011-02-21 00:00:00')
>>> next(between)
Timestamp('2011-02-28 00:00:00')
```

friendlysam.parts.Node.parts

Node.**parts**(*depth='inf'*, *include_self=True*)

Get contained parts, recursively.

See also properties *children*, *children_and_self*, *descendants*, *descendants_and_self*.

Parameters

- **depth** (*integer or 'inf', optional*) – The recursion depth to search with. depth=1 searches among the parts directly contained by this part. depth=2 among children and their children, etc.
- **include_self** (*boolean, optional*) – Include this part in the results?

Returns A set of parts.

Examples

```
>>> child = Part(name='Baby')
>>> parent = Part(name='Mommy')
>>> parent.add_part(child)
>>> grandparent = Part('Granny')
>>> grandparent.add_part(parent)
>>> grandparent.children == {parent}
True
>>> grandparent.descendants == {parent, child}
True
>>> grandparent.descendants_and_self == {grandparent, parent, child}
True
>>> grandparent.parts(depth=1, include_self=False) == grandparent.children
True
```

friendlysam.parts.Node.remove_part

Node.**remove_part**(*part*)

Remove a part from this part.

Parameters **p** (*Part* or subclass instance) – The part to remove.

Raises KeyError – If the part is not there.

friendlysam.parts.Node.set_cluster

Node.**set_cluster**(*cluster*)

Add this node to a *Cluster*.

You should probably use *Cluster.add_part()* instead.

Parameters **cluster** – The **:node:'Cluster'** instance to add to.

friendlysam.parts.Node.state_variables

Node.**state_variables**(*index*)

The state variables of the part.

Each subclass may define *state_variables()*, returning an iterable of the Varible instances that define the state of the part at the specified index.

friendlysam.models.MyopicDispatchModel is an example of how it can be used.

Parameters **index** – The index of the state.

Examples

```
>>> from friendlysam import VariableCollection, Domain
>>> class ChocolateFactory(Node):
...     def __init__(self):
...         self.total = VariableCollection('total production')
...         self.mc = VariableCollection('milk chocolate production')
...         self.production['dark chocolate'] = lambda t: self.total(t) - self.mc(t)
...         self.production['milk chocolate'] = self.mc
...
...     def state_variables(self, t):
...         return (self.total, self.mc)
```

friendlysam.parts.Node.step_time

Node.**step_time**(*index*, *num_steps*)

A function for stepping forward or backward in time.

A *Part* (or subclass) instance may use any logic for stepping in time. To change time stepping, you may have to change `time_step` or override *step_time()*.

Parameters

- **index** (*any object*) – The index to step from.
- **num_steps** (*int*) – The number of steps to take.

Returns the new time, `num_steps` away from index.

Examples

If your model is indexed in evenly spaced integers, the default implementation is enough. A step is taken as follows:

```
def step_time(self, index, num_steps):
    return index + self.time_unit * num_steps
```

The default `time_unit` is `1`, so time stepping is done by adding an integer.

```
>>> part = Part()
>>> part.step_time(3, -2)
1
>>> part.time_unit = 10
>>> part.step_time(3, 2)
23
```

Let's assume your model is indexed with `pandas.Timestamp` in 2-hour increments, then it's still sufficient to change the time unit:

```
>>> from pandas import Timestamp, Timedelta
>>> part = Part()
>>> part.time_unit = Timedelta('2h')
>>> part.step_time(Timestamp('2015-06-10 16:00'), 1)
Timestamp('2015-06-10 18:00:00')
>>> part.step_time(Timestamp('2015-06-10 16:00'), -3)
Timestamp('2015-06-10 10:00:00')
```

If your model is indexed with something more complicated, you may have to change the *step_time()* method. For example, assume a model is indexed with two-element tuples where the first element is an integer representing time. Then override the *step_time()* as follows:

```
>>> def my_step_time(index, step):
...     t, other = index
...     t += step
...     return (t, other)
...
>>> part = Part()
>>> part.step_time = my_step_time
>>> part.step_time((1, 'ABC'), 2)
(3, 'ABC')
```

friendlysam.parts.Node.times

Node.**times**(*start*, **range_args*)

Get a sequence of times.

See also: *iter_times()*, *iter_times_between()*, *times_between()*.

This works exactly like *iter_times()*, but returns a tuple.

Parameters

- **start** (*any object*) – The index to start from.
- ***range_args** – Args exactly like for the built-in `range()`.

Examples

```
>>> part = Part()
>>> part.times(3, 5)
(3, 4, 5, 6, 7)
>>> part.times(0, -5, 1, 2)
(-5, -3, -1)
```

friendlysam.parts.Node.times_between

Node.**times_between**(*start*, *end*)

Get a tuple of all times between two points.

See also: *times_between()*, *iter_times()*, *times()*.

Takes one time step at a time from `start` while `<= end`. This works exactly like *iter_times_between()*, but returns a tuple.

Parameters

- **start** (*any object*) – The index to start from.
- **end** (*any object*) – The index to go to.

Note: This function only works if times are orderable, or specifically that the <= operator is implemented.

Examples

```
>>> from pandas import Timestamp, Timedelta
>>> part = Part()
>>> part.time_unit = Timedelta('7 days')
>>> start, end = Timestamp('2011-02-14'), Timestamp('2011-02-28')
>>> part.times_between(start, end)
(Timestamp('2011-02-14 00:00:00'), Timestamp('2011-02-21 00:00:00'), Timestamp('2011-02-28 00:00
```

friendlysam.parts.Node.unset_cluster

Node.**unset_cluster**(*cluster*)

Remove from a *Cluster*.

You should probably use *Cluster.remove_part()* instead.

Parameters **cluster** – The **:node:‘Cluster‘** instance to remove from.

Node.accumulation	A dictionary of accumulation functions.
Node.children	Parts in this part, excluding self.
Node.children_and_self	Parts in this part, including self.
Node.constraints	For defining and generating constraints.
Node.consumption	A dictionary of consumption functions.
Node.descendants	All children, children of children, etc, excluding self.
Node.descendants_and_self	All children, children of children, etc, including self.
Node.inflows	A dictionary of sets of inflow functions.
Node.name	A name for the object.
Node.outflows	A dictionary of sets of outflow functions.
Node.production	A dictionary of production functions.
Node.resources	The set of resources this node handles.
Node.time_unit	The time unit used in step_time().

friendlysam.parts.Node.accumulation

Node.**accumulation**

A dictionary of accumulation functions.

See *consumption*.

friendlysam.parts.Node.children

Node.**children**

Parts in this part, excluding self.

To add children, use *add_part()*.

friendlysam.parts.Node.children_and_self

Node.**children_and_self**

Parts in this part, including self.

To add children, use *add_part()*.

friendlysam.parts.Node.constraints

Node.**constraints**

For defining and generating constraints.

This is a *ConstraintCollection* instance. Add functions or iterables of functions to it. Each function should return a constraint or an iterable of constraints. (In this context, a constraint is *Constraint*, *Relation*, *SOS1* or *SOS2*.)

If a constraint function returns a *Relation*, it automatically packaged in a *Constraint* object and marked with *origin* after creation. A constraint function may also return an iterable of constraints, even a generator.

All the added constraint functions are called when *make()* is called.

Examples

There are many ways to formulate constraint functions. Here is a wonderfully contrived example:

```
>>> from friendlysam.opt import VariableCollection, Constraint, Eq
>>> class MyNode(Node):
...     def __init__(self, k):
...         self.k = k
...         self.var = VariableCollection('x')
...         self.production['foo'] = self.var
...         # += and .add() are just alternative syntaxes
...         self.constraints.add(lambda t: self.var(t) >= k[t] - 1)
...         self.constraints += self.constraint_func_1, self.constraint_func_2
...         self.constraints.add(self.constraint_func_3)
...
...     def constraint_func_1(self, t):
...         return Constraint(
...             self.var(t) <= self.k[t] * 2,
...             desc='Some description')
...
...     def constraint_func_2(self, t):
...         constraints = []
...         t_plus_1 = self.step_time(t, 1)
...         constraints.append(self.var(t) <= self.k[t] * self.var(t_plus_1))
...         constraints.append(self.var(t) >= self.k[t_plus_1] * self.var(t_plus_1))
...         return constraints
...
...     def constraint_func_3(self, t):
...         for prev in self.times_between(0, t):
...             expr = Eq(self.k[prev] * self.var(prev), self.k[t] * self.var(t))
...             desc = 'Why make such a constraint? (k(t)={})'.format(self.k[t])
...             yield Constraint(expr, desc=desc)
...
>>> my_k = {i: i ** 2.4 for i in range(100)}
>>> node = MyNode(my_k)
>>> constraints = node.constraints.make(20)
>>> len(constraints)
26
```

Constraints can also be added from "outside":

```
>>> node.constraints += lambda index: node.production['foo'](index) >= index
>>> constraints = node.constraints.make(20)
```

```
>>> len(constraints)
27
```

friendlysam.parts.Node.consumption

Node.**consumption**
A dictionary of consumption functions.

Each key in the dictionary is a resource, and the value is a function, taking one argument `index`, returning the consumption at that index.

friendlysam.parts.Node.descendants

Node.**descendants**
All *children*, children of children, etc, excluding `self`.

friendlysam.parts.Node.descendants_and_self

Node.**descendants_and_self**
All *children*, children of children, etc, including `self`.

friendlysam.parts.Node.inflows

Node.**inflows**
A dictionary of sets of inflow functions.

Each key in the dictionary is a resource, and the corresponding value is a set. Each item in each set is a function, taking one argument `index`, returning the an inflow of that resource at that index.

friendlysam.parts.Node.name

Node.**name**
A name for the object.

The name is for debugging purposes. It has nothing to do with the identity of the object, so does not have to be unique

friendlysam.parts.Node.outflows

Node.**outflows**
A dictionary of sets of outflow functions.

Each key in the dictionary is a resource, and the corresponding value is a set. Each item in each set is a function, taking one argument `index`, returning the an outflow of that resource at that index.

friendlysam.parts.Node.production

Node.**production**
A dictionary of production functions.

See *consumption*.

friendlysam.parts.Node.resources

Node.**resources**
The set of resources this node handles.

This is the set of all keys found in the following dictionaries:

- *consumption*
- *production*
- *accumulation*
- *inflows*
- *outflows*

friendlysam.parts.Node.time_unit

Node.**time_unit**
The time unit used in *step_time()*.

The default value is 1.

For more info, see *step_time()*.

friendlysam.parts.FlowNetwork

class friendlysam.parts.**FlowNetwork**(*resource*, *name=None*)
Manages flows between nodes.

FlowNetwork creates flow variables and can connect *Node* instances by changing their *inflows* and *outflows*.

Parameters

- **resource** – The resource flowing in the network.
- **name** (*str, optional*) – The *name* of the network.

Examples

We create three nodes: A producer, a storage, and a consumer.

```
>>> producer = Node(name='producer')
>>> producer.production['R'] = VariableCollection('prod')
>>> consumer = Node(name='consumer')
>>> consumer.consumption['R'] = VariableCollection('cons')
>>> storage = Storage(resource='R', name='storage')
```

Connect the producer to the storage, and the storage to the consumer.

```
>>> network = FlowNetwork(resource='R', name='network')
>>> network.connect(producer, storage)
>>> network.connect(storage, consumer)
>>> for part in [producer, consumer, storage]:
...     for constr in part.constraints.make(5):
...         print(constr.origin.owner)
...         print(constr.expr)
...         print()
...
producer
prod(5) == network.flow(producer-->storage)(5)

consumer
network.flow(storage-->consumer)(5) == cons(5)

storage
network.flow(producer-->storage)(5) == network.flow(storage-->consumer)(5) + storage.volume(6) -
```

FlowNetwork.add_part(part)	Add a part to this part.
FlowNetwork.connect(n1, n2[, bidirectional, ...])	Connect two nodes.
FlowNetwork.find(name)	Try to find a part by name.
FlowNetwork.get_flow(n1, n2)	Get a flow between two nodes.
FlowNetwork.iter_times(start, *range_args)	A generator yielding a sequence of times.
FlowNetwork.iter_times_between(start, end)	A generator yielding all times between two points.
FlowNetwork.parts([depth, include_self])	Get contained parts, recursively.
FlowNetwork.remove_part(part)	
FlowNetwork.state_variables(index)	The state variables are all the flow variables.
FlowNetwork.step_time(index, num_steps)	A function for stepping forward or backward in time.
FlowNetwork.times(start, *range_args)	Get a sequence of times.
FlowNetwork.times_between(start, end)	Get a tuple of all times between two points.

friendlysam.parts.FlowNetwork.add_part

FlowNetwork.**add_part**(*part*)

Add a part to this part.

Parameters **part** (*Part* or subclass instance) – The part to add.

Raises InsanityError – If the calling part is a descendant of the part to add. (This would generate a cyclic relationship.)

friendlysam.parts.FlowNetwork.connect

FlowNetwork.**connect**(*n1*, *n2*, *bidirectional=False*, *capacity=None*)

Connect two nodes.

Creates a flow and adds it to n1.outflows[resource] and n2.inflows[resource], if it does not already exist. Calling again makes no difference.

The flow must be nonnegative. For bidirectional flows, use bidirectional=True.

Parameters

- **n1** – The node the flow goes from.

- **n2** – The node the flow goes to.
- **bidirectional** (*boolean, optional*) – Create a two-way flow?
- **capacity** (*float, optional*) – The maximum amount that can flow between the nodes. Creates an upper bound `ub=capacity` on the flow *Variable* for each index.

friendlysam.parts.FlowNetwork.find

`FlowNetwork.`**find**(*name*)

Try to find a part by name.

Searches among *descendants_and_self*, comparing the *name*. If there is exactly one match, it is returned.

Parameters **name** – The name to search for.

Returns attr:*descendants_and_self*.

Return type A part named `name`, if one exists among

Raises `ValueError` – If there is no match or several matches.

friendlysam.parts.FlowNetwork.get_flow

`FlowNetwork.`**get_flow**(*n1*, *n2*)

Get a flow between two nodes.

Parameters

- **n1** – The node the flow goes from.
- **n2** – The node the flow goes to.

Returns a `VariableCollection`

Raises `KeyError` – If the flow does not exist.

friendlysam.parts.FlowNetwork.iter_times

`FlowNetwork.`**iter_times**(*start*, **range_args*)

A generator yielding a sequence of times.

See also: *times()*, *iter_times_between()*, *times_between()*.

Equivalent to:

```
for num_steps in range(*range_args):
    yield self.step_time(start, num_steps)
```

Parameters

- **start** (*any object*) – The index to start from.
- ***range_args** – Args exactly like for the built-in `range()`.

Examples

```
>>> part = Part()
>>> for t in part.iter_times(3, 5):
...     print(t)
...
3
4
5
6
7
>>> for t in part.iter_times(0, -5, 1, 2):
...     print(t)
...
-5
-3
-1
```

friendlysam.parts.FlowNetwork.iter_times_between

FlowNetwork.**iter_times_between**(*start*, *end*)

A generator yielding all times between two points.

See also: *times_between()*, *iter_times()*, *times()*.

Takes one time step at a time from `start` while <= `end`.

Parameters

- **start** (*any object*) – The index to start from.
- **end** (*any object*) – The index to go to.

Note: This function only works if times are orderable, or specifically that the <= operator is implemented.

Examples

```
>>> from pandas import Timestamp, Timedelta
>>> part = Part()
>>> part.time_unit = Timedelta('7 days')
>>> start, end = Timestamp('2011-02-14'), Timestamp('2011-02-28')
>>> between = part.iter_times_between(start, end)
>>> next(between)
Timestamp('2011-02-14 00:00:00')
>>> next(between)
Timestamp('2011-02-21 00:00:00')
>>> next(between)
Timestamp('2011-02-28 00:00:00')
```

friendlysam.parts.FlowNetwork.parts

FlowNetwork.**parts**(*depth='inf'*, *include_self=True*)

Get contained parts, recursively.

See also properties *children*, *children_and_self*, *descendants*, *descendants_and_self*.

Parameters

- **depth** (*integer or 'inf', optional*) – The recursion depth to search with. depth=1 searches among the parts directly contained by this part. depth=2 among children and their children, etc.
- **include_self** (*boolean, optional*) – Include this part in the results?

Returns A set of parts.

Examples

```
>>> child = Part(name='Baby')
>>> parent = Part(name='Mommy')
>>> parent.add_part(child)
>>> grandparent = Part('Granny')
>>> grandparent.add_part(parent)
>>> grandparent.children == {parent}
True
>>> grandparent.descendants == {parent, child}
True
>>> grandparent.descendants_and_self == {grandparent, parent, child}
True
>>> grandparent.parts(depth=1, include_self=False) == grandparent.children
True
```

friendlysam.parts.FlowNetwork.remove_part

FlowNetwork.**remove_part**(*part*)

friendlysam.parts.FlowNetwork.state_variables

FlowNetwork.**state_variables**(*index*)

The state variables are all the flow variables.

friendlysam.parts.FlowNetwork.step_time

FlowNetwork.**step_time**(*index*, *num_steps*)

A function for stepping forward or backward in time.

A *Part* (or subclass) instance may use any logic for stepping in time. To change time stepping, you may have to change `time_step` or override *step_time()*.

Parameters

- **index** (*any object*) – The index to step from.
- **num_steps** (*int*) – The number of steps to take.

Returns the new time, `num_steps` away from index.

Examples

If your model is indexed in evenly spaced integers, the default implementation is enough. A step is taken as follows:

```
def step_time(self, index, num_steps):
    return index + self.time_unit * num_steps
```

The default `time_unit` is `1`, so time stepping is done by adding an integer.

```
>>> part = Part()
>>> part.step_time(3, -2)
1
>>> part.time_unit = 10
>>> part.step_time(3, 2)
23
```

Let's assume your model is indexed with `pandas.Timestamp` in 2-hour increments, then it's still sufficient to change the time unit:

```
>>> from pandas import Timestamp, Timedelta
>>> part = Part()
>>> part.time_unit = Timedelta('2h')
>>> part.step_time(Timestamp('2015-06-10 16:00'), 1)
Timestamp('2015-06-10 18:00:00')
>>> part.step_time(Timestamp('2015-06-10 16:00'), -3)
Timestamp('2015-06-10 10:00:00')
```

If your model is indexed with something more complicated, you may have to change the *step_time()* method. For example, assume a model is indexed with two-element tuples where the first element is an integer representing time. Then override the *step_time()* as follows:

```
>>> def my_step_time(index, step):
...     t, other = index
...     t += step
...     return (t, other)
...
>>> part = Part()
>>> part.step_time = my_step_time
>>> part.step_time((1, 'ABC'), 2)
(3, 'ABC')
```

friendlysam.parts.FlowNetwork.times

FlowNetwork.**times**(*start*, **range_args*)

Get a sequence of times.

See also: *iter_times()*, *iter_times_between()*, *times_between()*.

This works exactly like *iter_times()*, but returns a tuple.

Parameters

- **start** (*any object*) – The index to start from.
- ***range_args** – Args exactly like for the built-in `range()`.

Examples

```
>>> part = Part()
>>> part.times(3, 5)
(3, 4, 5, 6, 7)
>>> part.times(0, -5, 1, 2)
(-5, -3, -1)
```

friendlysam.parts.FlowNetwork.times_between

FlowNetwork.**times_between**(*start*, *end*)

Get a tuple of all times between two points.

See also: *times_between()*, *iter_times()*, *times()*.

Takes one time step at a time from `start` while `<= end`. This works exactly like *iter_times_between()*, but returns a tuple.

Parameters

- **start** (*any object*) – The index to start from.
- **end** (*any object*) – The index to go to.

Note: This function only works if times are orderable, or specifically that the <= operator is implemented.

Examples

```
>>> from pandas import Timestamp, Timedelta
>>> part = Part()
>>> part.time_unit = Timedelta('7 days')
>>> start, end = Timestamp('2011-02-14'), Timestamp('2011-02-28')
>>> part.times_between(start, end)
(Timestamp('2011-02-14 00:00:00'), Timestamp('2011-02-21 00:00:00'), Timestamp('2011-02-28 00:00
```

FlowNetwork.children	Parts in this part, excluding `self`.
FlowNetwork.children_and_self	Parts in this part, including `self`.
FlowNetwork.constraints	For defining and generating constraints.
FlowNetwork.descendants	All `children`, children of children, etc, excluding `self`.
FlowNetwork.descendants_and_self	All `children`, children of children, etc, including `self`.
FlowNetwork.graph	A graph of all the flows.
FlowNetwork.name	A name for the object.
FlowNetwork.time_unit	The time unit used in `step_time()`.

friendlysam.parts.FlowNetwork.children

FlowNetwork.**children**

Parts in this part, excluding `self`.

To add children, use *add_part()*.

friendlysam.parts.FlowNetwork.children_and_self

FlowNetwork.**children_and_self**
Parts in this part, including `self`.

To add children, use *`add_part()`*.

friendlysam.parts.FlowNetwork.constraints

FlowNetwork.**constraints**
For defining and generating constraints.

This is a *`ConstraintCollection`* instance. Add functions or iterables of functions to it. Each function should return a constraint or an iterable of constraints. (In this context, a constraint is *`Constraint`*, *`Relation`*, *`SOS1`* or *`SOS2`*.)

If a constraint function returns a *`Relation`*, it automatically packaged in a *`Constraint`* object and marked with *`origin`* after creation. A constraint function may also return an iterable of constraints, even a generator.

All the added constraint functions are called when *`make()`* is called.

Examples

There are many ways to formulate constraint functions. Here is a wonderfully contrived example:

```
>>> from friendlysam.opt import VariableCollection, Constraint, Eq
>>> class MyNode(Node):
...     def __init__(self, k):
...         self.k = k
...         self.var = VariableCollection('x')
...         self.production['foo'] = self.var
...         # += and .add() are just alternative syntaxes
...         self.constraints.add(lambda t: self.var(t) >= k[t] - 1)
...         self.constraints += self.constraint_func_1, self.constraint_func_2
...         self.constraints.add(self.constraint_func_3)
...
...     def constraint_func_1(self, t):
...         return Constraint(
...             self.var(t) <= self.k[t] * 2,
...             desc='Some description')
...
...     def constraint_func_2(self, t):
...         constraints = []
...         t_plus_1 = self.step_time(t, 1)
...         constraints.append(self.var(t) <= self.k[t] * self.var(t_plus_1))
...         constraints.append(self.var(t) >= self.k[t_plus_1] * self.var(t_plus_1))
...         return constraints
...
...     def constraint_func_3(self, t):
...         for prev in self.times_between(0, t):
...             expr = Eq(self.k[prev] * self.var(prev), self.k[t] * self.var(t))
...             desc = 'Why make such a constraint? (k(t)={})'.format(self.k[t])
...             yield Constraint(expr, desc=desc)
...
>>> my_k = {i: i ** 2.4 for i in range(100)}
>>> node = MyNode(my_k)
>>> constraints = node.constraints.make(20)
```

```
>>> len(constraints)
26
```

Constraints can also be added from "outside":

```
>>> node.constraints += lambda index: node.production['foo'](index) >= index
>>> constraints = node.constraints.make(20)
>>> len(constraints)
27
```

friendlysam.parts.FlowNetwork.descendants

FlowNetwork.**descendants**
All *children*, children of children, etc, excluding `self`.

friendlysam.parts.FlowNetwork.descendants_and_self

FlowNetwork.**descendants_and_self**
All *children*, children of children, etc, including `self`.

friendlysam.parts.FlowNetwork.graph

FlowNetwork.**graph**
A graph of all the flows.

Gets a NetworkX `DiGraph` representation of the graph of how nodes are connected. See https://networkx.github.io/ for details.

The graph object is a copy of the internal graph, so changing it does not affect the `FlowNetwork`

Examples

```
>>> FlowNetwork('resource').graph
<networkx.classes.digraph.DiGraph object at 0x...>
```

friendlysam.parts.FlowNetwork.name

FlowNetwork.**name**
A name for the object.

The name is for debugging purposes. It has nothing to do with the identity of the object, so does not have to be unique

friendlysam.parts.FlowNetwork.time_unit

FlowNetwork.**time_unit**
The time unit used in *step_time()*.

The default value is 1.

For more info, see *step_time()*.

friendlysam.parts.Cluster

class `friendlysam.parts.`**`Cluster`**(**parts*, *resource=None*, *name=None*)

A node containing other nodes, fully connected.

A cluster is used to create a free flow of a resource `R` among a set of nodes. All *`children`* of a cluster get their *`balance_constraints`* turned off for the resource `R`, and instead the cluster makes an aggregated balance constraint for all the nodes. In this way, a *`Cluster`* is like a *`FlowNetwork`* for resource `R` where all the parts are connected to one another.

Parameters

- **`*parts`** (*optional*) – Zero or more parts to put in the cluster.
- **`resource`** – The resource this cluster handles.
- **`name`** (*optional*) – A name for the cluster.

Examples

Let's create three nodes:

```
>>> producer = Node(name='producer')
>>> producer.production['R'] = VariableCollection('prod')
>>> consumer = Node(name='consumer')
>>> consumer.consumption['R'] = VariableCollection('cons')
>>> storage = Storage(resource='R', name='storage')
>>> nodes = [consumer, producer, storage]
```

Now they all make a balance constraint at any given index:

```
>>> sum(len(n.constraints.make(5)) for n in nodes)
3
```

After clustering, they don't make balance constraints:

```
>>> cluster = Cluster(*nodes, resource='R', name='cluster')
>>> sum(len(n.constraints.make(5)) for n in nodes) # They all make a balance constraint
0
```

But the *`Cluster`* does:

```
>>> for constr in cluster.constraints.make(5):
...     print(constr.expr)
...
prod(5) == cons(5) + storage.volume(6) - storage.volume(5)
```

`Cluster.add_part`(part)	Add a part to this cluster.
`Cluster.balance_constraints`(index)	Balance constraints for all resources.
`Cluster.cluster`(resource)	Get a *`Cluster`* this node is in.
`Cluster.find`(name)	Try to find a part by name.
`Cluster.iter_times`(start, *range_args)	A generator yielding a sequence of times.
`Cluster.iter_times_between`(start, end)	A generator yielding all times between two points.
`Cluster.parts`([depth, include_self])	Get contained parts, recursively.
`Cluster.remove_part`(part)	Remove a part from this part.
`Cluster.set_cluster`(cluster)	Add this node to a *`Cluster`*.
`Cluster.state_variables`(index)	Cluster does not have state variables.

Continued on next page

Table 2.46 – continued from previous page

Cluster.step_time(index, num_steps)	A function for stepping forward or backward in time.
Cluster.times(start, *range_args)	Get a sequence of times.
Cluster.times_between(start, end)	Get a tuple of all times between two points.
Cluster.unset_cluster(cluster)	Remove from a *Cluster*.

friendlysam.parts.Cluster.add_part

Cluster.**add_part**(*part*)

Add a part to this cluster.

Parameters **part** (*Part* or subclass instance) – The part to add.

Raises InsanityError – If the calling part is a descendant of the part to add. (This would generate a cyclic relationship.)

friendlysam.parts.Cluster.balance_constraints

Cluster.**balance_constraints**(*index*)

Balance constraints for all resources.

Returns one constraint for each resource in *resources*, except for the resources for which this node is in a *Cluster*.

Parameters **index** – The index to get the resources for.

Returns The balance constraints.

Return type set

friendlysam.parts.Cluster.cluster

Cluster.**cluster**(*resource*)

Get a *Cluster* this node is in.

Parameters **resource** – The *Cluster.resource*.

Returns The Cluster if this node is in a *Cluster* with Cluster.resource`` == resource‘‘, None otherwise.

Return type *cluster*

friendlysam.parts.Cluster.find

Cluster.**find**(*name*)

Try to find a part by name.

Searches among *descendants_and_self*, comparing the *name*. If there is exactly one match, it is returned.

Parameters **name** – The name to search for.

Returns attr:*descendants_and_self*.

Return type A part named name, if one exists among

Raises ValueError – If there is no match or several matches.

friendlysam.parts.Cluster.iter_times

Cluster.**iter_times**(*start*, **range_args*)

A generator yielding a sequence of times.

See also: *times()*, *iter_times_between()*, *times_between()*.

Equivalent to:

```
for num_steps in range(*range_args):
    yield self.step_time(start, num_steps)
```

Parameters

- **start** (*any object*) – The index to start from.
- ***range_args** – Args exactly like for the built-in `range()`.

Examples

```
>>> part = Part()
>>> for t in part.iter_times(3, 5):
...     print(t)
...
3
4
5
6
7
>>> for t in part.iter_times(0, -5, 1, 2):
...     print(t)
...
-5
-3
-1
```

friendlysam.parts.Cluster.iter_times_between

Cluster.**iter_times_between**(*start*, *end*)

A generator yielding all times between two points.

See also: *times_between()*, *iter_times()*, *times()*.

Takes one time step at a time from `start` while `<= end`.

Parameters

- **start** (*any object*) – The index to start from.
- **end** (*any object*) – The index to go to.

Note: This function only works if times are orderable, or specifically that the <= operator is implemented.

Examples

```
>>> from pandas import Timestamp, Timedelta
>>> part = Part()
>>> part.time_unit = Timedelta('7 days')
>>> start, end = Timestamp('2011-02-14'), Timestamp('2011-02-28')
>>> between = part.iter_times_between(start, end)
>>> next(between)
Timestamp('2011-02-14 00:00:00')
>>> next(between)
Timestamp('2011-02-21 00:00:00')
>>> next(between)
Timestamp('2011-02-28 00:00:00')
```

friendlysam.parts.Cluster.parts

Cluster.**parts**(*depth='inf'*, *include_self=True*)

Get contained parts, recursively.

See also properties *children*, *children_and_self*, *descendants*, *descendants_and_self*.

Parameters

- **depth** (*integer or 'inf', optional*) – The recursion depth to search with. depth=1 searches among the parts directly contained by this part. depth=2 among children and their children, etc.
- **include_self** (*boolean, optional*) – Include this part in the results?

Returns A set of parts.

Examples

```
>>> child = Part(name='Baby')
>>> parent = Part(name='Mommy')
>>> parent.add_part(child)
>>> grandparent = Part('Granny')
>>> grandparent.add_part(parent)
>>> grandparent.children == {parent}
True
>>> grandparent.descendants == {parent, child}
True
>>> grandparent.descendants_and_self == {grandparent, parent, child}
True
>>> grandparent.parts(depth=1, include_self=False) == grandparent.children
True
```

friendlysam.parts.Cluster.remove_part

Cluster.**remove_part**(*part*)

Remove a part from this part.

Parameters **p** (*Part* or subclass instance) – The part to remove.

Raises KeyError – If the part is not there.

friendlysam.parts.Cluster.set_cluster

Cluster.**set_cluster**(*cluster*)
Add this node to a *Cluster*.

You should probably use *Cluster.add_part()* instead.

Parameters **cluster** – The :node:‘Cluster‘ instance to add to.

friendlysam.parts.Cluster.state_variables

Cluster.**state_variables**(*index*)
Cluster does not have state variables.

friendlysam.parts.Cluster.step_time

Cluster.**step_time**(*index*, *num_steps*)
A function for stepping forward or backward in time.

A *Part* (or subclass) instance may use any logic for stepping in time. To change time stepping, you may have to change time_step or override *step_time()*.

Parameters

- **index** (*any object*) – The index to step from.
- **num_steps** (*int*) – The number of steps to take.

Returns the new time, num_steps away from index.

Examples

If your model is indexed in evenly spaced integers, the default implementation is enough. A step is taken as follows:

```
def step_time(self, index, num_steps):
    return index + self.time_unit * num_steps
```

The default time_unit is 1, so time stepping is done by adding an integer.

```
>>> part = Part()
>>> part.step_time(3, -2)
1
>>> part.time_unit = 10
>>> part.step_time(3, 2)
23
```

Let's assume your model is indexed with pandas.Timestamp in 2-hour increments, then it's still sufficient to change the time unit:

```
>>> from pandas import Timestamp, Timedelta
>>> part = Part()
>>> part.time_unit = Timedelta('2h')
>>> part.step_time(Timestamp('2015-06-10 16:00'), 1)
Timestamp('2015-06-10 18:00:00')
>>> part.step_time(Timestamp('2015-06-10 16:00'), -3)
Timestamp('2015-06-10 10:00:00')
```

If your model is indexed with something more complicated, you may have to change the *step_time()* method. For example, assume a model is indexed with two-element tuples where the first element is an integer representing time. Then override the *step_time()* as follows:

```
>>> def my_step_time(index, step):
...     t, other = index
...     t += step
...     return (t, other)
...
>>> part = Part()
>>> part.step_time = my_step_time
>>> part.step_time((1, 'ABC'), 2)
(3, 'ABC')
```

friendlysam.parts.Cluster.times

Cluster.**times**(*start*, **range_args*)

Get a sequence of times.

See also: *iter_times()*, *iter_times_between()*, *times_between()*.

This works exactly like *iter_times()*, but returns a tuple.

Parameters

- **start** (*any object*) – The index to start from.
- ***range_args** – Args exactly like for the built-in `range()`.

Examples

```
>>> part = Part()
>>> part.times(3, 5)
(3, 4, 5, 6, 7)
>>> part.times(0, -5, 1, 2)
(-5, -3, -1)
```

friendlysam.parts.Cluster.times_between

Cluster.**times_between**(*start*, *end*)

Get a tuple of all times between two points.

See also: *times_between()*, *iter_times()*, *times()*.

Takes one time step at a time from `start` while `<= end`. This works exactly like *iter_times_between()*, but returns a tuple.

Parameters

- **start** (*any object*) – The index to start from.
- **end** (*any object*) – The index to go to.

Note: This function only works if times are orderable, or specifically that the <= operator is implemented.

Examples

```
>>> from pandas import Timestamp, Timedelta
>>> part = Part()
>>> part.time_unit = Timedelta('7 days')
>>> start, end = Timestamp('2011-02-14'), Timestamp('2011-02-28')
>>> part.times_between(start, end)
(Timestamp('2011-02-14 00:00:00'), Timestamp('2011-02-21 00:00:00'), Timestamp('2011-02-28 00:00
```

friendlysam.parts.Cluster.unset_cluster

Cluster.**unset_cluster**(*cluster*)

Remove from a *Cluster*.

You should probably use *Cluster.remove_part()* instead.

Parameters **cluster** – The **:node:‘Cluster‘** instance to remove from.

Cluster.accumulation	A dictionary of accumulation functions.
Cluster.children	Parts in this part, excluding `self`.
Cluster.children_and_self	Parts in this part, including `self`.
Cluster.constraints	For defining and generating constraints.
Cluster.consumption	A dictionary of consumption functions.
Cluster.descendants	All `children`, children of children, etc, excluding `self`.
Cluster.descendants_and_self	All `children`, children of children, etc, including `self`.
Cluster.inflows	A dictionary of sets of inflow functions.
Cluster.name	A name for the object.
Cluster.outflows	A dictionary of sets of outflow functions.
Cluster.production	A dictionary of production functions.
Cluster.resource	The resource this cluster collects.
Cluster.resources	The set of resources this node handles.
Cluster.time_unit	The time unit used in `step_time()`.

friendlysam.parts.Cluster.accumulation

Cluster.**accumulation**

A dictionary of accumulation functions.

See *consumption*.

friendlysam.parts.Cluster.children

Cluster.**children**

Parts in this part, excluding `self`.

To add children, use *add_part()*.

friendlysam.parts.Cluster.children_and_self

Cluster.**children_and_self**

Parts in this part, including `self`.

To add children, use *add_part()*.

friendlysam.parts.Cluster.constraints

Cluster.**constraints**

For defining and generating constraints.

This is a *ConstraintCollection* instance. Add functions or iterables of functions to it. Each function should return a constraint or an iterable of constraints. (In this context, a constraint is *Constraint*, *Relation*, *SOS1* or *SOS2*.)

If a constraint function returns a *Relation*, it automatically packaged in a *Constraint* object and marked with *origin* after creation. A constraint function may also return an iterable of constraints, even a generator.

All the added constraint functions are called when *make()* is called.

Examples

There are many ways to formulate constraint functions. Here is a wonderfully contrived example:

```
>>> from friendlysam.opt import VariableCollection, Constraint, Eq
>>> class MyNode(Node):
...     def __init__(self, k):
...         self.k = k
...         self.var = VariableCollection('x')
...         self.production['foo'] = self.var
...         # += and .add() are just alternative syntaxes
...         self.constraints.add(lambda t: self.var(t) >= k[t] - 1)
...         self.constraints += self.constraint_func_1, self.constraint_func_2
...         self.constraints.add(self.constraint_func_3)
...
...     def constraint_func_1(self, t):
...         return Constraint(
...             self.var(t) <= self.k[t] * 2,
...             desc='Some description')
...
...     def constraint_func_2(self, t):
...         constraints = []
...         t_plus_1 = self.step_time(t, 1)
...         constraints.append(self.var(t) <= self.k[t] * self.var(t_plus_1))
...         constraints.append(self.var(t) >= self.k[t_plus_1] * self.var(t_plus_1))
...         return constraints
...
...     def constraint_func_3(self, t):
...         for prev in self.times_between(0, t):
...             expr = Eq(self.k[prev] * self.var(prev), self.k[t] * self.var(t))
...             desc = 'Why make such a constraint? (k(t)={})'.format(self.k[t])
...             yield Constraint(expr, desc=desc)
...
>>> my_k = {i: i ** 2.4 for i in range(100)}
>>> node = MyNode(my_k)
>>> constraints = node.constraints.make(20)
>>> len(constraints)
26
```

Constraints can also be added from "outside":

```
    >>> node.constraints += lambda index: node.production['foo'](index) >= index
    >>> constraints = node.constraints.make(20)
    >>> len(constraints)
    27
```

friendlysam.parts.Cluster.consumption

Cluster.**consumption**
A dictionary of consumption functions.

Each key in the dictionary is a resource, and the value is a function, taking one argument `index`, returning the consumption at that index.

friendlysam.parts.Cluster.descendants

Cluster.**descendants**
All *children*, children of children, etc, excluding `self`.

friendlysam.parts.Cluster.descendants_and_self

Cluster.**descendants_and_self**
All *children*, children of children, etc, including `self`.

friendlysam.parts.Cluster.inflows

Cluster.**inflows**
A dictionary of sets of inflow functions.

Each key in the dictionary is a resource, and the corresponding value is a set. Each item in each set is a function, taking one argument `index`, returning the an inflow of that resource at that index.

friendlysam.parts.Cluster.name

Cluster.**name**
A name for the object.

The name is for debugging purposes. It has nothing to do with the identity of the object, so does not have to be unique

friendlysam.parts.Cluster.outflows

Cluster.**outflows**
A dictionary of sets of outflow functions.

Each key in the dictionary is a resource, and the corresponding value is a set. Each item in each set is a function, taking one argument `index`, returning the an outflow of that resource at that index.

friendlysam.parts.Cluster.production

`Cluster.`**`production`**
A dictionary of production functions.

See *consumption*.

friendlysam.parts.Cluster.resource

`Cluster.`**`resource`**
The resource this cluster collects. Read only.

friendlysam.parts.Cluster.resources

`Cluster.`**`resources`**
The set of resources this node handles.

This is the set of all keys found in the following dictionaries:

- *consumption*
- *production*
- *accumulation*
- *inflows*
- *outflows*

friendlysam.parts.Cluster.time_unit

`Cluster.`**`time_unit`**
The time unit used in *step_time()*.

The default value is 1.

For more info, see *step_time()*.

friendlysam.parts.Storage

class `friendlysam.parts.`**`Storage`**(*resource*, *capacity=None*, *maxchange=None*, *name=None*)
Simple storage model.

The storage has a volume function. It should be thought of as the volume at the beginning of a time step, such that `volume(t) + accumulation[resource](t) == volume(t+1)`, or more exactly,

```
>>> s = Storage('my_resource')
>>> t = 42
>>> t_plus_1 = s.step_time(t, 1)
>>> s.accumulation['my_resource'](t) == s.volume(t_plus_1) - s.volume(t)
True
```

Parameters

- **`resource`** – The resource to store.

- **capacity** (*float, optional*) – The maximum amount that can be stored. If `None` (the default), there is no limit.
- **maxchange** (*float, optional*) – The `maxchange` of the storage.
- **name** (*str, optional*) – The *name* of the node.

Examples

```
>>> from pandas import Timestamp, Timedelta
>>> battery = Storage('power', name='Battery')
>>> battery.time_unit = Timedelta('3h')
>>> t = Timestamp('2015-06-10 18:00')
>>> print(battery.accumulation['power'](t))
Battery.volume(2015-06-10 21:00:00) - Battery.volume(2015-06-10 18:00:00)
```

Storage.add_part(part)	Add a part to this part.
Storage.balance_constraints(index)	Balance constraints for all resources.
Storage.cluster(resource)	Get a *Cluster* this node is in.
Storage.find(name)	Try to find a part by name.
Storage.iter_times(start, *range_args)	A generator yielding a sequence of times.
Storage.iter_times_between(start, end)	A generator yielding all times between two points.
Storage.parts([depth, include_self])	Get contained parts, recursively.
Storage.remove_part(part)	Remove a part from this part.
Storage.set_cluster(cluster)	Add this node to a *Cluster*.
Storage.state_variables(index)	The only state variable is `volume (index)`.
Storage.step_time(index, num_steps)	A function for stepping forward or backward in time.
Storage.times(start, *range_args)	Get a sequence of times.
Storage.times_between(start, end)	Get a tuple of all times between two points.
Storage.unset_cluster(cluster)	Remove from a *Cluster*.

friendlysam.parts.Storage.add_part

`Storage.`**`add_part`**(*part*)

Add a part to this part.

Parameters **part** (*Part* or subclass instance) – The part to add.

Raises `InsanityError` – If the calling part is a descendant of the part to add. (This would generate a cyclic relationship.)

friendlysam.parts.Storage.balance_constraints

`Storage.`**`balance_constraints`**(*index*)

Balance constraints for all resources.

Returns one constraint for each resource in *resources*, except for the resources for which this node is in a *Cluster*.

Parameters **index** – The index to get the resources for.

Returns The balance constraints.

Return type set

friendlysam.parts.Storage.cluster

Storage.**cluster**(*resource*)

Get a *Cluster* this node is in.

Parameters **resource** – The *Cluster.resource*.

Returns The Cluster if this node is in a *Cluster* with Cluster.resource`` == resource‘‘, None otherwise.

Return type *cluster*

friendlysam.parts.Storage.find

Storage.**find**(*name*)

Try to find a part by name.

Searches among *descendants_and_self*, comparing the *name*. If there is exactly one match, it is returned.

Parameters **name** – The name to search for.

Returns attr:*descendants_and_self*.

Return type A part named name, if one exists among

Raises ValueError – If there is no match or several matches.

friendlysam.parts.Storage.iter_times

Storage.**iter_times**(*start*, **range_args*)

A generator yielding a sequence of times.

See also: *times()*, *iter_times_between()*, *times_between()*.

Equivalent to:

```
for num_steps in range(*range_args):
    yield self.step_time(start, num_steps)
```

Parameters

- **start** (*any object*) – The index to start from.
- ***range_args** – Args exactly like for the built-in range().

Examples

```
>>> part = Part()
>>> for t in part.iter_times(3, 5):
...     print(t)
...
3
4
5
6
7
```

```
>>> for t in part.iter_times(0, -5, 1, 2):
...     print(t)
...
-5
-3
-1
```

friendlysam.parts.Storage.iter_times_between

Storage.**iter_times_between**(*start*, *end*)

A generator yielding all times between two points.

See also: *times_between()*, *iter_times()*, *times()*.

Takes one time step at a time from `start` while `<= end`.

Parameters

- **start** (*any object*) – The index to start from.
- **end** (*any object*) – The index to go to.

Note: This function only works if times are orderable, or specifically that the <= operator is implemented.

Examples

```
>>> from pandas import Timestamp, Timedelta
>>> part = Part()
>>> part.time_unit = Timedelta('7 days')
>>> start, end = Timestamp('2011-02-14'), Timestamp('2011-02-28')
>>> between = part.iter_times_between(start, end)
>>> next(between)
Timestamp('2011-02-14 00:00:00')
>>> next(between)
Timestamp('2011-02-21 00:00:00')
>>> next(between)
Timestamp('2011-02-28 00:00:00')
```

friendlysam.parts.Storage.parts

Storage.**parts**(*depth='inf'*, *include_self=True*)

Get contained parts, recursively.

See also properties *children*, *children_and_self*, *descendants*, *descendants_and_self*.

Parameters

- **depth** (*integer or 'inf', optional*) – The recursion depth to search with. depth=1 searches among the parts directly contained by this part. depth=2 among children and their children, etc.
- **include_self** (*boolean, optional*) – Include this part in the results?

Returns A set of parts.

Examples

```
>>> child = Part(name='Baby')
>>> parent = Part(name='Mommy')
>>> parent.add_part(child)
>>> grandparent = Part('Granny')
>>> grandparent.add_part(parent)
>>> grandparent.children == {parent}
True
>>> grandparent.descendants == {parent, child}
True
>>> grandparent.descendants_and_self == {grandparent, parent, child}
True
>>> grandparent.parts(depth=1, include_self=False) == grandparent.children
True
```

friendlysam.parts.Storage.remove_part

Storage.**remove_part**(*part*)

Remove a part from this part.

Parameters **p** (*Part* or subclass instance) – The part to remove.

Raises KeyError – If the part is not there.

friendlysam.parts.Storage.set_cluster

Storage.**set_cluster**(*cluster*)

Add this node to a *Cluster*.

You should probably use *Cluster.add_part()* instead.

Parameters **cluster** – The **:node:‘Cluster‘** instance to add to.

friendlysam.parts.Storage.state_variables

Storage.**state_variables**(*index*)

The only state variable is `volume (index)`.

friendlysam.parts.Storage.step_time

Storage.**step_time**(*index*, *num_steps*)

A function for stepping forward or backward in time.

A *Part* (or subclass) instance may use any logic for stepping in time. To change time stepping, you may have to change `time_step` or override *step_time()*.

Parameters

- **index** (*any object*) – The index to step from.
- **num_steps** (*int*) – The number of steps to take.

Returns the new time, `num_steps` away from index.

Examples

If your model is indexed in evenly spaced integers, the default implementation is enough. A step is taken as follows:

```
def step_time(self, index, num_steps):
    return index + self.time_unit * num_steps
```

The default `time_unit` is `1`, so time stepping is done by adding an integer.

```
>>> part = Part()
>>> part.step_time(3, -2)
1
>>> part.time_unit = 10
>>> part.step_time(3, 2)
23
```

Let's assume your model is indexed with `pandas.Timestamp` in 2-hour increments, then it's still sufficient to change the time unit:

```
>>> from pandas import Timestamp, Timedelta
>>> part = Part()
>>> part.time_unit = Timedelta('2h')
>>> part.step_time(Timestamp('2015-06-10 16:00'), 1)
Timestamp('2015-06-10 18:00:00')
>>> part.step_time(Timestamp('2015-06-10 16:00'), -3)
Timestamp('2015-06-10 10:00:00')
```

If your model is indexed with something more complicated, you may have to change the *step_time()* method. For example, assume a model is indexed with two-element tuples where the first element is an integer representing time. Then override the *step_time()* as follows:

```
>>> def my_step_time(index, step):
...     t, other = index
...     t += step
...     return (t, other)
...
>>> part = Part()
>>> part.step_time = my_step_time
>>> part.step_time((1, 'ABC'), 2)
(3, 'ABC')
```

friendlysam.parts.Storage.times

Storage.**times**(*start*, **range_args*)

Get a sequence of times.

See also: *iter_times()*, *iter_times_between()*, *times_between()*.

This works exactly like *iter_times()*, but returns a tuple.

Parameters

- **start** (*any object*) – The index to start from.
- ***range_args** – Args exactly like for the built-in `range()`.

Examples

```
>>> part = Part()
>>> part.times(3, 5)
(3, 4, 5, 6, 7)
>>> part.times(0, -5, 1, 2)
(-5, -3, -1)
```

friendlysam.parts.Storage.times_between

Storage.**times_between**(*start*, *end*)

Get a tuple of all times between two points.

See also: *times_between()*, *iter_times()*, *times()*.

Takes one time step at a time from `start` while `<= end`. This works exactly like *iter_times_between()*, but returns a tuple.

Parameters

- **start** (*any object*) – The index to start from.
- **end** (*any object*) – The index to go to.

Note: This function only works if times are orderable, or specifically that the <= operator is implemented.

Examples

```
>>> from pandas import Timestamp, Timedelta
>>> part = Part()
>>> part.time_unit = Timedelta('7 days')
>>> start, end = Timestamp('2011-02-14'), Timestamp('2011-02-28')
>>> part.times_between(start, end)
(Timestamp('2011-02-14 00:00:00'), Timestamp('2011-02-21 00:00:00'), Timestamp('2011-02-28 00:00
```

friendlysam.parts.Storage.unset_cluster

Storage.**unset_cluster**(*cluster*)

Remove from a *Cluster*.

You should probably use *Cluster.remove_part()* instead.

Parameters **cluster** – The :node:‘Cluster‘ instance to remove from.

Storage.accumulation	A dictionary of accumulation functions.
Storage.children	Parts in this part, excluding `self`.
Storage.children_and_self	Parts in this part, including `self`.
Storage.constraints	For defining and generating constraints.
Storage.consumption	A dictionary of consumption functions.
Storage.descendants	All `children`, children of children, etc, excluding `self`.
Storage.descendants_and_self	All `children`, children of children, etc, including `self`.
Storage.inflows	A dictionary of sets of inflow functions.

Continued on next page

Table 2.49 – continued from previous page

Storage.name	A name for the object.
Storage.outflows	A dictionary of sets of outflow functions.
Storage.production	A dictionary of production functions.
Storage.resource	The resource this storage stores.
Storage.resources	The set of resources this node handles.
Storage.time_unit	The time unit used in `step_time()`.

friendlysam.parts.Storage.accumulation

Storage.**accumulation**
A dictionary of accumulation functions.

See *consumption*.

friendlysam.parts.Storage.children

Storage.**children**
Parts in this part, excluding `self`.

To add children, use *add_part()*.

friendlysam.parts.Storage.children_and_self

Storage.**children_and_self**
Parts in this part, including `self`.

To add children, use *add_part()*.

friendlysam.parts.Storage.constraints

Storage.**constraints**
For defining and generating constraints.

This is a *ConstraintCollection* instance. Add functions or iterables of functions to it. Each function should return a constraint or an iterable of constraints. (In this context, a constraint is *Constraint*, *Relation*, *SOS1* or *SOS2*.)

If a constraint function returns a *Relation*, it automatically packaged in a *Constraint* object and marked with *origin* after creation. A constraint function may also return an iterable of constraints, even a generator.

All the added constraint functions are called when *make()* is called.

Examples

There are many ways to formulate constraint functions. Here is a wonderfully contrived example:

```
>>> from friendlysam.opt import VariableCollection, Constraint, Eq
>>> class MyNode(Node):
...     def __init__(self, k):
...         self.k = k
...         self.var = VariableCollection('x')
...         self.production['foo'] = self.var
```

```
...         # += and .add() are just alternative syntaxes
...         self.constraints.add(lambda t: self.var(t) >= k[t] - 1)
...         self.constraints += self.constraint_func_1, self.constraint_func_2
...         self.constraints.add(self.constraint_func_3)
...
...     def constraint_func_1(self, t):
...         return Constraint(
...             self.var(t) <= self.k[t] * 2,
...             desc='Some description')
...
...     def constraint_func_2(self, t):
...         constraints = []
...         t_plus_1 = self.step_time(t, 1)
...         constraints.append(self.var(t) <= self.k[t] * self.var(t_plus_1))
...         constraints.append(self.var(t) >= self.k[t_plus_1] * self.var(t_plus_1))
...         return constraints
...
...     def constraint_func_3(self, t):
...         for prev in self.times_between(0, t):
...             expr = Eq(self.k[prev] * self.var(prev), self.k[t] * self.var(t))
...             desc = 'Why make such a constraint? (k(t)={})'.format(self.k[t])
...             yield Constraint(expr, desc=desc)
...
>>> my_k = {i: i ** 2.4 for i in range(100)}
>>> node = MyNode(my_k)
>>> constraints = node.constraints.make(20)
>>> len(constraints)
26
```

Constraints can also be added from "outside":

```
>>> node.constraints += lambda index: node.production['foo'](index) >= index
>>> constraints = node.constraints.make(20)
>>> len(constraints)
27
```

friendlysam.parts.Storage.consumption

Storage.**consumption**

A dictionary of consumption functions.

Each key in the dictionary is a resource, and the value is a function, taking one argument `index`, returning the consumption at that index.

friendlysam.parts.Storage.descendants

Storage.**descendants**

All *children*, children of children, etc, excluding `self`.

friendlysam.parts.Storage.descendants_and_self

Storage.**descendants_and_self**

All *children*, children of children, etc, including `self`.

friendlysam.parts.Storage.inflows

Storage.**inflows**
A dictionary of sets of inflow functions.

Each key in the dictionary is a resource, and the corresponding value is a set. Each item in each set is a function, taking one argument `index`, returning the an inflow of that resource at that index.

friendlysam.parts.Storage.name

Storage.**name**
A name for the object.

The name is for debugging purposes. It has nothing to do with the identity of the object, so does not have to be unique

friendlysam.parts.Storage.outflows

Storage.**outflows**
A dictionary of sets of outflow functions.

Each key in the dictionary is a resource, and the corresponding value is a set. Each item in each set is a function, taking one argument `index`, returning the an outflow of that resource at that index.

friendlysam.parts.Storage.production

Storage.**production**
A dictionary of production functions.

See *consumption*.

friendlysam.parts.Storage.resource

Storage.**resource**
The resource this storage stores. Read only.

friendlysam.parts.Storage.resources

Storage.**resources**
The set of resources this node handles.

This is the set of all keys found in the following dictionaries:

- *consumption*
- *production*
- *accumulation*
- *inflows*
- *outflows*

friendlysam.parts.Storage.time_unit

Storage.**time_unit**
The time unit used in *step_time()*.

The default value is 1.

For more info, see *step_time()*.

friendlysam.parts.ConstraintCollection

class friendlysam.parts.**ConstraintCollection**(*owner*)
Generates constraints from functions.

This class aggregates functions that generate constraints. It is primarily meant to be used as an attribute of the *Part* class.

Add functions to it, functions that return constraints. Then call the *make()* with an index and receive the set of all constraints generated by the contained functions with that index.

See docs for *Part.constraints* for details.

ConstraintCollection.add(addition)	Add a constraint function, or an iterable of constraint functions.
ConstraintCollection.make(index)	Create constraints from contained functions.

friendlysam.parts.ConstraintCollection.add

ConstraintCollection.**add**(*addition*)
Add a constraint function, or an iterable of constraint functions.

Parameters **addition** (*callable or iterable of callables*) – Constraint function(s) to add.

Examples

c = ConstraintCollection(owner) c.add(constraint_func) c.add([func1, func2])

c += constraint_func c += [func1, func2]

friendlysam.parts.ConstraintCollection.make

ConstraintCollection.**make**(*index*)
Create constraints from contained functions.

Parameters **index** – The index to call the constraint functions with.

Examples

c = ConstraintCollection(owner) c.add(func1) c += func2 # Alternative syntax. c(index) # Returns a set with all the constraints from func1 and func2

MyopicDispatchModel([t0, horizon, step, ...])	docstring for MyopicDispatchModel

friendlysam.models.MyopicDispatchModel

class friendlysam.models.**MyopicDispatchModel**(*t0=None*, *horizon=None*, *step=None*, *name=None*, *require_cost=True*)

docstring for MyopicDispatchModel

MyopicDispatchModel.advance()
MyopicDispatchModel.cost(t)
MyopicDispatchModel.state_variables(t)

friendlysam.models.MyopicDispatchModel.advance

MyopicDispatchModel.**advance**()

friendlysam.models.MyopicDispatchModel.cost

MyopicDispatchModel.**cost**(*t*)

friendlysam.models.MyopicDispatchModel.state_variables

MyopicDispatchModel.**state_variables**(*t*)

MyopicDispatchModel.children	Parts in this part, excluding `self`.
MyopicDispatchModel.children_and_self	Parts in this part, including `self`.
MyopicDispatchModel.constraints	For defining and generating constraints.
MyopicDispatchModel.descendants	All `children`, children of children, etc, excluding `self`.
MyopicDispatchModel.descendants_and_self	All `children`, children of children, etc, including `self`.
MyopicDispatchModel.name	A name for the object.
MyopicDispatchModel.time_unit	The time unit used in `step_time()`.

friendlysam.models.MyopicDispatchModel.children

MyopicDispatchModel.**children**

Parts in this part, excluding `self`.

To add children, use `add_part()`.

friendlysam.models.MyopicDispatchModel.children_and_self

MyopicDispatchModel.**children_and_self**

Parts in this part, including `self`.

To add children, use `add_part()`.

friendlysam.models.MyopicDispatchModel.constraints

MyopicDispatchModel.**constraints**

For defining and generating constraints.

This is a `ConstraintCollection` instance. Add functions or iterables of functions to it. Each function should return a constraint or an iterable of constraints. (In this context, a constraint is *Constraint*, *Relation*, *SOS1* or *SOS2*.)

If a constraint function returns a *Relation*, it automatically packaged in a *Constraint* object and marked with *origin* after creation. A constraint function may also return an iterable of constraints, even a generator.

All the added constraint functions are called when `make()` is called.

Examples

There are many ways to formulate constraint functions. Here is a wonderfully contrived example:

```
>>> from friendlysam.opt import VariableCollection, Constraint, Eq
>>> class MyNode(Node):
...     def __init__(self, k):
...         self.k = k
...         self.var = VariableCollection('x')
...         self.production['foo'] = self.var
...         # += and .add() are just alternative syntaxes
...         self.constraints.add(lambda t: self.var(t) >= k[t] - 1)
...         self.constraints += self.constraint_func_1, self.constraint_func_2
...         self.constraints.add(self.constraint_func_3)
...
...     def constraint_func_1(self, t):
...         return Constraint(
...             self.var(t) <= self.k[t] * 2,
...             desc='Some description')
...
...     def constraint_func_2(self, t):
...         constraints = []
...         t_plus_1 = self.step_time(t, 1)
...         constraints.append(self.var(t) <= self.k[t] * self.var(t_plus_1))
...         constraints.append(self.var(t) >= self.k[t_plus_1] * self.var(t_plus_1))
...         return constraints
...
...     def constraint_func_3(self, t):
...         for prev in self.times_between(0, t):
...             expr = Eq(self.k[prev] * self.var(prev), self.k[t] * self.var(t))
...             desc = 'Why make such a constraint? (k(t)={})'.format(self.k[t])
...             yield Constraint(expr, desc=desc)
...
>>> my_k = {i: i ** 2.4 for i in range(100)}
>>> node = MyNode(my_k)
>>> constraints = node.constraints.make(20)
>>> len(constraints)
26
```

Constraints can also be added from "outside":

```
>>> node.constraints += lambda index: node.production['foo'](index) >= index
>>> constraints = node.constraints.make(20)
>>> len(constraints)
27
```

friendlysam.models.MyopicDispatchModel.descendants

MyopicDispatchModel.**descendants**
All *children*, children of children, etc, excluding `self`.

friendlysam.models.MyopicDispatchModel.descendants_and_self

MyopicDispatchModel.**descendants_and_self**
All *children*, children of children, etc, including `self`.

friendlysam.models.MyopicDispatchModel.name

MyopicDispatchModel.**name**
A name for the object.

The name is for debugging purposes. It has nothing to do with the identity of the object, so does not have to be unique

friendlysam.models.MyopicDispatchModel.time_unit

MyopicDispatchModel.**time_unit**
The time unit used in `step_time()`.

The default value is 1.

For more info, see `step_time()`.

2.9.5 Utilities

get_list(func, indices)	Get a list of function values at indices.
get_series(func, indices, **kwargs)	Get a pandas Series of function values at indices.

friendlysam.util.get_list

friendlysam.util.**get_list**(*func*, *indices*)
Get a list of function values at indices.

Parameters

- **func** (*callable*) – The callable to get values from.
- **indices** (*iterable*) – An iterable of index values to pass to `func`.

Returns values as `float`

Return type list

Examples

```
>>> from friendlysam import Storage
>>> s = Storage('power', name='Battery')
>>> for i in range(5):
```

```
...     s.volume(i).value = i ** 2
...
>>> get_list(s.accumulation['power'], range(4))
[1.0, 3.0, 5.0, 7.0]
```

friendlysam.util.get_series

friendlysam.util.**get_series**(*func*, *indices*, ***kwargs*)

Get a pandas Series of function values at indices.

Equivalent to `pandas.Series(index=indices, data=get_list(func, indices), **kwargs)`.

Parameters

- **func** (*callable*) – The callable to get values from.
- **indices** (*iterable*) – An iterable of index values to pass to `func`.

Returns Values at indices.

Return type pandas.Series

Examples

```
>>> from friendlysam import Storage
>>> s = Storage('power', name='Battery')
>>> for i in range(5):
...     s.volume(i).value = i ** 2
...
>>> get_series(s.accumulation['power'], range(4))
0    1
1    3
2    5
3    7
dtype: float64
```

2.9.6 Exceptions

ConstraintError(*args[, constraint])	Raised when there is something wrong with a *Constraint*.
NoValueError	Raised when a variable or expression has no value.
SolverError	A generic exception raised by a solver instance.

friendlysam.opt.ConstraintError

exception friendlysam.opt.**ConstraintError**(**args*, *constraint=None*, ***kwargs*)

Raised when there is something wrong with a *Constraint*.

Parameters

- **constraint** (*optional*) – The constraint that caused the problem.
- ***args** – Passed on to parent exception constructor.
- ****kwargs** – Passed on to parent exception constructor.

friendlysam.opt.NoValueError

exception `friendlysam.opt.`**`NoValueError`**
Raised when a variable or expression has no value.

friendlysam.opt.SolverError

exception `friendlysam.opt.`**`SolverError`**
A generic exception raised by a solver instance.

InsanityError	Raised when a sanity check fails.

friendlysam.InsanityError

exception `friendlysam.`**`InsanityError`**
Raised when a sanity check fails.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Symbols

A

B

C