modelx Documentation

Release 0.0.25

Fumito Hamamura

Dec 01, 2019

Contents

Escape from spreadsheet models!

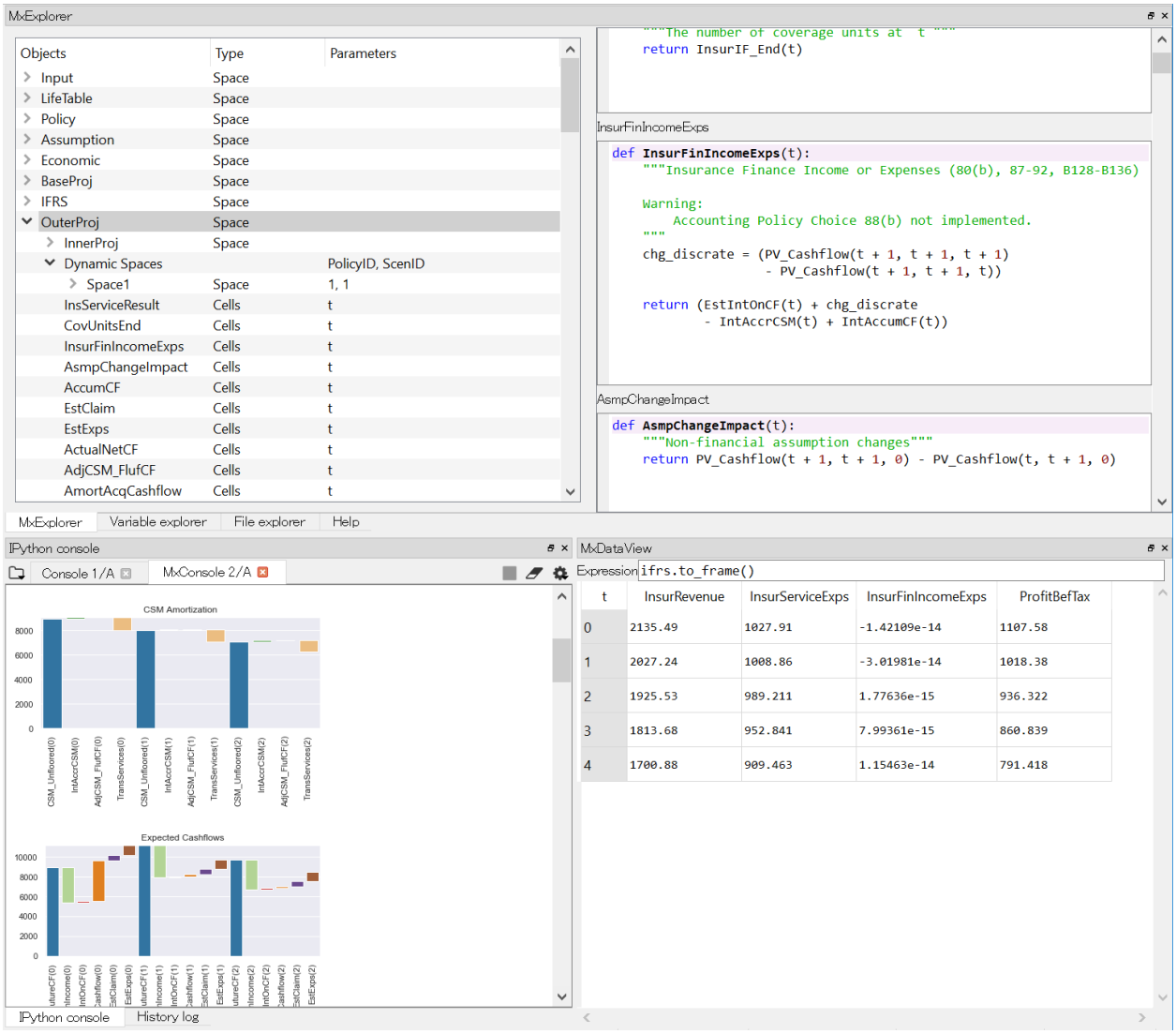


Fig. 1: Spyder plugin for modelx

CHAPTER 1

Latest Updates

- *19 October 2019:* modelx v0.0.25 is released. See *modelx v0.0.25 (19 October 2019)* release notes for details.
- *4 October 2019:* modelx v0.0.24 is released. See *modelx v0.0.24 (4 October 2019)* release notes for details.
- *8 August 2019:* modelx v0.0.23 is released. See *modelx v0.0.23 (9 August 2019)* release notes for details.
- *4 June 2019:* modelx v0.0.22 is released. See *modelx v0.0.22 (4 June 2019)* release notes for details.
- *24 March 2019:* spyder-modelx v0.0.9 is released. See *Release Notes* for details.
- *24 March 2019:* modelx v0.0.21 is released. See *Release Notes* for details.
- *2 February 2019:* modelx v0.0.20 is released. See *Release Notes* for details.
- *13 January 2019:* spyder-modelx v0.0.8 is released. See *Release Notes* for details.
- *13 January 2019:* modelx v0.0.19 is released. See *Release Notes* for details.

. . . See more updates

CHAPTER 2

What is modelx?

modelx is a Python package to build object-oriented models containing formulas and values to carry out complex calculations. You can think of it as a hierarchical and multidimensional extension of spreadsheet, but there's so much more to it!

CHAPTER 3

Feature highlights

modelx comes with features that enable users to interactively develop, run and scrutinize complex models in smart ways:

- Only little Python knowledge required
- Model composed of a tree of Spaces containing Cells
- Cells containing formulas and data
- Dynamic name binding for evaluating formulas within a Space
- Space inheritance
- Dynamic parametrized spaces created interactively
- GUI as Spyder plugin (spyder-modelx)
- Cells graph to track cells interdependency (Under development)
- Saving to / loading from files
- Conversion to Pandas objects
- Reading from Excel files

CHAPTER 4

Who is modelx for?

modelx is designed to be domain agnostic.

The modelx was created by actuary, and its primary use is to develop actuarial projection models. **lifelib** (https://lifelib.io) is a library of actuarial models that are built on top of modelx.

However, modelx is intentionally designed to eliminate domain specific features so that potential audience for modelx can be wider than actuaries, whoever needs to develop complex models of any sorts that are too much to deal with by spreadsheets.

CHAPTER 5

How modelx works

modelx exposes its API functions and classes such as Model, Space and Cells to its users, and the users build their models from those classes, by defining calculation formulas in the form of Python functions and associating those calculations with Cells objects.

Below is a very simple working example in which following operations are demonstrated:

- a new model is created,
- and in the model, a new space is created,
- and in the space, a new cells is created , which is associated with the Fibonacci series.

```
from modelx import *

model, space = new_model(), new_space()

@defcells
def fibo(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibo(n - 1) + fibo(n - 2)
```

To get a Fibonacci number for, say 10, you can do:

```
>>> fibo(10)
55
>>> fibo.series
n
0       0
1       1
2       1
3       2
4       3
5       5
```

(continues on next page)

(continued from previous page)

```
6      8
7     13
8     21
9     34
10    55
Name: fibo, dtype: int64
```

Refer to **lifelib** (https://lifelib.io) fo more complex examples.

CHAPTER 6

Python and modelx

Aside from modelx being a Python package and written entirely in Python, modelx utilizes Python in that it lets users define formulas by writing Python functions and converting it to modelx formulas. However, there is a critical difference between how Python functions are interpreted by Python and how modelx formulas are interpreted by modelx.

Python employs lexical scoping, i.e. the namespace in which function code is executed is determined by textual context. The global namespace of a function is the module that the function is defined in. In contrast, the evaluation of modelx formulas is based on dynamic scoping. Each Cells belongs to a space, and the space has associated namespace (a mapping of names to objects). The formula associated with the cells is evaluated in that namespace. So, what module a formula is defined (in the form of a Python function) does not affect the result of formula evaluation. It is what space the cells belongs to that affects the result.

CHAPTER 7

License

modelx is free software; you can redistribute it and/or modify it under the terms of GNU Lesser General Public License v3 (LGPLv3).

Contributions, productive comments, requests and feedback from the community are always welcome. Information on modelx development is found at Github https://github.com/fumitoh/modelx

CHAPTER 8

Development State

modelx is in its early alpha-release stage, and its specifications are subject to changes without consideration on backward compatibility. The source files of you models may need to be modified manually, if there are updates that break backward compatibility in newer versions of modelx.

Likewise, model files saved with one version may not load with a newer version. When updating modelx to a newer version, make sure you rebuild model files saved using older versions of modelx from their source code.

Warning: If you have embedded modelx in actuarial production processes, then it is encouraged to connect with the author on linkedin or on github , as modelx features you rely on might change or be removed in future releases without the author knowing those features are in use.

CHAPTER 9

History

modelx was originally conceived and written by Fumito Hamamura and it was first released in October 2017.

CHAPTER 10

What Next

- *Installation*
- *Tutorial*
- *Reference*

10.1 What's New

Warning: modelx is in its early alpha-release stage, and its specifications are subject to changes without consideration on backward compatibility. The source files of your models may need to be modified manually, if there are updates that break backward compatibility in newer versions of modelx.

Likewise, model files saved with one version may not load with a newer version. When updating modelx to a newer version, make sure you rebuild model files saved using older versions of modelx from their source code.

10.1.1 Updates

- *19 October 2019:* modelx v0.0.25 is released. See *modelx v0.0.25 (19 October 2019)* release notes for details.
- *4 October 2019:* modelx v0.0.24 is released. See *modelx v0.0.24 (4 October 2019)* release notes for details.
- *8 August 2019:* modelx v0.0.23 is released. See *modelx v0.0.23 (9 August 2019)* release notes for details.
- *4 June 2019:* modelx v0.0.22 is released. See *modelx v0.0.22 (4 June 2019)* release notes for details.
- *24 March 2019:* spyder-modelx v0.0.9 is released. See *Release Notes* for details.
- *24 March 2019:* modelx v0.0.21 is released. See *Release Notes* for details.
- *2 February 2019:* modelx v0.0.20 is released. See *Release Notes* for details.
- *13 January 2019:* spyder-modelx v0.0.8 is released. See *Release Notes* for details.
- *13 January 2019:* modelx v0.0.19 is released. See *Release Notes* for details.

. . . See more updates

Updates

- *19 October 2019:* modelx v0.0.25 is released. See *modelx v0.0.25 (19 October 2019)* release notes for details.
- *4 October 2019:* modelx v0.0.24 is released. See *modelx v0.0.24 (4 October 2019)* release notes for details.
- *8 August 2019:* modelx v0.0.23 is released. See *modelx v0.0.23 (9 August 2019)* release notes for details.
- *4 June 2019:* modelx v0.0.22 is released. See *modelx v0.0.22 (4 June 2019)* release notes for details.
- *24 March 2019:* spyder-modelx v0.0.9 is released. See *Release Notes* for details.
- *24 March 2019:* modelx v0.0.21 is released. See *Release Notes* for details.
- *2 February 2019:* modelx v0.0.20 is released. See *Release Notes* for details.
- *13 January 2019:* spyder-modelx v0.0.8 is released. See *Release Notes* for details.
- *13 January 2019:* modelx v0.0.19 is released. See *Release Notes* for details.
- *31 December 2018:* spyder-modelx v0.0.7 is released. See *Release Notes* for details.
- *31 December 2018:* modelx v0.0.18 is released. See *Release Notes* for details.
- *2 December 2018:* *Spyder plugin* page is added. A plugin image is added in the landing page.
- *2 December 2018:* spyder-modelx v0.0.6 is released. See *Release Notes* for details.
- *2 December 2018:* modelx v0.0.17 is released. See *Release Notes* for details.
- *27 October 2018:* The *Overview* page of this site is updated and merged into the landing page.
- *21 October 2018:* modelx v0.0.16 is released. See *Release Notes* for details.
- *20 September 2018:* modelx v0.0.15 is released. See *Release Notes* for details.
- *8 September 2018:* The *Installation* page is updated and *Installing Spyder plugin for modelx* section is added.
- *3 September 2018:* modelx v0.0.14 is released. See *Release Notes* for details
- *5 August 2018:* *modelx v0.0.13* is released.
- *5 August 2018:* The *Reference* guide is updated so that base class members are also described in their concrete classes' pages.
- *16 June 2018:* *modelx v0.0.12* is released.
- *27 May 2018:* modelx v0.0.11 is released.
- *6 May 2018:* modelx v0.0.10 is released.
- *20 April 2018:* The *Technical Guide* section is added.

10.1.2 Release Notes

modelx v0.0.25 (19 October 2019)

This release introduces a feature to trace the call stack of formula calculations in response to user's feature request. The tracing is useful when the user wants to get the information on the execution of cells formulas, such as how much time each formula takes from start to finish, or what formulas are called in what order to identify performance bottlenecks.

What's new in v0.0.25

- *Enhancements*
- *Bug Fixes*

Enhancements

- The API functions blow are introduce for the stack tracing feature (GH13).
  - `start_stacktrace()`
  - `stop_stacktrace()`
  - `get_stacktrace()`
  - `clear_stacktrace()`

Bug Fixes

- Error when writing models containing non-ascii strings as refs.

modelx v0.0.24 (4 October 2019)

Code around implementing inheritance is extensively refactored in this release, and a couple of small enhancements are incorporated in response to user's feature requests.

What's new in v0.0.24

- *Enhancements*
- *Bug Fixes*

Enhancements

- Models with modules included in them as references can now be saved with `save()` method (GH8 Comment).
- `name` parameter is added to `read_model()` to overwrite the opened model name (GH8).

Bug Fixes

- Getting cells values from the shell iteratively was too slow (GH12)

modelx v0.0.23 (9 August 2019)

What's new in v0.0.23

New Features

This release introduces methods to create Space and Cells from Pandas objects or CSV files.

When the model is written out, the data source Pandas objects are saved as files in the folder of the parent space. The CSV files are copied in the parent space folder.

Methods to Create Cells

- *UserSpace.new_cells_from_pandas*
- *UserSpace.new_cells_from_csv*

The first method above creates one or more cells in the parent space from a Pandas DataFrame or Series object passed as an argument. If a DataFrame is passed, created cells correspond to the DataFrame's columns. The second method creates cells from a CSV file. In either case, the created cells are populated with values read from the date source.

Methods to Create a Space and Cells

- *Model.new_space_from_pandas*
- *UserSpace.new_space_from_pandas*
- *Model.new_space_from_csv*
- *UserSpace.new_space_from_csv*

Those methods above create a UserSpace in the parent object (Model or UserSpace) from the data source (DataFrame/Series or CSV file) and then creates one or more Cells in the created space. The created UserSpace can have parameters by specifying which parameters should be interpreted as Space parameters in stead of Cells parameters. When the UserSpace has parameters, DynamicSpaces are also created in the UserSpace, and Cells in the DynamicSpaces are also populated with values from the data source.

Other Enhancements

- *write_model()* and *read_model()* now supports writing/reading models with multiple cells created together by the same execution of *new_cells_from_excel()* method.
- Added *modelx.models* attribute, an alias for *get_models()*

Backward Incompatible Changes

- `StaticSpace` is now renamed to *UserSpace*.

Bug Fixes

- Fix the default values of `names_row` and `param_cols` parameters of *new_cells_from_excel()*
- Fix an error when passing a lambda function whose definition spans across multiple lines in a function call.

modelx v0.0.22 (4 June 2019)

Overview

The most notable improvement among others in this release is the introduction of a new feature to write/read models to/from text files for better version control experience.

Prior to this release, models could only be saved("pickled") into a binary file. Maintaining models as binary files is not ideal for version control, as it disables the use of rich features offered by modern version control systems such as git. When you wanted to save a model as text, you needed to write the entire python script to build the model from the source files. Changes made to the models interactively through IPython console could not be saved as human-readable text.

This release introduces *write_model()* function (or equivalent *write()* method) and *read_model()* function, to write/read a model to/from a tree of folders containing text files.

The text files created by *write_model()* function are written as syntactically correct Python scripts with some literals expressed in JSON. However, in most cases they are not semantically correct. These files can only be interpreted through *read_model()* function.

Another notable improvement is the extended depth of formula recursion. Previously the maximum depth of formula recursion was set to 1000 by default. With this release the maximum depth is extended to 65000.

Enhancements

- Add *write_model()* function, *write()* method and *read_model()* function.
- The maximum depth of formula recursion is extended from 1000 to 65000 by default.
- Add `set_property` method to *Model*, *UserSpace*, *Cells*.
- Add `doc` method to *Model*, *UserSpace*, *Cells*.
- *new_space_from_excel()* can now create a static space when `space_param_order` is not given.

Backward Incompatible Changes

- Remove `_self_cells` and `_derived_cells` from *UserSpace*

Bug Fixes

- Fix *add_bases()* and *set_formula()*.

Older modelx releases

v0.0.21 (23 March 2019)

Updates include refactoring to separate static and dynamic space classes, use tuple for CellNode implementation, gaining approximately 20% performance improvement.

Backwards Incompatible Changes

- Space class is now split into two separate concrete classes, *UserSpace*, *DynamicSpace* and one base class BaseSpace.
- module_ parameter of the methods below are renamed to module.
  - *import_module()*
  - *import_funcs()*
  - *new_cells_from_module()*
  - *new_space_from_module()*
- Methods and properties below on Space classes are renamed to be private, as these are expected not to be used directly by users for normal usage.
  - *_is_base()*
  - *_is_sub()*
  - *_is_static()*
  - *_is_derived()*
  - *_is_defined()*
  - *_is_root()*
  - *_is_dynamic()*
  - _self_cells
  - _derived_cells
  - *_self_spaces*
  - *_derived_spaces*

Enhancements

- IPython error traceback message is not suppressed by default. *setup_ipython()* is added to suppress the default message.
- *set_recursion()* is added to change the maximum depth of formula recursion.

Bug Fixes

- Fix *formula* as setter by assignment expression i.e. alias to *set_formula()*.
- Fix *refs*.

v0.0.20 (2 February 2019)

Enhancements

- *CellNode* repr to show "parameter=arguments".
- Add *formula* property.

Bug Fixes

- Fix duplicate multiple bases of a dynamic space.

v0.0.19 (13 January 2019)

Enhancements / bug fixes for defining and using dynamics spaces whose base includes dynamics spaces.

Enhancements

- Add `name` parameter to *`open_model()`*.
- Pass dynamic arguments down to its children.
- Iterating over cells with single parameter returns values instead of tuples of single elements.
- View's _baseattrs to not include items with _ prefixed names.

Bug Fixes

- Fix bases of derived dynamic spaces. If dynamic spaces are to be the base spaces of a dynamic sub space, then the static base spaces of the dynamic spaces become the base spaces in replacement for the dynamic spaces.
- Fix *AttributeError: 'BoundFunction' object has no attribute 'altfunc'* on unpickled models.
- Dedent function definitions for those defined inside blocks of other function definition.
- Fix error in conversion to DataFrame when merging indexes with different types.

v0.0.18 (31 December 2018)

This release is mainly for adding interface functions/methods to spyder-modelx *v0.0.7 (31 December 2018)*

Enhancements

- Add *`preds`* and *`succs`* properties to *`CellNode`*.
- Add *`node()`* to *`Cells`*
- Rename `literaldict` property to `_baseattrs` for `Interface`, `BaseView` and their subclasses.
- Rename `set_keys` method of `SelectedView` to `_set_keys` .

Bug Fixes

- Raise not KeyError but AttributeError upon hasattr/getattr on Space.

v0.0.17 (2 December 2018)

This release is mainly for adding interface to functions to spyder-modelx *v0.0.6 (2 December 2018)*

Enhancements

- *get_object()* to get a modelx object from its full name.

Bug Fixes

- Error when modelx tries to get IPython shell before it becomes available.

v0.0.16 (21 October 2018)

spyder-modelx plugin introduces a new widget to view cells values in a table. This release reflects some updates in modelx to make the new widget work.

Enhancements

- *cur_model()* and *cur_space()* now accept model and space objects as their arguments respectively, in addition to the names of model or space objects.
- Add `model` property to all Interface subclasses.
- Traceback messages upon erroneous formula calls are now limited to 6 trace stack entries.
- Error messages upon erroneous formula calls are now simplified not to show file traceback.

Backwards Incompatible Changes

- The parameters to *cur_model()* and *cur_space()* are renamed from `name` to `model` and `space` respectively, due to the enhancement for these functions to accept objects, in addition to the names of the objects.

v0.0.15 (20 September 2018)

Enhancements

- Importing a module overrides formulas if their cells already exist.

v0.0.14 (3 September 2018)

This version is mainly for updating modelx Qt widgets, in order for the widgets to work with the initial version of spyder-modelx, Spyder plugin for modelx.

Enhancements

- Add property `_baseattrs` and `BaseView._baseattrs`. This property is used by spyder-modelx.

Bug Fixes

- Fix crashes when *cur_model()* is called with `name` argument to change the current model.

v0.0.13 (5 August 2018)

Space implementation has been largely rewritten in this release to make the inheritance logic more robust.

Warning: Support for Python 3.4, 3.5 is dropped in this release. Now only Python 3.6 and newer are supported. This is mainly due to the fact that modelx utilizes the order preservation nature of `dict` introduced in Python 3.6. `dict` performance improvement in Python 3.6 is also the reason to drop support for older versions.

Support for NetworkX ver 1.x is also dropped in this release. NetworkX version 2.x is now required.

Enhancements

- *`add_bases()`* and *`remove_bases()`* are added.
- *`bases`* is added.

Backwards Incompatible Changes

- Support for Python older than 3.6 is dropped. Now Python 3.6 or above is required.
- Support for NetworkX version 1 is dropped. Now NetworkX version 2 is required.
- Dynamic spaces now inherit their parent spaces by default.
- *`new_cells()`* raises an error when the cells already exists.
- *`formula`* now returns Formula object instead of string.

Bug Fixes

- `repr()` on SpaceView and CellsView now list only selected items.

v0.0.12 (16 June 2018)

Enhancements

- *`cells`* returns an immutable mapping of cells named *`CellsView`* supporting *`to_frame()`* method, which returns a DataFrame object containing cells values. If an iterator of arguments are given as `arg`, values of the cells for the arguments are calculated and only the given arguments are included in the DataFrame index(es). For more, see *`the reference page`*
- Cells are now of a Mapping type, which implements `keys()`, `values()`, `items()` methods to get their arguments and values.
- Subscription(`[]`) operator on *`cells`* now accepts multiple args of cell names and a sequence of cell names, such as `['foo', 'bar']` and `[['foo', 'bar']]`, which returns an immutable mapping (view) that includes only specified cells.

Backwards Incompatible Changes

- *`frame`* returns does not include empty or all-None cells.

Bug Fixes

- Fix issues specific to networkx v1.x.
- Fix `import_module()` to handle *bases* properly.

v0.0.11 (27 May 2018)

Bug Fixes

- Fix Space.refs
- Fix conversion of scalar cells to Pandas objects

v0.0.10 (6 May 2018)

Enhancements

- Add `is_*` methods to Space.
- Rename a model by adding `_BAKn` to its original name when another model with the same name is created.
- Add *`rename()`*.
- `name in space` expression is allowed where `name` is a string.
- `_space` local reference is available to refer to the parent space from its child cells.

Backwards Incompatible Changes

- `get_self` function is removed.

Bug Fixes

- Call stack max depth is set to 1000 to run all lifelib samples successfully.
- Fix an error around graph unpickling.
- Keep the same derived objects after they are updated.

v0.0.9 (1 April 2018)

Enhancements

- Add `show_tree` to show model tree in inline mode.
- Add `get_tree` to get model tree in automatic mode.

Bug Fixes

- Make `get_modeltree` available directly under `modelx`.

v0.0.8 (25 March 2018)

Enhancements

- Make `get_modeltree` available directly under `modelx`.
- Add `import_module()` and *`import_funcs()`* properties.
- Add *`all_spaces`* to contain all child spaces, including dynamic spaces.
- Add `self_spaces` and `derived_spaces` properties.
- Add `configure_python()` and `restore_python()`.
- Add *`reload()`* to reload the source module.
- *`Model`* and *`UserSpace`* to list their members on `dir()`.
- Raise an error upon zero division in formulas.
- Add `parent` property.

Backwards Incompatible Changes

- Base spaces are now indelible.
- *`spaces`* now contains only statics spaces. Now *`static_spaces`* is an alias to *`spaces`*.

Bug Fixes

- Remove overridden cells from `derived_cells`
- Update `self_cells` when new cells are added.
- Fix stack overflow with Anaconda 64-bit Python on Windows.

Thanks

- Stanley Ng

v0.0.7 (27 February 2018)

Backwards Incompatible Changes

- Renamed *`UserSpace`* constructor parameter `paramfunc` to `formula`.
- Renamed *`new_cells()`* parameter `func` to `formula`.
- Renamed `Interface` `can_have_none` to `allow_none`.

Bug Fixes

- Fix *`open_model()`* to make *`cur_model()`* properly return unpickled model.

spyder-modelx releases

v0.0.9 (24 March 2019)

- MxAnalyzer now has two tabs and supports successors as well as predecessors.
- MxExplorer and MxDataView now support multiple MxConsoles.
- Fix MxAnalyzer nodes holding Cells objects as their values.

v0.0.8 (13 January 2019)

- Fix MxAnalyzer crash.
- MxAnalyzer can now handle Cells with no arguments.

v0.0.7 (31 December 2018)

- Add MxAnalyzer widget. See *MxAnalyzer* section for details.
- MxExplorer now shows reference items.
- MxExplorer now displays dynamic spaces as parametrized expressions.

v0.0.6 (2 December 2018)

- Add Formula list widget to MxExplorer.
- Support Spyder 3.3.2.

v0.0.5 (21 October 2018)

- Add MxDataView widget. See *MxDataView* section for details.

10.2 Installation

Note: For lifelib users, when installing lifelib using `pip`, modelx is automatically installed due to its dependency, so no need to install modelx separately.

10.2.1 Python version

modelx requires Python 3.6 or newer. modelx does not work with Python 3 older than version 3.6 or any version of Python 2.

10.2.2 Package dependency

The packages listed below are either required by modelx, or can be used with modelx to develop models more efficiently.

- NetworkX (>=2.0)
- Pandas
- OpenPyXL
- Spyder (>=3.2.5)

networkx

NetworkX is a required package that modelx depends on. Version 2.0 or newer is required.

Pandas

Although you can install modelx without Pandas, it is highly recommended that you have Pandas installed, together with other packages Pandas depends on, such as NumPy, so that you can export Spaces and Cells to Pandas DataFrame and Series.

Openpyxl

OpenPyXL is a package that supports reading from and writing to Excel files. Openpyxl is also not required, but it is desirable to hav it installed to enable modelx to interface with Excel files.

Spyder

If you use modelx with Spyder, a plugin for modelx is available. `spyder-modelx` is a separate package to add custom IPython consoles and Modelx explorer, a widget that shows the current model in a tree view. The supported Spyder version is 3.2.5 or newer. For how to install the plugin, see *here*.

10.2.3 Installing modelx

Note: If you install *Spyder plugin for modelx* as explained below, no need to install modelx separately as modelx is installed together with the plugin, so skip to the *Plugin installation* section.

Just like other Python packages, you can install `modelx` by running `pip` command from a terminal on Linux, or from a command prompt on Windows.

To install the current version of `modelx` with `pip`:

```
$ pip install modelx
```

To upgrade to a newer version using the `--upgrade` flag:

```
$ pip install --upgrade modelx
```

To make `modelx` available only to you but others, install it into your user directory using the `--user` flag:

```
$ pip install --user modelx
```

If you prefer to install `modelx` from files placed locally on your machine instead of directly fetching from the Web, you can manually download `modelx` files from GitHub or PyPI.

Unpack the downloaded files and run the following command at the top of the source directory:

```
$ pip install .
```

10.2.4 Spyder integration

Spyder is a popular open-source Python IDE, and a Spyder plugin for modelx is avaialble. To install and use the plugin, see the *Spyder plugin* page

10.3 Spyder plugin

Spyder is a popular open-source Python IDE, and it's bundled in with Anaconda by default. Spyder allows plugins to be installed to add extra features to itself.

Spyder plugin for modelx enriches user interface to modelx in Spyder. The plugin adds custom IPython consoles and GUI widgets for using modelx in Spyder.

The plugin is under active development, and currently comes with a primary version of components, including:

- MxConsole
- MxExplorer
- MxDataView
- MxAnalyzer

MxConsole appears as tabs in Spyder's default IPython console, and runs custom IPython shells. Users should use these shells instead of Spyder's default shells in order for the other plugin widgets to interface with the user's Python sessions. The plugin widgets do not interface with Python running in default IPython consoles.

MxExplorer is the main plugin widgets, and it shows the object tree of the current modelx model in the active MxConsole. It can also lists formulas of cells in a selected space next to the tree.

MxDataView shows data in modelx objects in a spreadsheet-like tabular format.

MxAnalyzer is for tracking dependency of calculations. The user enter a Cell node, a combination of a Cell and arguments, and MxAnalyzer currently shows a tree of Cell nodes directly or indirectly used in calculating the value of the specified node.

The plugin widgets are "dockable" as Spyder's default widgets, meaning you can detach those widgets from the Spyder's main window to have their own separate windows, and "dock" them back in the main window at different positions to rearrange the widgets positions in the main window as you like.

10.3.1 Configure Spyder

Disable User Module Reloader

When you use modelx with Spyder, sometimes you may want to re-run the same file in the editor window multiple times in the same IPython session. You don't want to reload modelx because reloading modelx module creates multiple

instances of modelx systems within the same Python process, causing models created before and after a reload to reside in different modelx systems. To avoid that, you need to change *User Module Rloader (UMR)* setting.

From the Spyder menu, select *Tools->Preferences* to bring up Preferences window. Choose *Python interpreter* in the left pane, and you'll find an area titled *User Module Reloader (UMR)* on the bottom right side of the Preferences window. Leave *Enable UMR* option checked, click *Set UMR excluded(not reloaded) modules* and then UMR dialog box pops up as the figure blow. Enter "modelx" in the dialog box. This prevents Spyder from reloading the modelx module every time you re-run the same script from *Run* menu, while allowing other modules to be reloaded.

Note that you need to restart Spyder to bring the change into effect.

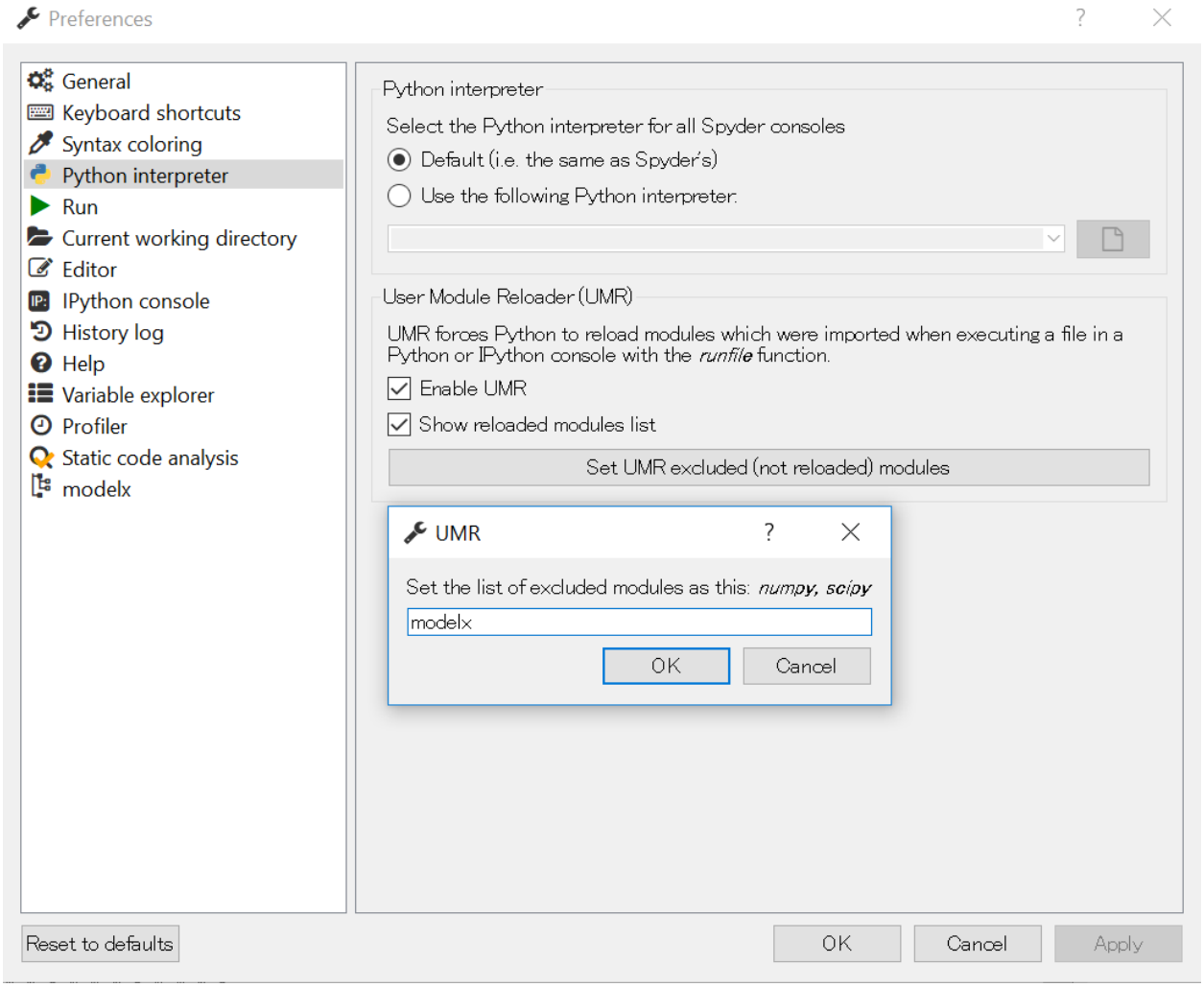


Fig. 1: User Module Reloader setting

10.3.2 Installing Spyder plugin for modelx

The plugin is available as a separate Python package named `spyder-modelx`.

The supported version of Spyder is 3.2.5 or newer. The plugin does not work with Spyder versions older than 3.2.5.

`spyder-modelx` package is available on PyPI, and can be installed using `pip` command.

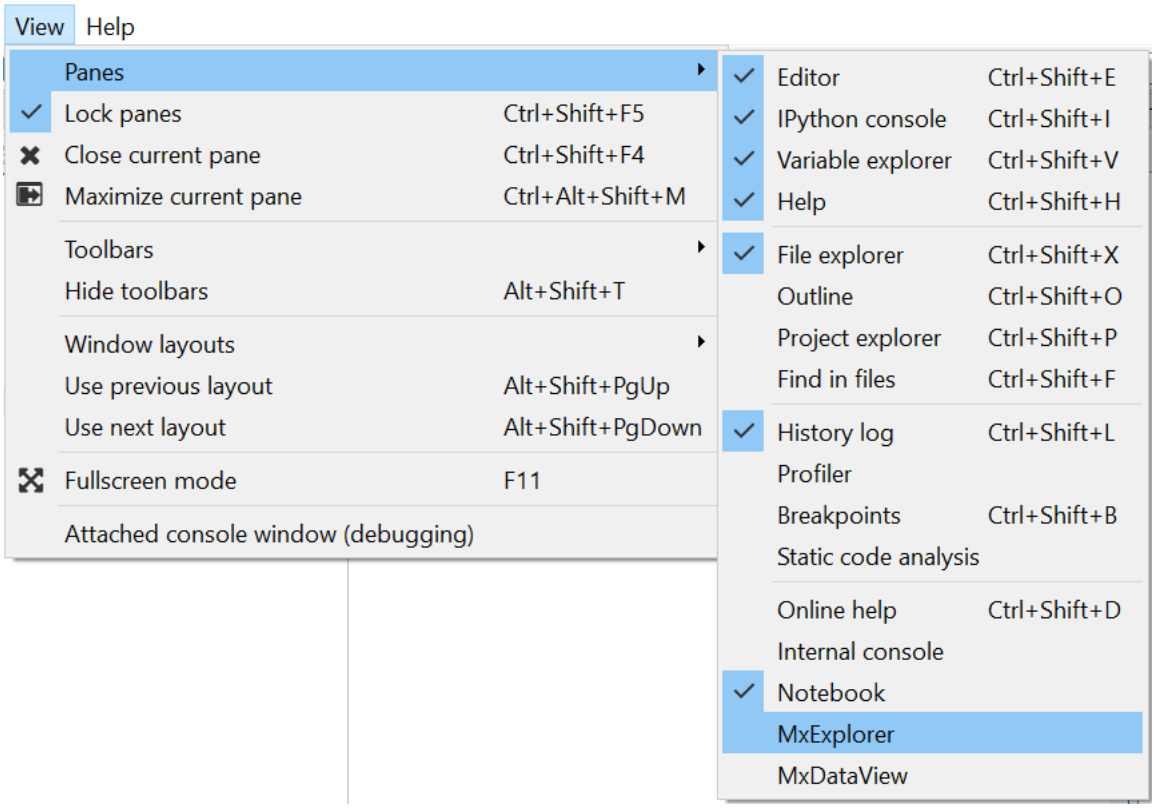
If you're using Anaconda distribution, run the following command in Anaconda command prompt to install the plugin after installing `modelx`:

```
$ pip install --no-deps spyder-modelx
```

If Spyder is running while the plugin gets installed, close Spyder once and restart it to bring the plugin into effect.

10.3.3 MxExplorer and MxConsole

To enable the modelx plugin, start Spyder, and go to *View->Panes* menu, and check *MxExplorer*.



Then the Modelx explorer tab appears in the upper right pane.

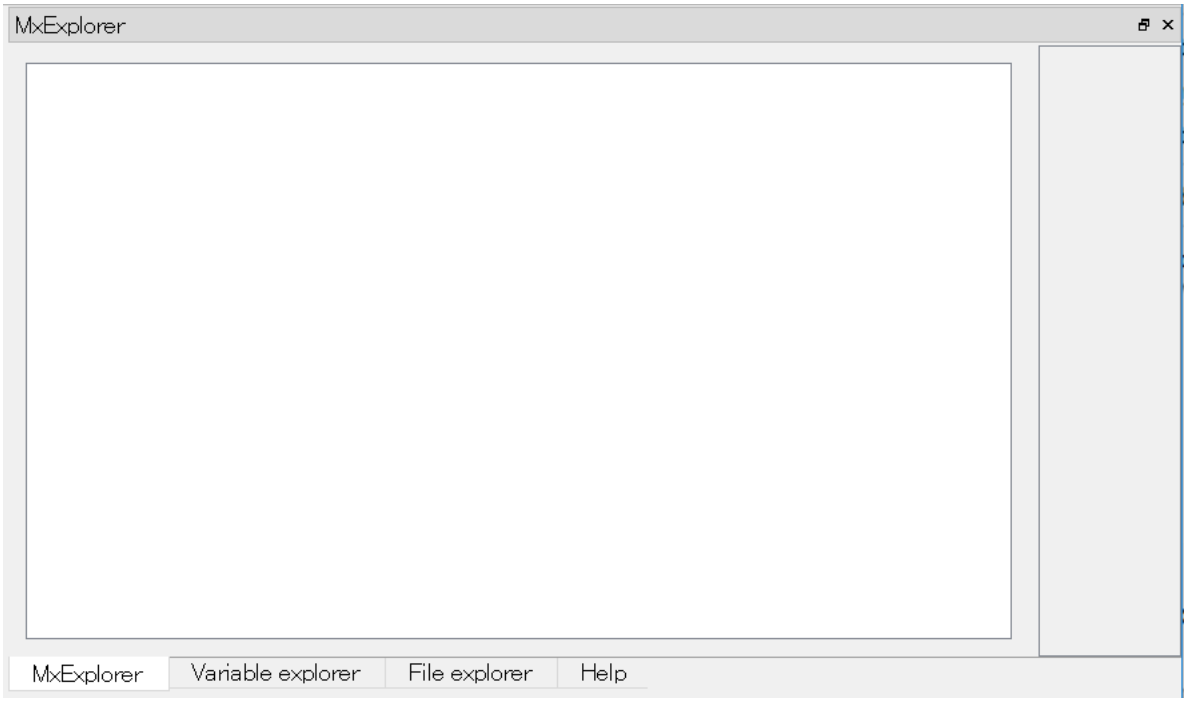
Right-click ont the IPython console tab in the lower right pane, then click *Open a MxConsole* menu.

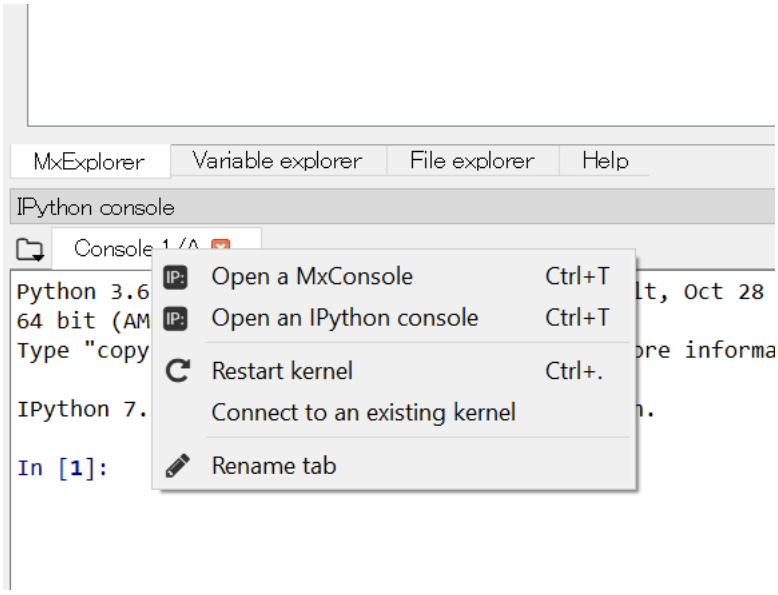
A modelx console named *MxConsole* starts. The modelx console works exactly the same as a regular IPython console, except that the modelx explorer shows the components of the current model in the IPython session of this console. To test the behaviour, create a new model and space in the modelx console like this:

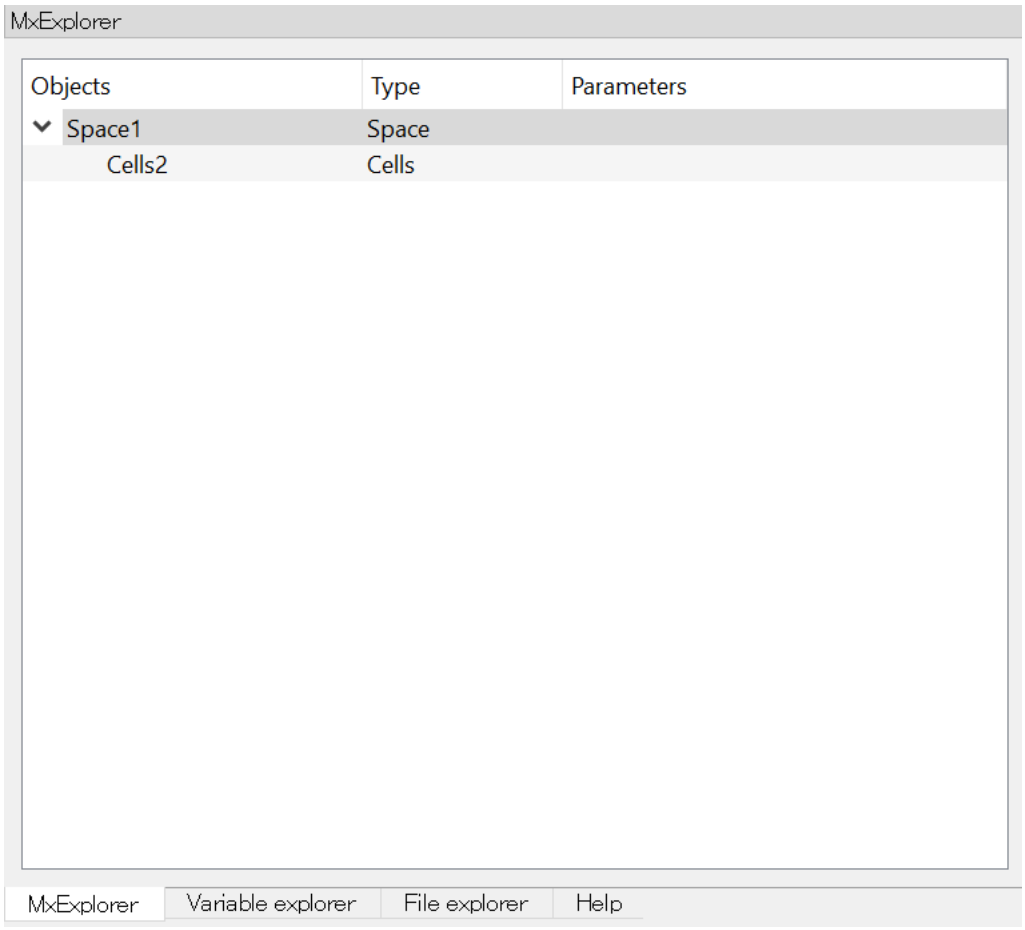
```
>>> import modelx as mx

>>> model, space = mx.new_model(), mx.new_space()

>>> cells = space.new_cells()
```

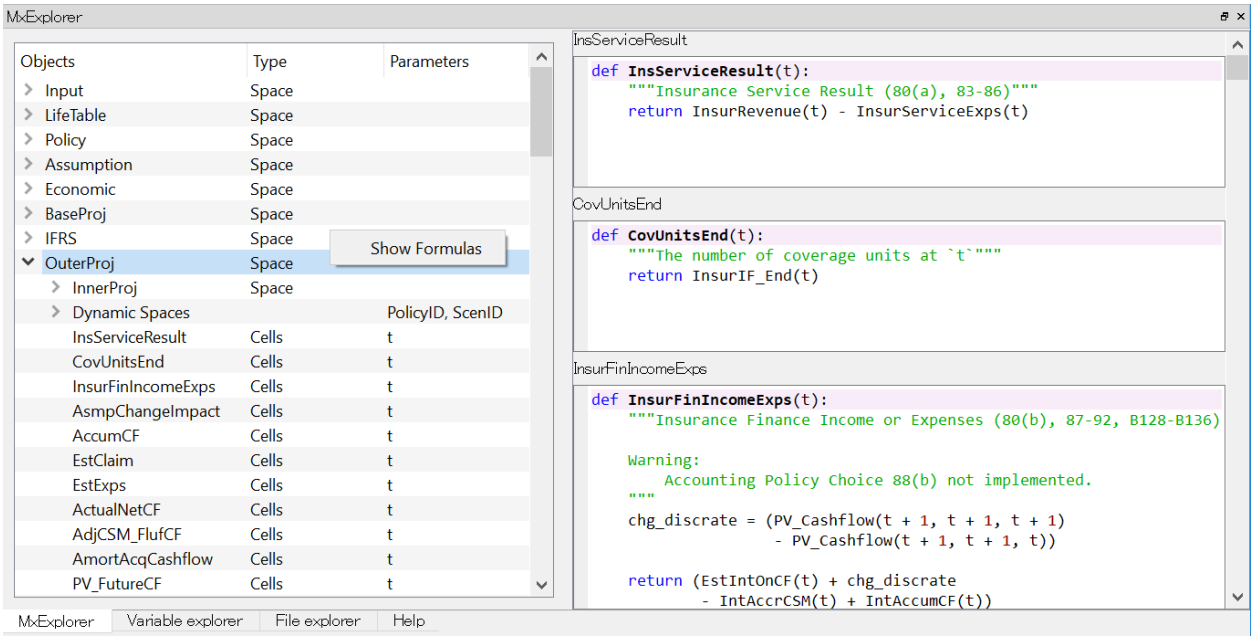
The modelx explorer shows the component tree of the created space.

MxExplorer can also list the formulas of all cells in a selected space. To see the formulas in a space, select the space in the tree, right-click to bring up the context menu, and the click *Show Formulas*. The list of the formulas appears to the right of the model tree in MxExplorer.


MxExplorer
MxExplorer
Variable explorer
File explorer
Help


MxExplorer
Variable explorer
File explorer
Help
IPython console
Open a MxConsole Ctrl+T
Open an IPython console Ctrl+T
Restart kernel Ctrl+.
Connect to an existing kernel
Rename tab
In [1]:

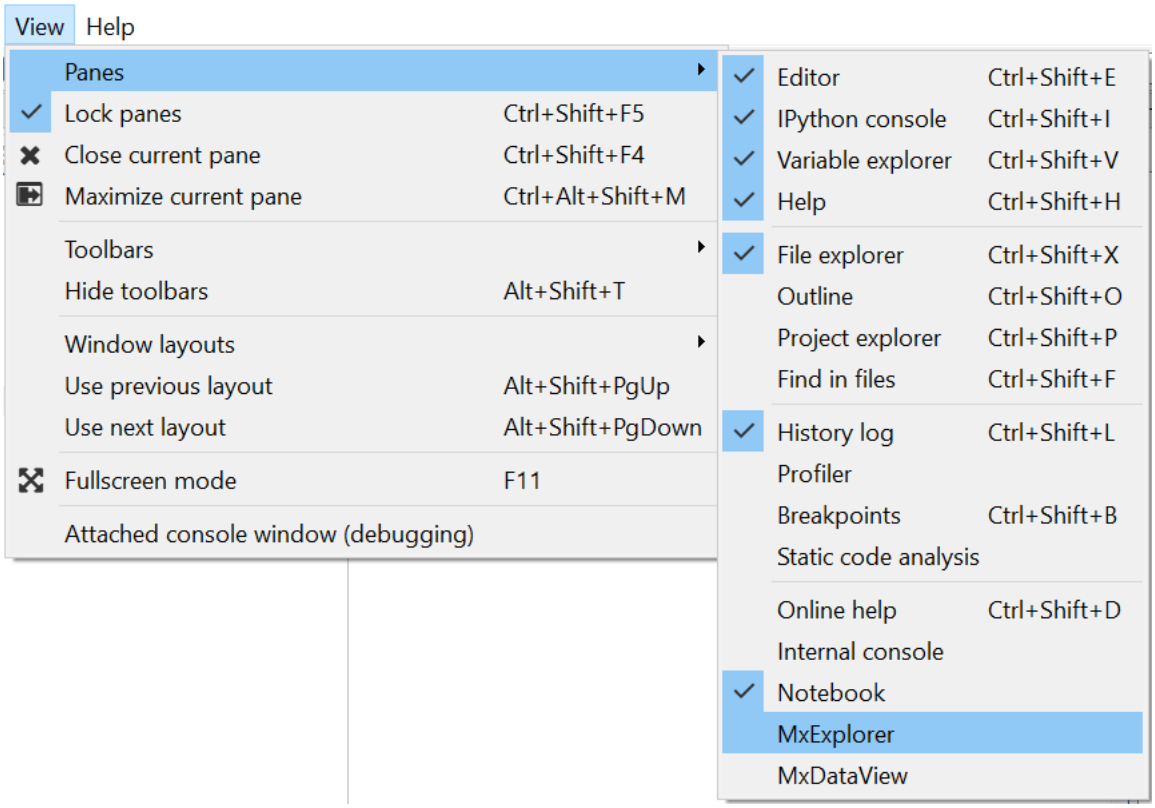

MxExplorer
Objects
Type
Parameters
Space1
Space
Cells2
Cells
MxExplorer
Variable explorer
File explorer
Help


MxExplorer
Objects
Type
Parameters
Input
Space
LifeTable
Space
Policy
Space
Assumption
Space
Economic
Space
BaseProj
Space
IFRS
Space
Show Formulas
OuterProj
Space
InnerProj
Space
Dynamic Spaces
PolicyID, ScenID
InsServiceResult
Cells
t
CovUnitsEnd
Cells
t
InsurFinIncomeExps
Cells
t
AsmpChangeImpact
Cells
t
AccumCF
Cells
t
EstClaim
Cells
t
EstExps
Cells
t
ActualNetCF
Cells
t
AdjCSM_FlufCF
Cells
t
AmortAcqCashflow
Cells
t
PV_FutureCF
Cells
t
InsServiceResult
def InsServiceResult(t):
"""Insurance Service Result (80(a), 83-86)"""
return InsurRevenue(t) - InsurServiceExps(t)
CovUnitsEnd
def CovUnitsEnd(t):
"""The number of coverage units at `t`"""
return InsurIF_End(t)
InsurFinIncomeExps
def InsurFinIncomeExps(t):
"""Insurance Finance Income or Expenses (80(b), 87-92, B128-B136)
Warning:
Accounting Policy Choice 88(b) not implemented.
"""
chg_discrate = (PV_Cashflow(t + 1, t + 1, t + 1)
- PV_Cashflow(t + 1, t + 1, t))
return (EstIntOnCF(t) + chg_discrate
- IntAccrCSM(t) + IntAccumCF(t))
MxExplorer
Variable explorer
File explorer
Help

10.3.4 MxDataView

MxDataView widget lets you see a DataFrame object in a spreadsheet-like tabular format.

If MxDataView widget is not shown, Go to *View->Panes* menu as you did with MxExploer, and check *MxDataView*.



To specify a DataFrame object to display, enter a Python expression that returns the DataFrame object, in the text box labeled *Expression*. The Python expression is evaluated in the global namespace of the Python session in the active MxConsole. The expression is re-evaluated every time MxConsole execute a Python command.

10.3.5 MxAnalyzer

MxAnalyzer is used for checking calculation dependency.

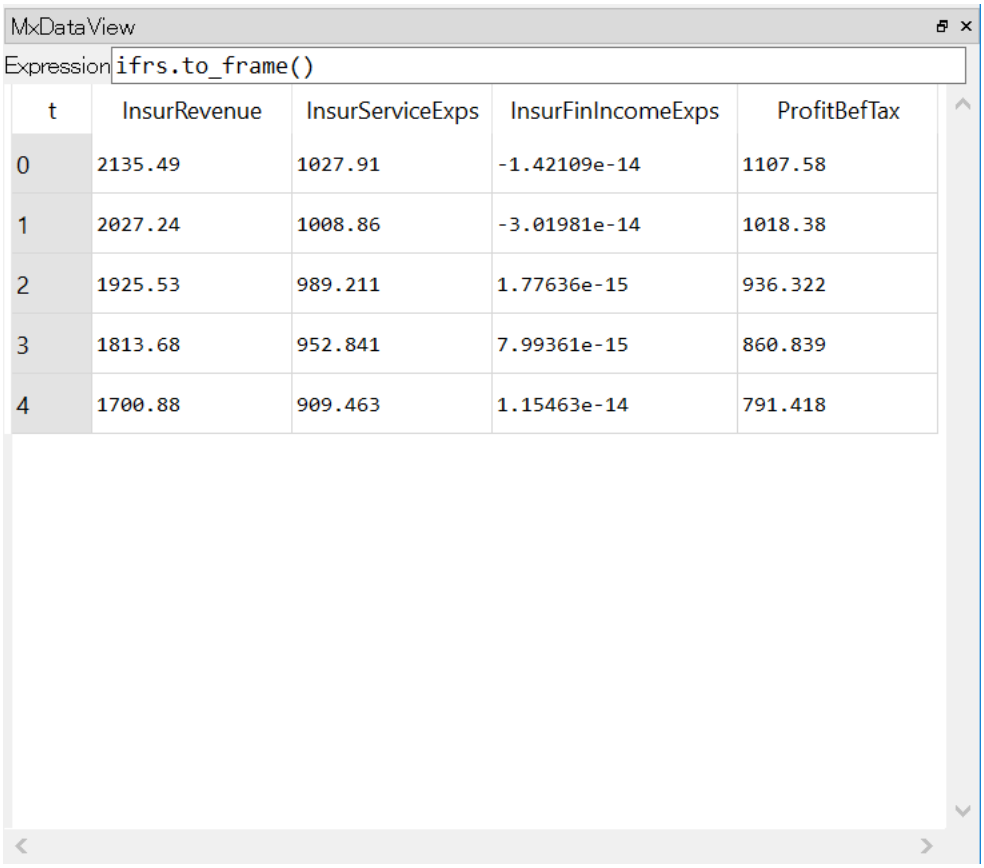
If MxDataView widget is not shown, Go to *View->Panes* menu as you did with MxExploer, and check *MxAnalyzer*.

Enter an expression that returns a Cell object in the top-left box, and arguments to the Cell in the to-right box. The Python expression is evaluated in the global namespace of the Python session in the active MxConsole. Then MxAnalyzer shows a tree of Cell nodes directly or indirectly used in calculating the value of the specified node.

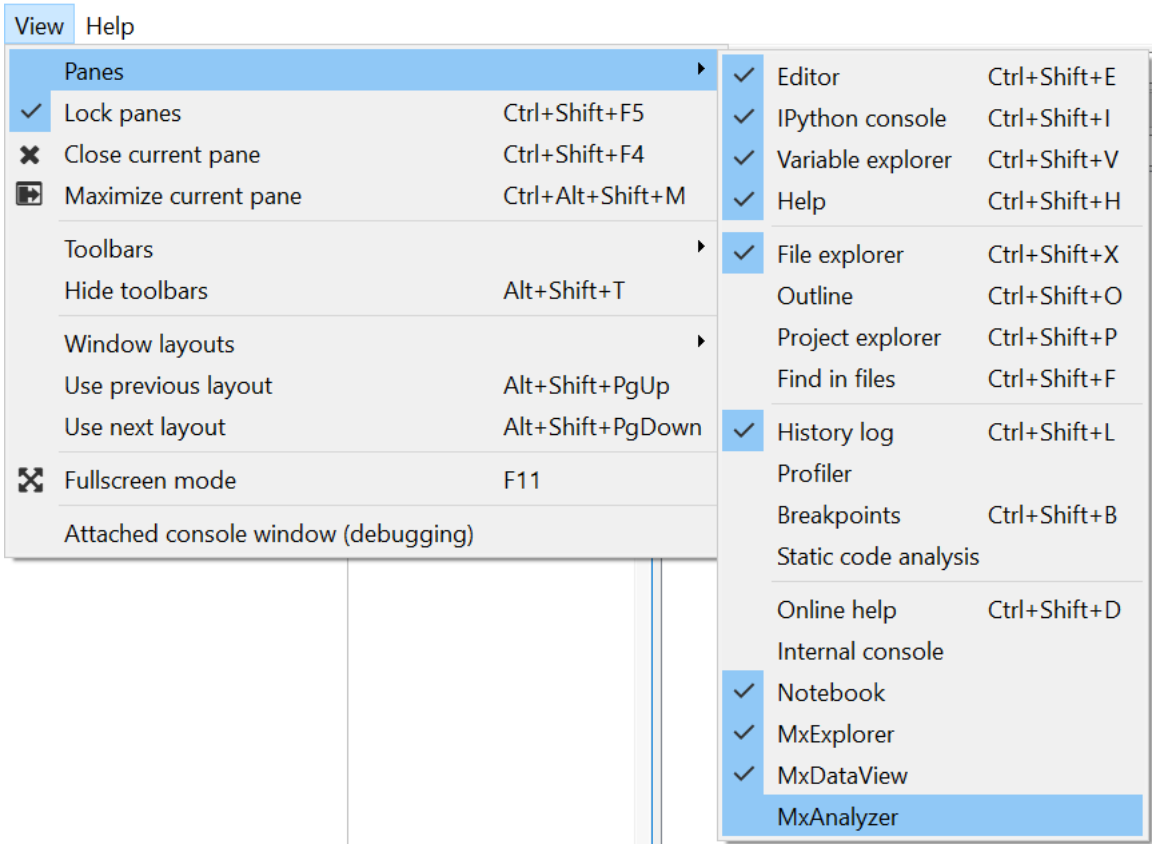
10.4 Tutorial

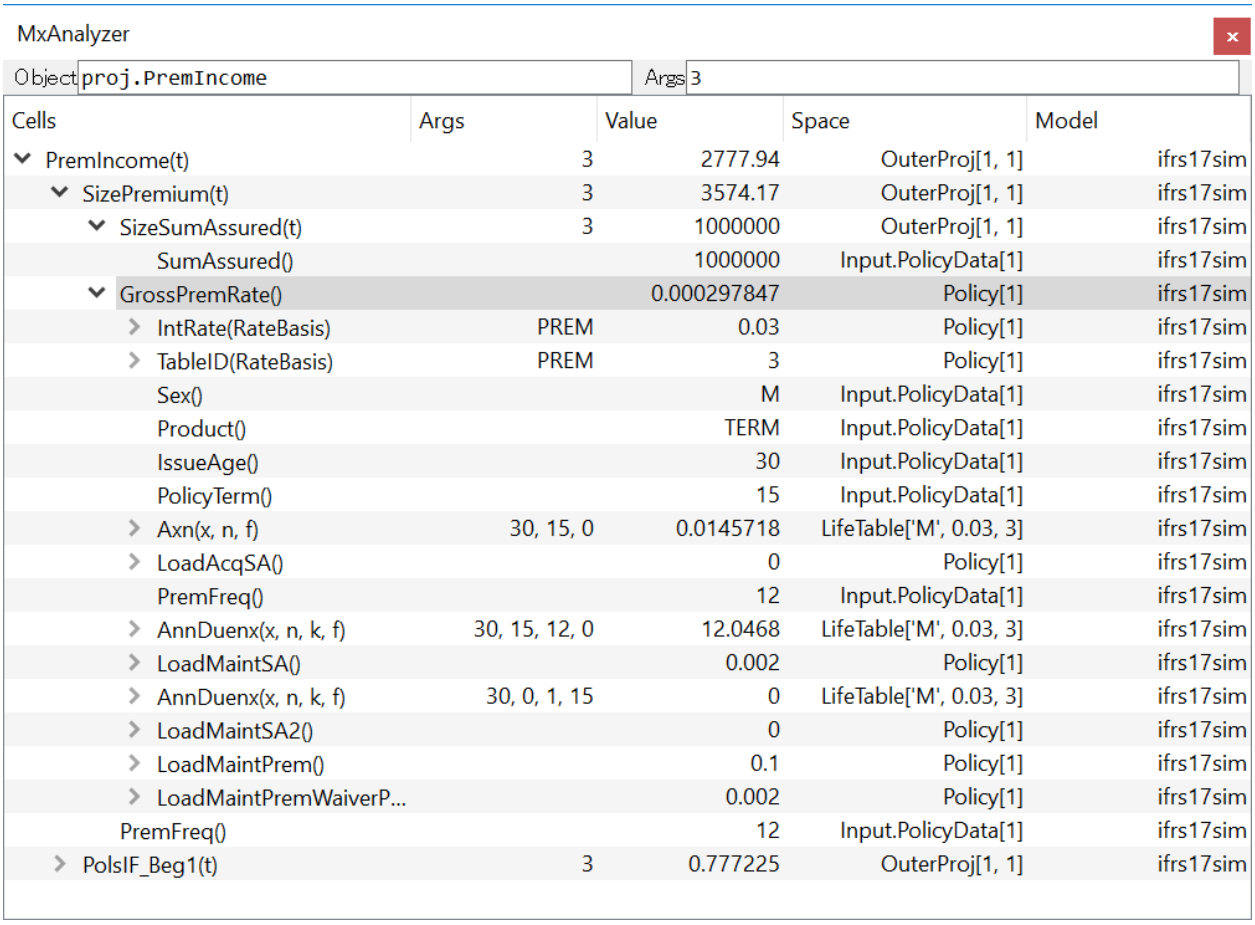
This tutorial aims to introduce core concepts and features of modelx, and demonstrate how to use modelx by going through some examples.

This tutorial supplements modelx reference, which is build from docstrings of the API functions and classes. The reference should cover the details of each API element, which may not be fully explained in this tutorial.



t	InsurRevenue	InsurServiceExps	InsurFinIncomeExps	ProfitBefTax
0	2135.49	1027.91	-1.42109e-14	1107.58
1	2027.24	1008.86	-3.01981e-14	1018.38
2	1925.53	989.211	1.77636e-15	936.322
3	1813.68	952.841	7.99361e-15	860.839
4	1700.88	909.463	1.15463e-14	791.418





Cells	Args	Value	Space	Model
PremIncome(t)	3	2777.94	OuterProj[1, 1]	ifrs17sim
SizePremium(t)	3	3574.17	OuterProj[1, 1]	ifrs17sim
SizeSumAssured(t)	3	1000000	OuterProj[1, 1]	ifrs17sim
SumAssured()		1000000	Input.PolicyData[1]	ifrs17sim
GrossPremRate()		0.000297847	Policy[1]	ifrs17sim
IntRate(RateBasis)	PREM	0.03	Policy[1]	ifrs17sim
TableID(RateBasis)	PREM	3	Policy[1]	ifrs17sim
Sex()		M	Input.PolicyData[1]	ifrs17sim
Product()		TERM	Input.PolicyData[1]	ifrs17sim
IssueAge()		30	Input.PolicyData[1]	ifrs17sim
PolicyTerm()		15	Input.PolicyData[1]	ifrs17sim
Axn(x, n, f)	30, 15, 0	0.0145718	LifeTable['M', 0.03, 3]	ifrs17sim
LoadAcqSA()		0	Policy[1]	ifrs17sim
PremFreq()		12	Input.PolicyData[1]	ifrs17sim
AnnDuenx(x, n, k, f)	30, 15, 12, 0	12.0468	LifeTable['M', 0.03, 3]	ifrs17sim
LoadMaintSA()		0.002	Policy[1]	ifrs17sim
AnnDuenx(x, n, k, f)	30, 0, 1, 15	0	LifeTable['M', 0.03, 3]	ifrs17sim
LoadMaintSA2()		0	Policy[1]	ifrs17sim
LoadMaintPrem()		0.1	Policy[1]	ifrs17sim
LoadMaintPremWaiverP...		0.002	Policy[1]	ifrs17sim
PremFreq()		12	Input.PolicyData[1]	ifrs17sim
PolsIF_Beg1(t)	3	0.777225	OuterProj[1, 1]	ifrs17sim

10.4.1 Typical workflow

modelx is a Python package, and you use it by writing a Python script and importing it, as you would normally do with any other Python package.

modelx is best suited for building complex numerical models composed of many formulas referencing each other, so when you start from scratch, the typical workflow would be to first write code for building a model, and then evaluate the model.

As we are going to see, it takes more than one line of code to build a model, so it's convenient to use a Python shell that allows you edit and execute a chunk of code at once for building a model, then get into an interative mode for letting you examine the model one expression or statement at a time.

IDLE, the Tk/Tcl based simple Python GUI shell that comes with CPython lets you do that. You can open an editor window, and when the part of building a model is done, you can press F5 to save and run the script in a Python shell window where you are prompted to enter Python code to evaluate the model. Jupyter Notebook and many other popular Python shell environments have similar capability.

10.4.2 Model, Space and Cells

Before taking a look at the very first example, you might want to have an idea on what Model, Space and Cells are, as those three types of objects are central to modelx.

Model, Space and Cells are to modelx what workbook, worksheet and cells are to a spreadsheet program respectively, although there are differences. The diagram below illustrates containment relationships between those objects.

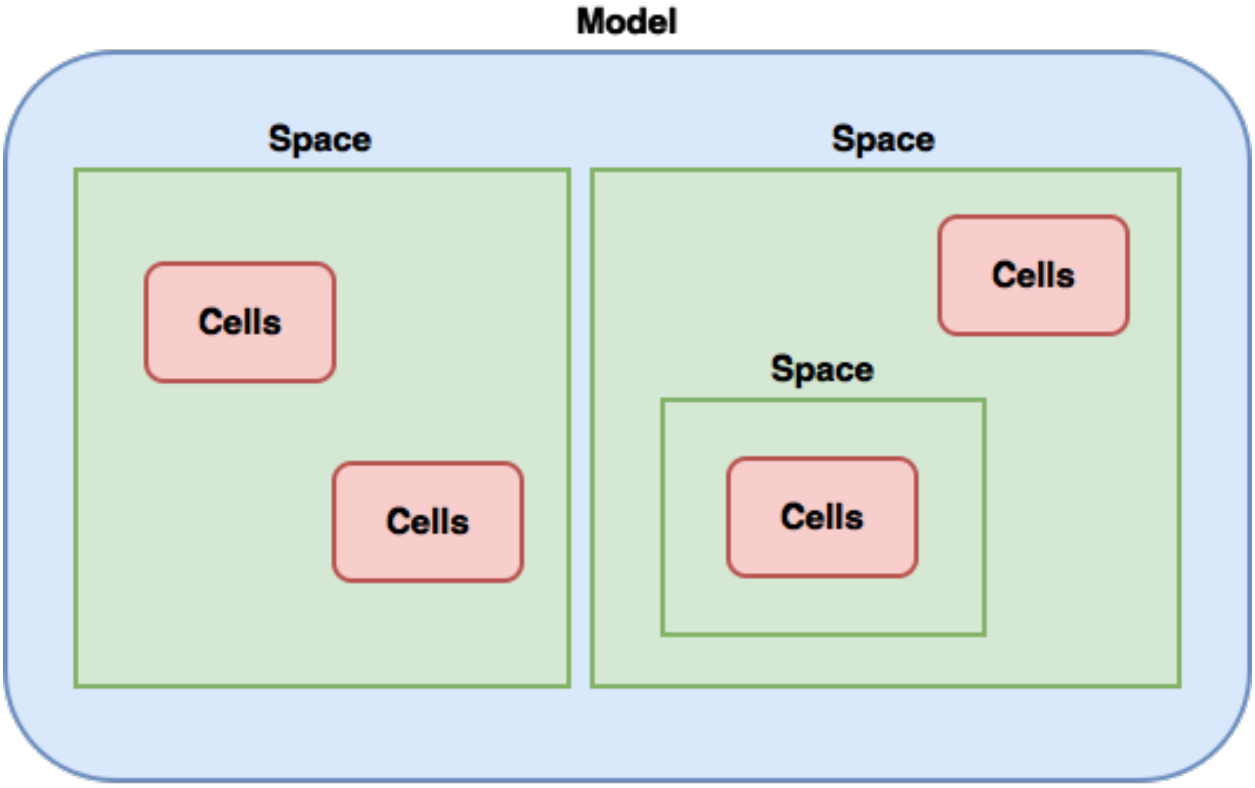


Fig. 2: Model, Space and Cells

A model is a workspace that contains all the modelx objects. It can be saved to a file and loaded again. A model contains spaces. In turn, spaces can contain cells and also other spaces (subspaces). Each cells ('cells' is singular

here) belongs to one and only one space. We call the object that contains another object, the parent of the contained object. The parent of a cells is the space that contains the cells, and the parent of a space is the model or space that contains the space. Spaces also serves as the namespace for contained cells but we'll get to this later.

A Cells can have a formula that calculates the cells' values, just like spreadsheet cells can have formulas. Cells values are either calculated by the formula or assigned as an input. We will learn how to define cells formulas through the examples soon.

10.4.3 Basic Operation

In this section, we'll start learning how to perform basic operations, such as creating models, spaces and cells, by talking a closer look at the simple example we saw in the overview section.

```
from modelx import *

model, space = new_model(), new_space()

@defcells
def fibo(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibo(n - 1) + fibo(n - 2)
```

Importing modelx

To start using modelx, import the package by the import statement, as is the case with any other package.

```
from modelx import *
```

By doing so, you get to use modelx API functions in `__main__` module. The entire list of modelx API functions are *Function Reference*. If you're not comfortable with importing modelx API functions directly into the global namespace of `__main__` module, you can alternatively import `modelx` as an abbreviated name, such as `mx`, for example:

```
import modelx as mx
```

in which case you can use modelx API functions prepended with `mx.`. We'll assume importing `*` in this tutorial, but be reminded that this is not a good practice when you write Python modules.

Creating a Model

The next statement performs two assignments in one line to make better use of horizontal space, but we'll decompose it into to the two assignment statements below for the sake of explanation:

```
model = new_model()
space = new_space()
```

In the first line, *`new_model()`* is a modelx API function that create a new model and returns it.

You can specify the name of the model by passing it as `name` argumet to the function, like `new_model(name='MyModel')`. If no name is given as the argument, the returned model is named automatically by modelx.

Creating a Space

```
space = new_space()
```

new_space(), in the line above creates a new space in the "current" model. In this case, the current model is set to the one we just created. modelx keeping track of the current model is somewhat analogous to how a spareadsheet program has "active" book.

Just as with the model, the name of the space can be specified by passing it to the method `name` argument, otherwise the space gets its name by modelx.

If you want to create a space in a model other than the current model, you can call `new_space()` method on the model, with or without the space name as its argument:

```
>>> space = model.new_space('MySpace')
```

Getting Models

To get all existing models, you can use `get_models()` function, which returns a mapping of the names of all existing models to the model objects:

```
>>> get_models()
{'Model1': <Model Model1>}
```

To get the current model, use *cur_model()* without arguments.

Getting Spaces

To get all spaces in a model mapped to their names, you can check `spaces` property of the model:

```
>>> model.spaces
mappingproxy({'Space1': <Space Space1 in Model1>})
```

The return MappingProxy objects acts like an immutable dictionary, so you can get Space1 by `model.spaces['Space1']`. You can see the returned space is the same object as what is referred as `space`:

```
>>> space is model.spaces['Space1']
True
```

To get one space, its name is available as an attribute of the containing model:

```
>>> model.Space1
<Space Space1 in Model1>
```

You can get the current space of the current model by calling *cur_space()* without arguments.

Creating Cells

There are a few ways to create a cells object and defiene the formula associated with the cells. As seen in the example above, one way is to define a python function with `defcells` decorator.

```
model, space = new_model(), new_space()

@defcells
def fibo(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibo(n - 1) + fibo(n - 2)
```

By `defcells` decorator, the name `fibo` in this scope points to the Cells object that has just been created from the formula definition.

By this definition, the cells is created in the current space in the current model. modelx keeps the last operated model as the current model, and the last operated space for each model as the current space. *`cur_model()`* API function returns the current model, and *`cur_space()`* method of a model holds its current space.

To specify the space to create a cells in, you can pass the space object as an argument to the `defcells` decorator. Below is the same as the definition above, but explicitly specifies in what space to define the cell:

```
@defcells(space)
def fibo(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibo(n - 1) + fibo(n - 2)
```

There are other ways to create cells by `defcells`. Refer to *`defcells()`* section in the reference manual for the details.

Another way to create a cells is to use Space's *`new_cells()`* method. Not that the command below doesn't work in the current context as we've already defined `fibo`:

```
>> space.new_cells(formula=fibo)
```

The `func` parameter can either be a function object, or a string of function definition.

Getting Cells

Similar to spaces in a model contained in the `spaces` property of the model, cells in a space are associated with their names and contained in the `cells` property of the model:

```
>>> fibo is space.cells['fibo']
True
```

As you can get a space in a model by attribute access with `.`, you can get a cells in a space by accessing the space attribute of the cells name with `.`:

```
>>> space.fibo
<Cells fibo(n) in Model1.Space1>
>>> fibo is space.fibo
True
```

Getting Values

The cells `fibo` does not have values yet right after it is created. To get cells' value for a certain argument, simply call `fibo` with the paratmer in parenthesis or in squre brackets:

```
>>> fibo[10]
55
>>> fibo(10)
55
```

Its values are calculated automatically by the associated formula, when the cells values are requested. Note that values are calculated not only for the specified argument, but also for the arguments that recursively referenced by the formula in order to get the value for the specified argument. To see for what arguments values are calculated, export `fibo` to a Pandas Series object. (You need to have Pandas installed, of course.):

```
>>> fibo[10]
55
>>> fibo.series
n
0       0
1       1
2       1
3       2
4       3
5       5
6       8
7      13
8      21
9      34
10     55
Name: fibo, dtype: int64
```

Since `fibo[10]` refers to `fibo[9]` and `fibo[8]`, `fibo[9]` refers to `fibo[8]` and `fibo[7]`, and the recursive reference goes on until it stops and `fibo[1]` and `fibo[0]`, values of `fibo` for argument `0` to `10` are calculated by just calling `fibo[10]`.

Note: It is important to understand in what namespace cells formulas are executed. Unlike Python functions, the global namespace of a cells formula has nothing to do with where in the source files the formula is defined. The names in the formula are resolved in the namespace associated with the cells' parent space. In that namespace, available names are cells contained in the space, spaces contained in the space (i.e. the subspaces of the space) and "references" accessible in the space.

Clearing Values

To clear cells values, you can use `clear()` method. Below shows what happens when the value of `fibo` at n = 5 is cleared:

```
>>> fibo.clear(5)
>>> fibo.series
n
0    0
1    1
2    1
3    2
```

(continues on next page)

(continued from previous page)

```
4    3
Name: fibo, dtype: int64
```

As you can see, not only at n = 5, but also for n = 6 to 10 values of `fibo` are cleared. This is because the calculations of `fibo[6]` to `fibo[10]` depend on the value of `fibo[5]`. Dependent values are cleared all together with the specified value.

To clear all values, simply call `clear()` witthout arguments:

```
>>> fibo.clear()
>>> fibo.series
Series([], Name: fibo, dtype: float64)
```

Setting Values

Other than letting the formula calculate cells values, you can input cells values manually by the set item (`[] =`) operation. If the cells already has a value at the specified parameter value, then the values of dependent cells are cleared first, then the specified value is assigned:

```
>>> fibo[10]
55
>>> fibo.series
n
0       0
1       1
2       1
3       2
4       3
5       5
6       8
7      13
8      21
9      34
10     55
Name: fibo, dtype: int64

>>> fibo[5] = 0
>>> fibo.series
n
0    0
1    1
2    1
3    2
4    3
5    0
Name: fibo, dtype: int64
```

10.4.4 Advanced Concepts

In this section, more concepts we haven't yet covered are introduced. Some of them are demonstrated by examples in the following section.

Space Members

Spaces can contain cells and other spaces. In fact, spaces have 3 kinds of their "members". You can get those members as if they are attributes of the containing spaces, by attribute access(`.`) expression.

Cells As we have seen in the previous example, spaces contain cells, and the cells belong to spaces. One cells must belong to one ane only one space.

The *`cells`* property of Space returns a dictionary of all the cells associated with their names.

(Sub)spaces As previously mentioned, spaces can be created in another space. Spaces in another space are called subspaces of the containing space. There 2 kinds of subspaces, static subspaces and dynamic subspace.

Static subspaces are those that are created manually, just like those created in models. There is no difference between spaces created directly under a model and static spaces created under a space, except for their parents being different types.

You can create a static subspace by calling `new_space` method of their parents:

```
model, space = new_model(), new_space()

space.new_space('Subspace1')

@defcells
def foo():
    return 123
```

You can get a subspace as an attribute of the parent space, or by accessing the parent space's *`spaces`* property:

```
>>> space.spaces['Subspace1'].foo()
123

>>> space.Subspace1.foo()
123
```

The other kind of subspaces, is dynamic subspaces. Unlike static suspaces, dynamic subspaces can only be created in spaces, but not directly in models.

Dynamic spaces are parametrized spaces that are created on-the-fly when requested through call(`()`) or subscript(`[]`) operation on their parent spaces. We'll explore more on dynamic spaces in the next section, in conjunction with space inheritance by going through an example.

References Often times you want access from cells formulas in a space to other objects than cells or subspaces in the same space. References are names bound to arbitrary objects that are accessible from within the same space:

```
model, space = new_model(), new_space()

@defcells
def bar():
    return 2 * n
```

`bar` cells above refers to `n`, which has not yet been defined. Without `n` being defined, calling `bar` will raise an error. To define a reference `n`, you can simply assign a value to `n` attribute of `space`:

```
>>> space.n = 3
>>> bar()
6
```

The *`refs`* property of space returns a mapping of reference names to their objects:

```
>>> list(space.refs.keys())
['__builtins__', 'n', '_self']
```

Be default, `__builtins__` and `_self` are defined in any space. In fact, `__builtins__` is defined by default as a "global" reference in the model. Global references are names accessible from any space in a model. Other than the default reference, you can define your own, by simply assigning a value as an attribute of the model:

```
>>> model.z = 4
>>> list(model.refs.keys())
['z', '__builtins__']

>>> list(space.refs.keys())
['z', '__builtins__', 'n', '_self']
```

`__builtins__` points to Python builtin module. It is defined to allow cells formulas to use builtin functions. `_self` points to the space itself. This allows cells formulas to get access to its parent space.

As mentioned earlier, formulas of cells are evaluated in the namespace that is associated with their parent spaces.

The namespace of a space is a mapping of names to the space members. As explained in the previous section, space members are either cells of the space, or subspaces of the space or references accessible from the space.

The table below breaks down all the members in the namespace by its types and sub-types.

cells	self cells	Cells defined in or overridden in the space
	derived cells	Cells inherited from one of the base spaces
spaces	self spaces	Subspace defined in or overridden in the space
	derived spaces	Subspace inherited from one of the base spaces
references (refs)	self references	References defined in or overridden in the space
	derived references	References inherited from one of the base spaces
	global references	Global references defined in the parent model
	local references	Only `_self` that refers to the space itself
	parameters	(Only in dynamic spaces) Space parameters

Each type of the members has "self" members and "derived" members. Those distinctions stem from space inheritance explained in the next section.

Space Inheritance

Space inheritance is a concept analogous to class inheritance in object-oriented programming languages. By making full use of space inheritance, you can organize multiple spaces sharing similar features into an inheritance tree of spaces, minimizing duplicated formula definitions, keeping your model organized and transparent while maintaining model integrity.

Inheritance lets one space use(inherit) other spaces, as base spaces. The inheriting space is called a derived space of the base spaces. The cells in the base spaces are copied automatically in the derived space. In the derived space, formulas of cells from base spaces can be overridden. You can also add cells to the derived space, that do not exist in any of the base spaces.

A space can have multiple base spaces. This is called multiple inheritance. The base spaces can have their base spaces, and derived-base relationships between spaces make up a directional graph of dependency. In case of multiple inheritance, we need a way to order base spaces to determine the priority of base spaces. modelx uses the same algorithm as Python for ordering bases, which is, C3 superclass linearization algorithm (a.k.a C3 Method Resolution Order or MRO). The links below are provided in case you want to know more about C3 MRO.

- https://en.wikipedia.org/wiki/C3_linearization
- https://www.python.org/download/releases/2.3/mro/

10.4.5 More complex example

Let's see how inheritance works by a simple code of pricing life insurance policies. First, we are goint to create a very simple life model as a space and name it `Life`. Then we'll populate the space with cells that calculate the number of death and remaining lives by age.

Then to price a term life policy, we will derive a `TermLife` space from the `Life` space, and add some cells to calculate death benefits paid to the insured, and their present value.

Next, we want to model an endowment policy. Since the endowment policy pays out a maturity benefit in addition to the death benefits covered by the term life policy, we derive a `Endowment` space from `TermLife`, and make a residual change to the `benefits` formula.

Creating the Life space

Below is a mathematical representation of the life model we'll build as a `Life` space.

$$l(x) = l(x-1) - d(x-1)$$
$$d(x) = l(x) * q$$

where, $l(x)$ denotes the number of lives at age x, $d(x)$ denotes the number of death occurring between the age x and age x + 1, q denotes the annual mortality rate (for simplicity, we'll assume a constant mortality rate of 0.003 for all ages for the moment.) One letter names like l, d, q would be too short for real world practices, but we use them here just for simplicity, as they often appear in classic actuarial textbooks. Yet another simplification is, we set the starting age of x at 50, just to get output shorter. As long as we use a constant mortality age, it shouldn't affect the results whether the starting age is 0 or 50. Below the modelx code for this life model:

```
model, life = new_model(), new_space('Life')

def l(x):
    if x == x0:
        return 100000
    else:
        return l(x - 1) - d(x - 1)

def d(x):
    return l(x) * q

def q():
    return 0.003

l, d, q = defcells(l, d, q)
life.x0 = 50
```

The second to last line of the code above has the same effect as putting `@defcells` decorator on top of each of the 3 function definitions. This line creates 3 new cells from the 3 functions in the `Life` space, and rebind names `l`, `d`, `q` to the 3 cells in the current scope.

You must have noticed that `l(x)` formula is referring to the name `x0`, which is not defined yet. The last line is for defining `x0` as the issue age in the `Life` model and assigning a value to it.

To examine the space, we can check values of the cells in `Life` as below:

```
>>> l(60)
97040.17769489168

>>> life.frame
                    l           d      q
x
 50.0  100000.000000  300.000000    NaN
 51.0   99700.000000  299.100000    NaN
 52.0   99400.900000  298.202700    NaN
 53.0   99102.697300  297.308092    NaN
 54.0   98805.389208  296.416168    NaN
 55.0   98508.973040  295.526919    NaN
 56.0   98213.446121  294.640338    NaN
 57.0   97918.805783  293.756417    NaN
 58.0   97625.049366  292.875148    NaN
 59.0   97332.174218  291.996523    NaN
 60.0   97040.177695         NaN    NaN
NaN              NaN         NaN  0.003
```

Deriving the Term Life space

Next, we'll see how we can extend this space to represent a term life policy. To simplify things, here we focus on one policy with the sum assured of 1 (in whatever unit of currency). With this assumption, if we define `benefits(x)` as the expected value at issue of benefits paid between the age x and x + 1, then it should equate to the probability of death between age x and x + 1, of the insured at the point of issue. In a math expression, this should be written:

$$benefits(x) = d(x)/l(x0)$$

where $l(x)$ and $d(x)$ are the same definition from the preceding example, and $x0$ denotes the issue age of the policy. And further we define the present value of benefits at age x as:

$$pv\_benefits(x) = \sum_{x'=x}^{x0+n} benefits(x')/(1 + disc\_rate)^{x'-x}$$

`n` denotes the policy term in years, and `disc_rate` denotes the discounting rate for the present value calculation.

Continued from the previous code, we are going to derive the `TermLife` space from the `Life` space, to add the benefits and present value calculations.

```
term_life = model.new_space(name='TermLife', bases=life)

@defcells
def benefits(x):
    if x < x0 + n:
        return d(x) / l(x0)
    if x <= x0 + n:
        return 0

@defcells
def pv_benefits(x):
    if x < x0:
        return 0
    elif x <= x0 + n:
        return benefits(x) + pv_benefits(x + 1) / (1 + disc_rate)
    else:
        return 0
```

The first line in the sample above creates `TermLife` space derived from the `Life` space, by passing the `Life` space as `bases` parameter to the `new_space` method of the model. The `TermLife` space at this point has the same cells as its sole base space `Life` space.

The following 2 cells definitions (2 function definitions with `defcells` decorators), are for adding the cells that did not exist in `Life` space. The formulas are referring to the names that are not defined yet. Those are `n`, `disc_rate`. We need to define those in the `TermLife` space. The reference `x0` is inherited from the `Life` space.

```
term_life.n = 10
term_life.disc_rate = 0
```

You get the following results by examining the `TermLife` space (The order of the columns in the DataFrame may be different on your screen).:

```
>>> term_life.pv_benefits(50)
0.02959822305108317

>>> term_life.frame

                d      q              l  pv_benefits  benefits
x
 50.0  300.000000    NaN  100000.000000     0.029598  0.003000
 51.0  299.100000    NaN   99700.000000     0.026598  0.002991
 52.0  298.202700    NaN   99400.900000     0.023607  0.002982
 53.0  297.308092    NaN   99102.697300     0.020625  0.002973
 54.0  296.416168    NaN   98805.389208     0.017652  0.002964
 55.0  295.526919    NaN   98508.973040     0.014688  0.002955
 56.0  294.640338    NaN   98213.446121     0.011733  0.002946
 57.0  293.756417    NaN   97918.805783     0.008786  0.002938
 58.0  292.875148    NaN   97625.049366     0.005849  0.002929
 59.0  291.996523    NaN   97332.174218     0.002920  0.002920
 60.0         NaN    NaN            NaN     0.000000  0.000000
 61.0         NaN    NaN            NaN     0.000000       NaN
NaN           NaN  0.003            NaN          NaN       NaN
```

You can see that the values of `l`, `d`, `q` cells are the same as those in `Life` space, as `Life` and `LifeTerm` have exactly the same formulas for those cells, but be aware that those cells are not shared between the base and derived spaces. Unlike class inheritance in OOP languages, space inheritance is in terms of space instances(or objects), not classes, so cells are copied from the base spaces to derived space upon creating the derived space.

Deriving the Endowment space

We're going to create another space to test overriding inherited cells. We will derive `Endowment` space from `LifeTerm` space. The diagram below shows the relationships of the 3 spaces considered here. A space from which an arrow originates is derived from the space the arrow points to.

The endowment policy pays out the maturity benefit of 1 at the end of its policy term. We have defined `benefits` cells as the expected value of benefits, so in addition to the death benefits considered in `LifeTerm` space, we'll add the maturity benefit by overriding the `benefits` definition in `Endowment` space. In reality, the insured will not get both death and maturity benefits, but here we are considering an probabilistic model, so the benefits would be the sum of expected value of death and maturity benefits:

```
endowment = model.new_space(name='Endowment', bases=term_life)

@defcells
def benefits(x):
```

(continues on next page)

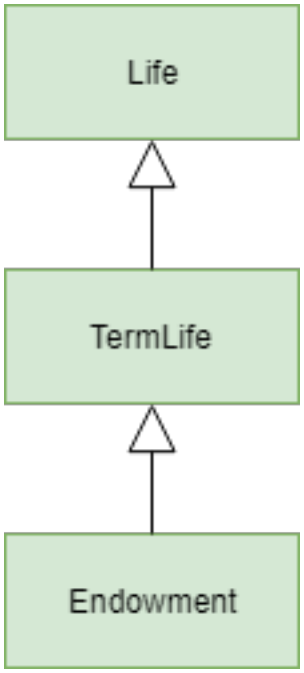


Fig. 3: Life, TermLife and Endowment

(continued from previous page)

```
    if x < x0 + n:
        return d(x) / l(x0)
    elif x == x0 + n:
        return l(x) / l(x0)
    else:
        return 0
```

And the same operations on the `Endowment` space produces the following results:

```
>>> endowment.pv_benefits(50)
1.0
>>> endowment.frame
       pv_benefits  benefits              l      q           d
x
 50.0     1.000000  0.003000  100000.000000    NaN  300.000000
 51.0     0.997000  0.002991   99700.000000    NaN  299.100000
 52.0     0.994009  0.002982   99400.900000    NaN  298.202700
 53.0     0.991027  0.002973   99102.697300    NaN  297.308092
 54.0     0.988054  0.002964   98805.389208    NaN  296.416168
 55.0     0.985090  0.002955   98508.973040    NaN  295.526919
 56.0     0.982134  0.002946   98213.446121    NaN  294.640338
 57.0     0.979188  0.002938   97918.805783    NaN  293.756417
 58.0     0.976250  0.002929   97625.049366    NaN  292.875148
 59.0     0.973322  0.002920   97332.174218    NaN  291.996523
 60.0     0.970402  0.970402   97040.177695    NaN         NaN
 61.0     0.000000       NaN            NaN    NaN         NaN
NaN            NaN       NaN            NaN  0.003         NaN
```

You can see `pv_benefits` for all ages and `benefits` for age 60 show values different from `TermLife` as we overrode `benefits`.

`pv_benefits(50)` being 1 is not surprising. The `disc_rate` set to 1 in `TermLife` space is also inherited to the `Endowment` space. The discounting rate of benefits being 1 means by taking the present value of the benefits, we are simply taking the sum of all expected values of future benefits, which must equates to 1, because the insured gets 1 by 100% chance.

Dynamic spaces

In many situations, you want to apply a set of calculations in a space, or a tree of spaces, to different data sets. You can achieve that by applying the space inheritance on dynamic spaces.

Dynamic spaces are parametrized spaces that are created on-the-fly when requested through call(`()`) or subscript(`[]`) operation on their parent spaces.

To define dynamic spaces in a parent space, you create the space with a parameter function whose signature is used to define space parameters. The parameter function should return, if any, a mapping of parameter names to their arguments, to be pass on to the `new_space` method, when the dynamic spaces are created.

To see how this works, let's continue with the previous example. In the last example, we manually set the issue age `x0` of the policy to 50, and the policy term `n` to 10. We'll extend this example and create policies as dynamic spaces with with different policy attributes. Assume we have 3 term life polices with the following attributes:

Policy ID	Issue Age	Policy Term
1	50	10
2	60	15
3	70	5

We'll create this sample data as a nested list:

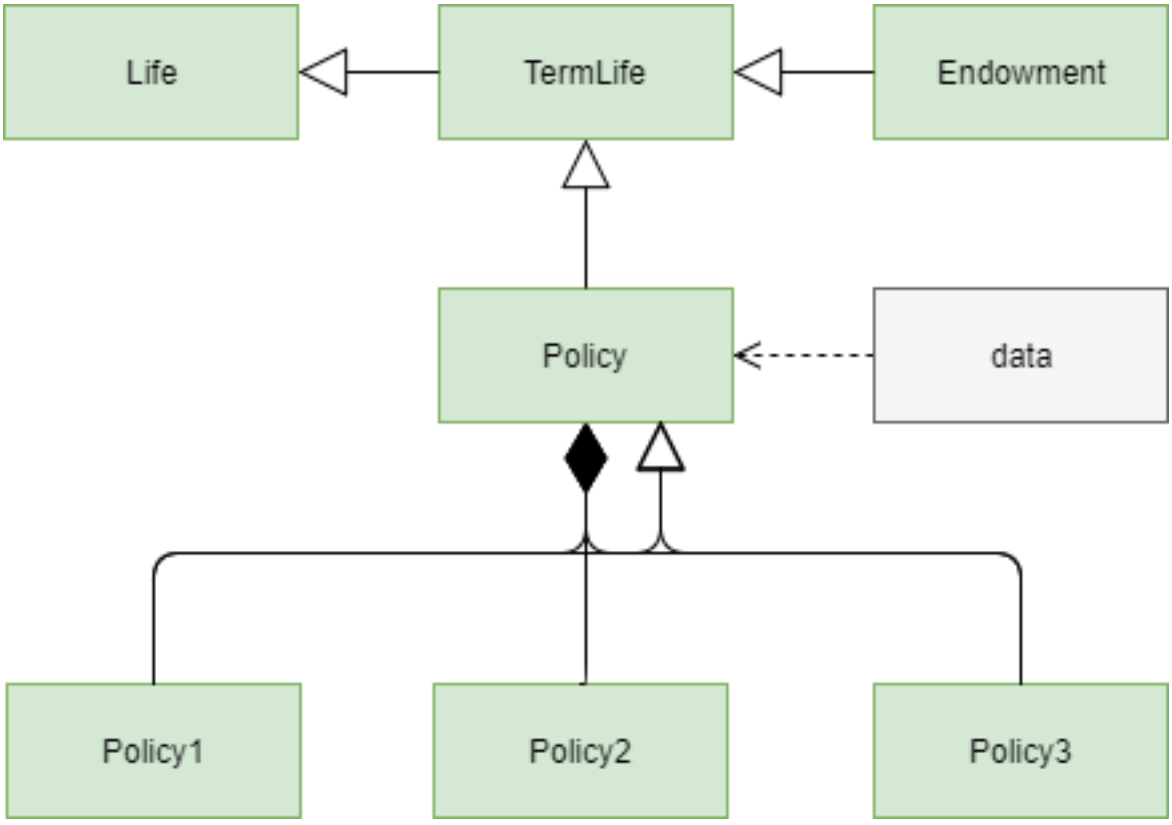
```
data = [[1, 50, 10], [2, 60, 15], [3, 70, 5]]
```

The diagram shows the design of the model we are going to create. The lines with a filled diamond shape on one end indicates that `Policy` model is the parent space of the 3 dynamic spaces, `Policy1`, `Policy2`, `Policy3`, each of which represents each of the 3 policies above. While `Policy` is the parent space of the 3 dynamic space, it is also the base space of them. `Policy` space inherits its members from `Term` model, and in turn `Policy` is inherited by the 3 dynamic spaces. This inheritance is represented by the unfilled arrowhead next the filled diamond.

Below is a script to extend the model as we designed above.

```
def params(policy_id):
    return {'name': 'Policy%s' % policy_id,
            'bases': _self}

policy = model.new_space(name='Policy', bases=term_life, formula=params)

policy.data = data

@defcells
def x0():
    return data[policy_id - 1][1]

@defcells
def n():
    return data[policy_id - 1][2]
```

The `params` function is passed to the constructor of the `Policy` space as the argument of `formula` parameter. The signature of `params` func is used to determine the parameter of the dynamic spaces, and the returned dictionary is



passed to the `new_space` as arguments when the dynamic spaces are created. `params` is called when you create the dynamic subspaces of `Policy`, by calling the n-the element of `Policy`. `params` is evaluated in the `Policy`'s namespace. `_self` is a spacial reference that points to `Policy`.

The parameter `policy_id` becomes available within the namespace of each dynamic space.

In each of the dynamic spaces, the values of `x0` and `n` are taken from `data` for each policy:

```
>>> policy(1).pv_benefits(50)
0.02959822305108317

>>> policy(2).pv_benefits(60)
0.04406717516109439

>>> policy(3).pv_benefits(70)
0.014910269595243001

>>> policy(3).frame
         n    x0           d  benefits              l  pv_benefits      q
x
NaN    5.0  70.0         NaN       NaN            NaN          NaN  0.003
 70.0  NaN   NaN  300.000000  0.003000  100000.000000     0.014910    NaN
 71.0  NaN   NaN  299.100000  0.002991   99700.000000     0.011910    NaN
 72.0  NaN   NaN  298.202700  0.002982   99400.900000     0.008919    NaN
 73.0  NaN   NaN  297.308092  0.002973   99102.697300     0.005937    NaN
 74.0  NaN   NaN  296.416168  0.002964   98805.389208     0.002964    NaN
 75.0  NaN   NaN         NaN  0.000000            NaN     0.000000    NaN
 76.0  NaN   NaN         NaN       NaN            NaN     0.000000    NaN
```

(continues on next page)

(continued from previous page)

```
>>> policy.spaces
{'Policy1': <Space Policy[1] in Model1>,
 'Policy2': <Space Policy[2] in Model1>,
 'Policy3': <Space Policy[3] in Model1>}
```

Dynamic spaces of a base space are not passed on to the derived spaces. When a space is derived from a base space that has dynamically created subspaces, those dynamically created subspaces themselves are not passed on to the derived spaces. Instead, the parameter function of the base space is inherited, so subspaces of the derived space are created upon call(using `()`) or subscript (using `[]`) operators the derived space with arguments.

10.4.6 Reading Excel files

You can read data stored in an Excel file into newly created cells. Space has two methods `new_cells_from_excel` and `new_space_from_excel`. `new_space_from_excel` is also available on Model. You need to have Openpyxl package available in your Python environment to use these methods.

`new_cells_from_excel` method reads values from a range in an Excel file, creates cells and populates them with the values in the range.

`new_space_from_excel` methods reads values from a range in an Excel file, creates a space, and in that space, creates dynamic spaces using one or some of the index rows and/or columns as space parameters, and creates cells in the dynamics spaces populating them with the values in the range.

Refer to the modelx reference for concrete description of those methods.

10.4.7 Exporting to Pandas objects

If you have Pandas installed in your Python environment, you can export values of cells to Pandas' DataFrame or Series objects. Spaces have `frame` property, which generates a DataFrame object whose columns are cells names, and whose indexes are cells parameters. Multiple cells in a space may have different sets of parameters. Generated

10.5 User Guide

Todo: This user guide is now under construction and contains only few sections, which are mostly incomplete at the moment.

10.5.1 Importing Data

modelx allows you to create Spaces or Cells populated with data from data sources.

Contents

Supported data sources

The supported types of data sources are:

- Excel files
- Comma-separated values (CSV) files
- Pandas DataFrame and Series objects

Methods to create Cells from data sources

The following methods of UserSpace create cells from the data sources.

- *UserSpace.new_cells_from_excel()*
- *UserSpace.new_cells_from_csv()*
- *UserSpace.new_cells_from_pandas()*

Methods to create Spaces and Cells from data sources

The methods below create a UserSpace and optionally DynamicSpaces in the Model or UserSpace, and then creates Cells in the Static/Dynamic Spaces with values imported from the data sources.

Model methods

- *Model.new_space_from_excel()*
- *Model.new_space_from_csv()*
- *Model.new_space_from_pandas()*

UserSpace methods

- *UserSpace.new_space_from_excel()*
- *UserSpace.new_space_from_csv()*
- *UserSpace.new_space_from_pandas()*

Saving models with data sources

If the user writes models to files and the model contains Spaces and Cells created from the data source files (Excel or CSV files), those data source files are copied into the model folder/directory. If a model contains Spaces and Cells created from Pandas DataFrame or Series objects, those objects are serialized and saved as binary files in the model folder.

If data source files or objects are modified after the creation of Space and Cells before the model is written to the files, the data sources are saved reflecting the changes. .. See *Saving Model* section for more details.

10.5.2 Saving Model

modelx offers two ways to save models.

Pickling Models

By "Pickling", a model is saved into a single file as a byte stream. It is like taking a snapshot of a model. "Pickling" is meant for a short-term storage.

- *save()*
- *open_model()*

Writing Models

By "Writing", a model is saved into text files under a directory tree.

- *write()*
- *write_model()*
- *read_model()*

10.6 Reference

10.6.1 Function Reference

modelx	Attributes defined in the top level module.
new_model([name])	Create and return a new model.
new_space([name, bases, formula])	Create and return a new space in the current model.
defcells([space, name])	Decorator/function to create cells from Python functions.
get_models()	Returns a dict that maps model names to models.
get_object(name)	Get a modelx object from its full name.
cur_model([model])	Get and/or set the current model.
cur_space([space])	Get and/or set the current space of the current model.
write_model(model, model_path)	Write model to files.
read_model(model_path[, name])	Read model from files.
open_model(path[, name])	Load a model saved from a file and return it.
setup_ipython()	Set up IPython shell for modelx.
restore_ipython()	Restore IPython' default error message.
set_recursion([maxdepth])	Set formula recursion limit.
start_stacktrace()	Activate stack tracing.
stop_stacktrace()	Deactivate stack tracing.
get_stacktrace()	Get stack trace.
clear_stacktrace()	Clear stack trace.

modelx

Attributes defined in the top level module.

modelx.**models**
Alias for *get_models()*. Available for Python 3.7 or newer

Type dict

modelx.new_model

modelx.**new_model**(*name=None*)
Create and return a new model.

The current model is set set to the created model.

Parameters **name** (str, optional) – The name of the model to create. Defaults to ModelN, with N being an automatically assigned integer.

Returns The new model.

modelx.new_space

modelx.**new_space**(*name=None*, *bases=None*, *formula=None*)
Create and return a new space in the current model.

The currentspace of the current model is set to the created model.

Parameters **name** (str, optional) – The name of the space to create. Defaults to SpaceN, with N being an automatically assigned integer.

Returns The new space.

modelx.defcells

modelx.**defcells**(*space=None*, *name=None*, **funcs*)
Decorator/function to create cells from Python functions.

Convenience decorator/function to create new cells directly from function definitions or function objects substituting for calling *new_cells* method of the parent space.

There are 3 ways to use defcells to define cells from functions.

1. As a decorator without arguments

To create a cells from a function definition in the current space of the current model with the same name as the function's:

```
@defcells
def foo(x):
    return x
```

2. As a decorator with arguments

To create a cells from a function definition in a given space and/or with a given name:

```
@defcells(space=space, name=name)
def foo(x):
    return x
```

3. As a function

To create a multiple cells from a multiple function definitions:

```
def foo(x):
    return x

def bar(y):
```

(continues on next page)

(continued from previous page)

```
    return foo(y)

foo, bar = defcells(foo, bar)
```

Parameters

- **space** (*optional*) – For the 2nd usage, a space to create the cells in. Defaults to the current space of the current model.
- **name** (*optional*) – For the 2nd usage, a name of the created cells. Defaults to the function name.
- ***funcs** – For the 3rd usage, function objects. (`space` and `name` also take function objects for the 3rd usage.)

Returns For the 1st and 2nd usage, the newly created single cells is returned. For the 3rd usage, a list of newly created cells are returned.

modelx.get_models

`modelx.`**`get_models`**`()`

Returns a dict that maps model names to models.

From Python 3.7, `models` attribute of `modelx` module is available as an alias for this function.

modelx.get_object

`modelx.`**`get_object`**`(`*name: str*`)`

Get a modelx object from its full name.

modelx.cur_model

`modelx.`**`cur_model`**`(`*model=None*`)`

Get and/or set the current model.

If `model` is given, set the current model to `model` and return it. `model` can be the name of a model object, or a model object itself. If `model` is not given, the current model is returned.

modelx.cur_space

`modelx.`**`cur_space`**`(`*space=None*`)`

Get and/or set the current space of the current model.

If `name` is given, the current space of the current model is set to `name` and return it. If `name` is not given, the current space of the current model is returned.

modelx.write_model

`modelx.`**`write_model`**`(`*model*, *model_path*`)`

Write model to files.

Write `model` to text files in a folder(directory) tree at `model_path`.

Model attributes, such as its name and refs, are output in the file named *_model.py*, directly under *model_path*. For each space in the model, a text file is created with the same name as the space with ".py" extension. The tree structure of the spaces is represented by the tree of folders, i.e. child spaces of a space is stored in a folder named the space.

Generated text files are Python pseudo-scripts, i.e. they are syntactically correct but semantically not-correct Python scripts, that can only be interpreted through *read_model()* function.

Dynamic spaces and cells values are not stored.

For spaces and cells created by *new_space_from_excel()* and *new_cells_from_excel()*, the source Excel files are copied into the same directory where the text files for the spaces the methods are associated with are located. Then when the model is read by *read_model()* function, the methods are invoked to create the spaces or cells.

Method *write()* performs the same operation.

Todo: This function and *write()* method do not work property in combination with *new_cells_from_excel()*, when *new_cells_from_excel()* creates more than one cells at a time.

Parameters

- **model** – Model object to write
- **model_path** (*str*) – Folder path where the model will be output.

modelx.read_model

modelx.**read_model**(*model_path*, *name=None*)

Read model from files.

Read model form a folder(directory) tree `model_path`. The model must be saved by *write_model()* function or *write()* method.

Parameters

- **model_path** (*str*) – A folder(directory) path where model is stored.
- **name** (*str, optional*) – Model name to overwrite the saved name.

Returns A Model object constructed from the files.

modelx.open_model

modelx.**open_model**(*path*, *name=None*)

Load a model saved from a file and return it.

Parameters

- **path** (*str*) – Path to the file to load the model from.
- **name** (*optional*) – If specified, the model is renamed to this name.

Returns A new model created from the file.

modelx.setup_ipython

modelx.**setup_ipython**()
Set up IPython shell for modelx.

Suppress IPython's default traceback messages upon error.

modelx.restore_ipython

modelx.**restore_ipython**()
Restore IPython' default error message.

Bring back IPython's default traceback message upon error for debugging.

modelx.set_recursion

modelx.**set_recursion**(*maxdepth=1000*)
Set formula recursion limit.

Parameters **maxdepth** – The maximum depth of the modelx interpreter stack.

modelx.start_stacktrace

modelx.**start_stacktrace**()
Activate stack tracing.

Start tracing the call stack of formula calculations held internally in modelx.

The tracing is useful when the user wants to get the information on the execution of cells formulas, such as how much time each formula takes from start to finish, or what formulas are called in what order to identify performance bottlenecks.

While the tracing is active, the history of loading and unloading cells and its arguments to/from the call stack is recorded with timestamps and available through calling *get_stacktrace()* function. The tracing continues until the user calls *stop_stacktrace()*.

Up to 10000 records are kept. Exceeding 10000, records are removed from the oldest.

Warning: Activating stack tracing may slow down formula calculations. You should activate it only when needed for inspection purposes.

See also:

stop_stacktrace() *get_stacktrace()* *clear_stacktrace()*

New in version 0.0.25.

modelx.stop_stacktrace

modelx.**stop_stacktrace**()
Deactivate stack tracing.

Stop tracing the call stack of formula calculations started by *start_stacktrace()*. If the tracing is not active, a runtime error is raised.

See also:

start_stacktrace() get_stacktrace() clear_stacktrace()

New in version 0.0.25.

modelx.get_stacktrace

modelx.**get_stacktrace**()

Get stack trace.

Get the call stack trace. The stack trace is a list of tuples each of which represents one of two types of operations, push("ENTER") or pop("EXIT") to/from the call stack. The table blow shows data sored in the tuple elements.

Index	Content
0	"ENTER" or "EXIT"
1	Stack position
2	Time (Seconds elapsed from the epoch)
3	String to represent Cells object
4	A tuple of arguments to the Cells object

The call stack trace must be activated by *start_stacktrace()* to get the trace, otherwise a runtime error is raised,

Returns A list of tuples each of which is a record of stack history.

See also:

start_stacktrace() stop_stacktrace() clear_stacktrace()

New in version 0.0.25.

modelx.clear_stacktrace

modelx.**clear_stacktrace**()

Clear stack trace.

If the tracing is not active, a runtime error is raised.

See also:

start_stacktrace() stop_stacktrace() get_stacktrace()

New in version 0.0.25.

10.6.2 Class Reference

Model	Top-level container in modelx object hierarchy.
UserSpace	Container of cells, other spaces, and cells namespace.
DynamicSpace	Dynamically created space.
Cells	Data container with a formula to calculate its own values.
SpaceView(data)	A mapping of space names to space objects.
CellsView(data[, keys])	A mapping of cells names to cells objects.
CellNode(node)	A combination of a cells, its args and its value.

Model

class modelx.core.model.**Model**

Bases: modelx.core.spacecontainer.EditableSpaceContainer

Top-level container in modelx object hierarchy.

Model instances are the top-level objects and directly contain *UserSpace* objects, which in turn contain other spaces or *Cells* objects.

A model can be created by *new_model* API function.

Methods

close()	Close the model.
cur_space([name])	Set the current space to Space name and return it.
import_module([module, recursive])	Create a child space from an module.
new_space([name, bases, formula, refs])	Create a child space.
new_space_from_csv(filepath[, space, cells, ...])	Create spaces from from a comma-separated values (csv) file.
new_space_from_excel(book, range_[, sheet, ...])	Create a child space from an Excel range.
new_space_from_module(module[, recursive])	Create a child space from an module.
new_space_from_pandas(obj[, space, cells, ...])	Create child spaces from Pandas DataFrame or Series.
rename(name[, rename_old])	Rename the model itself
save(filepath)	Save the model to a file.
set_property(name, value)	Set property name
write(model_path)	Write model to files.

Attributes

allow_none	Whether a cells can have None as its value.
cellgraph	A directed graph of cells.
doc	Description string
fullname	Dotted name of the object.
model	The model this object belongs to.
name	Name of the object.
parent	The parent of this object.
properties	
refs	Return a mapping of global references.
spaces	A mapping of the names of child spaces to the Space objects

rename(*name*, *rename_old=False*)

Rename the model itself

save(*filepath*)

Save the model to a file.

close()

Close the model.

doc
Description string

When models or spaces are imported from modules, taken from modules docstring. For cells, set to its formula's docstring.

write(*model_path*)
Write model to files.

This method performs the *write_model()* on self. See *write_model()* section for the details.

Parameters **model_path** (*str*) – Folder(directory) path where the model is saved.

cellgraph
A directed graph of cells.

refs
Return a mapping of global references.

_baseattrs
A dict of members expressed in literals

_to_attrdict(*attrs=None*)
Get extra attributes

allow_none
Whether a cells can have None as its value.

This is a property of Model, Space and Cells. If `allow_none` of a cells is False, the cells cannot have None as its value. Assigning None to the cells or its formula returning None raises an Error. If True, the cells can have None as their value. If set to None, `allow_none` of its parent is looked up, and the search continues until True or False is found.

Returns True if the cells can have None, False if it cannot, or None if a default value from the parent is to be used.

cur_space(*name=None*)
Set the current space to Space `name` and return it.

If called without arguments, the current space is returned. Otherwise, the current space is set to the space named `name` and the space is returned.

fullname
Dotted name of the object.

Names joined by dots, such as 'Model1.Space1.Cells1', each element in the string is the name of the parent object of the next one joined by a dot.

import_module(*module=None*, *recursive=False*, ***params*)
Create a child space from an module.

Parameters

- **module** – a module object or name of the module object.
- **recursive** – Not yet implemented.
- ****params** – arguments to pass to `new_space`

Returns The new child space created from the module.

model
The model this object belongs to.

This is a property of Model, Space and Cells. For models, this property is themselves.

name
Name of the object.

new_space(*name=None*, *bases=None*, *formula=None*, *refs=None*)
Create a child space.

Parameters

- **name** (*str, optional*) – Name of the space. Defaults to `SpaceN`, where `N` is a number determined automatically.
- **bases** (*optional*) – A space or a sequence of spaces to be the base space(s) of the created space.
- **formula** (*optional*) – Function to specify the parameters of dynamic child spaces. The signature of this function is used for setting parameters for dynamic child spaces. This function should return a mapping of keyword arguments to be passed to this method when the dynamic child spaces are created.

Returns The new child space.

new_space_from_csv(*filepath*, *space=None*, *cells=None*, *param=None*, *space_params=None*, *cells_params=None*, **args*, ***kwargs*)
Create spaces from from a comma-separated values (csv) file.

This method internally calls Pandas read_csv function, and creates cells by passing the returned DataFrame object to *new_space_from_pandas()*. The `filepath` argument to this method is passed to to read_csv as `filepath_or_buffer`, and the user can pass other arguments to read_csv by supplying those arguments to this method as variable-length parameters, `args` and `kargs`.

Parameters

- **filepath** (*str, path object, or file-like object*) – Path to the file.
- **space** – Sequence of strings to set cells name. string is also accepted if read_csv returns a Series because of its `squeeze` parameter set to `True`.
- **cells** – Sequence of strings to overwrite headers for cells names.
- **param** – Sequence of strings to set parameter name(s). A single string can also be passed to set a single parameter name when `frame` has a single level index (i.e. not MultiIndex).
- **space_params** – Sequence of strings or integers to specify space parameters by name or index.
- **cells_params** – Sequence of strings or integers to specify cells parameters by name or index.
- **args** – Any positional arguments to be passed to read_csv.
- **kwargs** – Any keyword arguments to be passed to read_csv.

See also:

`UserSpace.new_cells_from_csv()`: Create Cells from CSV.

new_space_from_excel(*book*, *range_*, *sheet=None*, *name=None*, *names_row=0*, *param_cols=(0,)*, *space_param_order=None*, *cells_param_order=None*, *transpose=False*, *names_col=None*, *param_rows=None*)
Create a child space from an Excel range.

To use this method, `openpyxl` package must be installed.

Parameters

- **book** (*str*) – Path to an Excel file.

- **range** (*str*) – Range expression, such as "A1", "$G4:$K10", or named range "NamedRange1".
- **sheet** (*str*) – Sheet name (case ignored).
- **name** (*str, optional*) – Name of the space. Defaults to `SpaceN`, where `N` is a number determined automatically.
- **names_row** (*optional*) – an index number indicating what row contains the names of cells and parameters. Defaults to the top row (0).
- **param_cols** (*optional*) – a sequence of index numbers indicating parameter columns. Defaults to only the leftmost column ([0]).
- **names_col** (*optional*) – an index number, starting from 0, indicating what column contains additional parameters.
- **param_rows** (*optional*) – a sequence of index numbers, starting from 0, indicating rows of additional parameters, in case cells are defined in two dimensions.
- **transpose** (*optional*) – Defaults to `False`. If set to `True`, "row(s)" and "col(s)" in the parameter names are interpreted inversely, i.e. all indexes passed to "row(s)" parameters are interpreted as column indexes, and all indexes passed to "col(s)" parameters as row indexes.
- **space_param_order** – a sequence to specify space parameters and their orders. The elements of the sequence denote the indexes of `param_cols` elements, and optionally the index of `param_rows` elements shifted by the length of `param_cols`. The elements of this parameter and `cell_param_order` must not overlap.
- **cell_param_order** (*optional*) – a sequence to reorder the parameters. The elements of the sequence denote the indexes of `param_cols` elements, and optionally the index of `param_rows` elements shifted by the length of `param_cols`. The elements of this parameter and `cell_space_order` must not overlap.

Returns The new child space created from the Excel range.

See also:

`UserSpace.new_cells_from_excel()`: Create Cells from Excel file.

new_space_from_module(*module*, *recursive=False*, ***params*)

Create a child space from an module.

Alias to *import_module()*.

Parameters

- **module** – a module object or name of the module object.
- **recursive** – Not yet implemented.
- ****params** – arguments to pass to `new_space`

Returns The new child space created from the module.

new_space_from_pandas(*obj*, *space=None*, *cells=None*, *param=None*, *space_params=None*, *cells_params=None*)

Create child spaces from Pandas DataFrame or Series.

Create a space named `space` and optionally and cells in it from Pandas DataFrame or Series passed in `obj`. If `space` is not given, the space is named `SpaceN` where `N` is automatically given by modelx. Parameter names are taken from `obj` indexes, unless `param` is given to override index names.

`obj` can have MultiIndex as its index. If the index(es) of `obj` has/have name(s), the parameter name(s) of the cells is/are set to the name(s), but can be overwritten by `param` parameter. If the index(es) of `obj` has/have no name(s), and `param` is not given, error is raised.

Parameters

- **obj** – DataFrame or Series.
- **space** – Space name.
- **param** – Sequence of strings to set parameter name(s). A single string can also be passed to set a single parameter name when `frame` has a single level index (i.e. not MultiIndex).
- **space_params** – Sequence of strings or integers to specify space parameters by name or index.
- **cells_params** – Sequence of strings or integers to specify cells parameters by name or index.

See also:

`UserSpace.new_cells_from_pandas()`: Create Cells from DataFrame or Series.

parent
The parent of this object. None for models.

The parent object of a cells is a space that contains the cells. The parent object of a space is either a model or another space that contains the space.

set_property(*name: str*, *value*)
Set property `name`

Set `value` to property `name` of an interface. Equivalent to `x.name = value`, where `x` is a Model/Space/Cells object.

spaces
A mapping of the names of child spaces to the Space objects

UserSpace

class modelx.core.space.**UserSpace**
Bases: `modelx.core.space.BaseSpace`, `modelx.core.spacecontainer.EditableSpaceContainer`

Container of cells, other spaces, and cells namespace.

UserSpace objects can contain cells and other spaces. Spaces have mappings of names to objects that serve as global namespaces of the formulas of the cells in the spaces.

Methods

add_bases(*bases)	Add base spaces.
cur_space([name])	Set the current space to Space `name` and return it.
has_params()	Check if the parameter function is set.
import_funcs(module)	Create a cells from a module.
import_module([module, recursive])	Create a child space from an module.
new_cells([name, formula])	Create a cells in the space.

Continued on next page

Table 5 – continued from previous page

new_cells_from_csv(filepath[, cells, param])	Create cells from a comma-separated values (csv) file.
new_cells_from_excel(book, range_[, sheet, ...])	Create multiple cells from an Excel range.
new_cells_from_module(module)	Create a cells from a module.
new_cells_from_pandas(obj[, cells, param])	Create new cells from Pandas Series or DataFrame object.
new_space([name, bases, formula, refs])	Create a child space.
new_space_from_csv(filepath[, space, cells, ...])	Create spaces from from a comma-separated values (csv) file.
new_space_from_excel(book, range_[, sheet, ...])	Create a child space from an Excel range.
new_space_from_module(module[, recursive])	Create a child space from an module.
new_space_from_pandas(obj[, space, cells, ...])	Create child spaces from Pandas DataFrame or Series.
reload()	Reload the source module and update the formulas.
remove_bases(*bases)	Remove base spaces.
set_formula(formula)	Set if the parameter function.
set_property(name, value)	Set property `name`
to_frame(*args)	Convert the space itself into a Pandas DataFrame object.

Attributes

all_spaces	A mapping associating names to all(static and dynamic) spaces.
allow_none	Whether a cells can have None as its value.
argvalues	A tuple of space arguments.
bases	List of base classes.
cells	A mapping of cells names to the cells objects in the space.
doc	Description string
dynamic_spaces	A mapping associating names to dynamic spaces.
formula	Property to get, set, delete formula.
frame	Alias of `to_frame()`.
fullname	Dotted name of the object.
model	The model this object belongs to.
name	Name of the object.
parameters	A tuple of parameter strings.
parent	The parent of this object.
properties	
refs	A map associating names to objects accessible by the names.
spaces	A mapping associating names to static spaces.
static_spaces	A mapping associating names to static spaces.

new_cells(*name=None*, *formula=None*)

Create a cells in the space.

Parameters

- **name** – If omitted, the model is named automatically `CellsN`, where `N` is an available number.
- **func** – The function to define the formula of the cells.

Returns The new cells.

add_bases(**bases*)
Add base spaces.

remove_bases(**bases*)
Remove base spaces.

import_funcs(*module*)
Create a cells from a module.

new_cells_from_module(*module*)
Create a cells from a module.

Alias to *import_funcs()*.

reload()
Reload the source module and update the formulas.

If the space was created from a module, reload the module and update the formulas of its cells.

If a cell in the space is not created from a function definition in the source module of the space, it is not updated.

If the formula of a cell in the space was created from a function definition in the source module of the space and the definition is missing from the updated module, the formula is cleared and values calculated directly or indirectly depending the cells are cleared.

If the formula of a cell in the space has not been changed before and after reloading the source module, the values held in the cell and relevant cells are retained.

Returns This method returns the space itself.

new_cells_from_excel(*book*, *range_*, *sheet=None*, *names_row=None*, *param_cols=None*, *param_order=None*, *transpose=False*, *names_col=None*, *param_rows=None*)
Create multiple cells from an Excel range.

This method reads values from a range in an Excel file, create cells and populate them with the values in the range. To use this method, `openpyxl` package must be installed.

The Excel file to read data from is specified by `book` parameters. The `range_` can be a range address, such as "$G4:$K10", or a named range. In case a range address is given, `sheet` must also be given.

By default, cells data are interpreted as being laid out side-by-side. `names_row` is a row index (starting from 0) to specify the row that contains the names of cells and parameters. Cells and parameter names must be contained in a single row. `param_cols` accepts a sequence (such as list or tuple) of column indexes (starting from 0) that indicate columns that contain cells arguments.

2-dimensional cells definitions

The optional `names_col` and `param_rows` parameters are used, when data for one cells spans more than one column. In such cases, the cells data is 2-dimensional, and there must be parameter row(s) across the columns that contain arguments of the parameters. A sequence of row indexes that indicate parameter rows is passed to `param_rows`. The names of those parameters must be contained in the same rows as parameter values (arguments), and `names_col` is to indicate the column position at which the parameter names are defined.

Horizontal arrangement

By default, cells data are interpreted as being placed side-by-side, regardless of whether one cells corresponds to a single column or multiple columns. `transpose` parameter is used to alter this orientation, and if it is set to `True`, cells values are interpreted as being placed one above the other. "row(s)" and "col(s)" in the parameter names are interpreted inversely, i.e. all indexes passed to "row(s)" parameters are interpreted as column indexes, and all indexes passed to "col(s)" parameters as row indexes.

Parameters

- **book** (*str*) – Path to an Excel file.
- **range** (*str*) – Range expression, such as "A1", "$G4:$K10", or named range "NamedRange1".
- **sheet** (*str*) – Sheet name (case ignored).
- **names_row** (*optional*) – an index number indicating what row contains the names of cells and parameters. Defaults to the top row (0).
- **param_cols** (*optional*) – a sequence of index numbers indicating parameter columns. Defaults to only the leftmost column ([0]).
- **names_col** (*optional*) – an index number, starting from 0, indicating what column contains additional parameters.
- **param_rows** (*optional*) – a sequence of index numbers, starting from 0, indicating rows of additional parameters, in case cells are defined in two dimensions.
- **transpose** (*optional*) – Defaults to `False`. If set to `True`, "row(s)" and "col(s)" in the parameter names are interpreted inversely, i.e. all indexes passed to "row(s)" parameters are interpreted as column indexes, and all indexes passed to "col(s)" parameters as row indexes.
- **param_order** (*optional*) – a sequence to reorder the parameters. The elements of the sequence are the indexes of `param_cols` elements, and optionally the index of `param_rows` elements shifted by the length of `param_cols`.

See also:

new_space_from_excel(): Create Spaces and Cells from Excel file.

new_cells_from_pandas(*obj*, *cells=None*, *param=None*)

Create new cells from Pandas Series or DataFrame object.

Return new cells created from Pandas Series or DataFrame object passed as `obj`.

`obj` can either be a Series or a DataFrame. If `obj` is a Series, a single cells is created. The cells' name is taken from the Series' name, but can be overwritten if a valid name is passed as `cells`.

If `obj` is a DataFrame, a cells is created for each column. The cells' names can be overwritten by a sequence of valid names passed as `cells`

Keys and values of the cells data are copied from `obj`.

`obj` can have MultiIndex. If the index(es) of `obj` has/have name(s), the parameter name(s) of the cells is/are set to the name(s), but can be overwritten by `param` parameter. If the index(es) of `obj` has/have no name(s), and `param` is not given, error is raised. Error is raised when `obj` has duplicated indexes.

Parameters

- **obj** – Pandas Series or DataFrame object
- **cells** (*str, optional*) – cells name. If `obj` has a valid name and this `cells` is not given, the name is used. If `obj` does not have a name and this `cells` is not given, the cells is named automatically.

- **param** – sequence of strings to set parameter name(s). A single string can also be passed to set a single parameter name when `obj` has a single level index (i.e. not MultiIndex).

Returns New cells if `obj` is a Series, CellsView if `obj` is DataFrame.

See also:

new_space_from_pandas(): Create Spaces and Cells from DataFrame or Series.

new_cells_from_csv(*filepath*, *cells=None*, *param=None*, **args*, ***kwargs*)
Create cells from a comma-separated values (csv) file.

This method internally calls Pandas read_csv function, and creates cells by passing the returned DataFrame object to *new_cells_from_pandas()*. The `filepath` argument to this method is passed to to read_csv as `filepath_or_buffer`, and the user can pass other arguments to read_csv by supplying those arguments to this method as variable-length parameters, `args` and `kargs`.

Parameters

- **filepath** (*str, path object, or file-like object*) – Path to the file.
- **cells** – Sequence of strings to set cells name. string is also accepted if read_csv returns a Series because of its `squeeze` parameter set to `True`.
- **param** – Sequence of strings to set parameter name(s). A single string can also be passed to set a single parameter name when `frame` has a single level index (i.e. not MultiIndex).
- **args** – Any positional arguments to be passed to read_csv.
- **kwargs** – Any keyword arguments to be passed to read_csv.

See also:

new_space_from_csv(): Create Spaces and Cells from CSV.

formula
Property to get, set, delete formula.

set_formula(*formula*)
Set if the parameter function.

doc
Description string

When models or spaces are imported from modules, taken from modules docstring. For cells, set to its formula's docstring.

_baseattrs
A dict of members expressed in literals

_derived_spaces
A mapping associating names to derived spaces.

_direct_bases
Directly inherited base classes

_is_base(*other*)
True if the space is a base space of `other`, False otherwise.

_is_defined()
True if the space is a defined space, False otherwise.

_is_derived()
True if the space is a derived space, False otherwise.

_is_dynamic()
True if the space is in a dynamic space, False otherwise.

_is_root()
True if ths space is a dynamic space, False otherwise.

_is_static()
True if the space is a static space, False if dynamic.

_is_sub(*other*)
True if the space is a sub space of `other`, False otherwise.

_self_refs
A mapping associating names to self refs.

_self_spaces
A mapping associating names to self spaces.

_to_attrdict(*attrs=None*)
Get extra attributes

all_spaces
A mapping associating names to all(static and dynamic) spaces.

allow_none
Whether a cells can have None as its value.

This is a property of Model, Space and Cells. If `allow_none` of a cells is False, the cells cannot have None as its value. Assigning None to the cells or its formula returning None raises an Error. If True, the cells can have None as their value. If set to None, `allow_none` of its parent is looked up, and the search continues until True or False is found.

Returns True if the cells can have None, False if it cannot, or None if a default value from the parent is to be used.

argvalues
A tuple of space arguments.

bases
List of base classes.

cells
A mapping of cells names to the cells objects in the space.

cur_space(*name=None*)
Set the current space to Space `name` and return it.

If called without arguments, the current space is returned. Otherwise, the current space is set to the space named `name` and the space is returned.

dynamic_spaces
A mapping associating names to dynamic spaces.

frame
Alias of `to_frame()`.

fullname
Dotted name of the object.

Names joined by dots, such as 'Model1.Space1.Cells1', each element in the string is the name of the parent object of the next one joined by a dot.

has_params()
Check if the parameter function is set.

import_module(*module=None*, *recursive=False*, ***params*)
Create a child space from an module.

Parameters

- **module** – a module object or name of the module object.
- **recursive** – Not yet implemented.
- ****params** – arguments to pass to `new_space`

Returns The new child space created from the module.

model
The model this object belongs to.

This is a property of Model, Space and Cells. For models, this property is themselves.

name
Name of the object.

new_space(*name=None*, *bases=None*, *formula=None*, *refs=None*)
Create a child space.

Parameters

- **name** (*str, optional*) – Name of the space. Defaults to `SpaceN`, where `N` is a number determined automatically.
- **bases** (*optional*) – A space or a sequence of spaces to be the base space(s) of the created space.
- **formula** (*optional*) – Function to specify the parameters of dynamic child spaces. The signature of this function is used for setting parameters for dynamic child spaces. This function should return a mapping of keyword arguments to be passed to this method when the dynamic child spaces are created.

Returns The new child space.

new_space_from_csv(*filepath*, *space=None*, *cells=None*, *param=None*, *space_params=None*, *cells_params=None*, **args*, ***kwargs*)
Create spaces from from a comma-separated values (csv) file.

This method internally calls Pandas read_csv function, and creates cells by passing the returned DataFrame object to *new_space_from_pandas()*. The `filepath` argument to this method is passed to to read_csv as `filepath_or_buffer`, and the user can pass other arguments to read_csv by supplying those arguments to this method as variable-length parameters, `args` and `kargs`.

Parameters

- **filepath** (*str, path object, or file-like object*) – Path to the file.
- **space** – Sequence of strings to set cells name. string is also accepted if read_csv returns a Series because of its `squeeze` parameter set to `True`.
- **cells** – Sequence of strings to overwrite headers for cells names.
- **param** – Sequence of strings to set parameter name(s). A single string can also be passed to set a single parameter name when `frame` has a single level index (i.e. not MultiIndex).
- **space_params** – Sequence of strings or integers to specify space parameters by name or index.
- **cells_params** – Sequence of strings or integers to specify cells parameters by name or index.

- **args** – Any positional arguments to be passed to read_csv.
- **kwargs** – Any keyword arguments to be passed to read_csv.

See also:

UserSpace.new_cells_from_csv(): Create Cells from CSV.

new_space_from_excel(*book*, *range_*, *sheet=None*, *name=None*, *names_row=0*, *param_cols=(0,)*, *space_param_order=None*, *cells_param_order=None*, *transpose=False*, *names_col=None*, *param_rows=None*)

Create a child space from an Excel range.

To use this method, `openpyxl` package must be installed.

Parameters

- **book** (*str*) – Path to an Excel file.
- **range** (*str*) – Range expression, such as "A1", "$G4:$K10", or named range "NamedRange1".
- **sheet** (*str*) – Sheet name (case ignored).
- **name** (*str, optional*) – Name of the space. Defaults to `SpaceN`, where `N` is a number determined automatically.
- **names_row** (*optional*) – an index number indicating what row contains the names of cells and parameters. Defaults to the top row (0).
- **param_cols** (*optional*) – a sequence of index numbers indicating parameter columns. Defaults to only the leftmost column ([0]).
- **names_col** (*optional*) – an index number, starting from 0, indicating what column contains additional parameters.
- **param_rows** (*optional*) – a sequence of index numbers, starting from 0, indicating rows of additional parameters, in case cells are defined in two dimensions.
- **transpose** (*optional*) – Defaults to `False`. If set to `True`, "row(s)" and "col(s)" in the parameter names are interpreted inversely, i.e. all indexes passed to "row(s)" parameters are interpreted as column indexes, and all indexes passed to "col(s)" parameters as row indexes.
- **space_param_order** – a sequence to specify space parameters and their orders. The elements of the sequence denote the indexes of `param_cols` elements, and optionally the index of `param_rows` elements shifted by the length of `param_cols`. The elements of this parameter and `cell_param_order` must not overlap.
- **cell_param_order** (*optional*) – a sequence to reorder the parameters. The elements of the sequence denote the indexes of `param_cols` elements, and optionally the index of `param_rows` elements shifted by the length of `param_cols`. The elements of this parameter and `cell_space_order` must not overlap.

Returns The new child space created from the Excel range.

See also:

UserSpace.new_cells_from_excel(): Create Cells from Excel file.

new_space_from_module(*module*, *recursive=False*, ***params*)

Create a child space from an module.

Alias to *import_module()*.

Parameters

- **module** – a module object or name of the module object.
- **recursive** – Not yet implemented.
- ****params** – arguments to pass to `new_space`

Returns The new child space created from the module.

new_space_from_pandas(*obj*, *space=None*, *cells=None*, *param=None*, *space_params=None*, *cells_params=None*)

Create child spaces from Pandas DataFrame or Series.

Create a space named `space` and optionally and cells in it from Pandas DataFrame or Series passed in `obj`. If `space` is not given, the space is named `SpaceN` where `N` is automatically given by modelx. Parameter names are taken from `obj` indexes, unless `param` is given to override index names.

`obj` can have MultiIndex as its index. If the index(es) of `obj` has/have name(s), the parameter name(s) of the cells is/are set to the name(s), but can be overwritten by `param` parameter. If the index(es) of `obj` has/have no name(s), and `param` is not given, error is raised.

Parameters

- **obj** – DataFrame or Series.
- **space** – Space name.
- **param** – Sequence of strings to set parameter name(s). A single string can also be passed to set a single parameter name when `frame` has a single level index (i.e. not MultiIndex).
- **space_params** – Sequence of strings or integers to specify space parameters by name or index.
- **cells_params** – Sequence of strings or integers to specify cells parameters by name or index.

See also:

UserSpace.new_cells_from_pandas(): Create Cells from DataFrame or Series.

parameters

A tuple of parameter strings.

parent

The parent of this object. None for models.

The parent object of a cells is a space that contains the cells. The parent object of a space is either a model or another space that contains the space.

refs

A map associating names to objects accessible by the names.

set_property(*name: str*, *value*)

Set property `name`

Set `value` to property `name` of an interface. Equivalent to `x.name = value`, where x is a Model/Space/Cells object.

spaces

A mapping associating names to static spaces.

static_spaces

A mapping associating names to static spaces.

Alias to *spaces()*

to_frame(**args*)
Convert the space itself into a Pandas DataFrame object.

DynamicSpace

class modelx.core.space.**DynamicSpace**
Bases: modelx.core.space.BaseSpace

Dynamically created space.

Dynamic spaces of a parametric space are created by accessing its elements for the first time, through subscription `[]` or call `()` operations on the parametric space.

Dynamic spaces are not editable like static spaces.

Methods

cur_space([name])	Set the current space to Space `name` and return it.
has_params()	Check if the parameter function is set.
set_property(name, value)	Set property `name`
to_frame(*args)	Convert the space itself into a Pandas DataFrame object.

Attributes

all_spaces	A mapping associating names to all(static and dynamic) spaces.
allow_none	Whether a cells can have None as its value.
argvalues	A tuple of space arguments.
bases	List of base classes.
cells	A mapping of cells names to the cells objects in the space.
doc	Description string
dynamic_spaces	A mapping associating names to dynamic spaces.
formula	Property to get, set, delete formula.
frame	Alias of `to_frame()`.
fullname	Dotted name of the object.
model	The model this object belongs to.
name	Name of the object.
parameters	A tuple of parameter strings.
parent	The parent of this object.
properties	
refs	A map associating names to objects accessible by the names.
spaces	A mapping associating names to static spaces.
static_spaces	A mapping associating names to static spaces.

_baseattrs
A dict of members expressed in literals

_derived_spaces
A mapping associating names to derived spaces.

_direct_bases
Directly inherited base classes

_is_base(*other*)
True if the space is a base space of `other`, False otherwise.

_is_defined()
True if the space is a defined space, False otherwise.

_is_derived()
True if the space is a derived space, False otherwise.

_is_dynamic()
True if the space is in a dynamic space, False otherwise.

_is_root()
True if ths space is a dynamic space, False otherwise.

_is_static()
True if the space is a static space, False if dynamic.

_is_sub(*other*)
True if the space is a sub space of `other`, False otherwise.

_self_refs
A mapping associating names to self refs.

_self_spaces
A mapping associating names to self spaces.

_to_attrdict(*attrs=None*)
Get extra attributes

all_spaces
A mapping associating names to all(static and dynamic) spaces.

allow_none
Whether a cells can have None as its value.

This is a property of Model, Space and Cells. If `allow_none` of a cells is False, the cells cannot have None as its value. Assigning None to the cells or its formula returning None raises an Error. If True, the cells can have None as their value. If set to None, `allow_none` of its parent is looked up, and the search continues until True or False is found.

Returns True if the cells can have None, False if it cannot, or None if a default value from the parent is to be used.

argvalues
A tuple of space arguments.

bases
List of base classes.

cells
A mapping of cells names to the cells objects in the space.

cur_space(*name=None*)
Set the current space to Space `name` and return it.

If called without arguments, the current space is returned. Otherwise, the current space is set to the space named `name` and the space is returned.

doc
Description string

When models or spaces are imported from modules, taken from modules docstring. For cells, set to its formula's docstring.

dynamic_spaces
A mapping associating names to dynamic spaces.

formula
Property to get, set, delete formula.

frame
Alias of `to_frame()`.

fullname
Dotted name of the object.

Names joined by dots, such as 'Model1.Space1.Cells1', each element in the string is the name of the parent object of the next one joined by a dot.

has_params()
Check if the parameter function is set.

model
The model this object belongs to.

This is a property of Model, Space and Cells. For models, this property is themselves.

name
Name of the object.

parameters
A tuple of parameter strings.

parent
The parent of this object. None for models.

The parent object of a cells is a space that contains the cells. The parent object of a space is either a model or another space that contains the space.

refs
A map associating names to objects accessible by the names.

set_property(*name: str*, *value*)
Set property `name`

Set `value` to property `name` of an interface. Equivalent to `x.name = value`, where x is a Model/Space/Cells object.

spaces
A mapping associating names to static spaces.

static_spaces
A mapping associating names to static spaces.

Alias to *spaces()*

to_frame(**args*)
Convert the space itself into a Pandas DataFrame object.

Cells

class modelx.core.cells.**Cells**
Bases: modelx.core.base.Interface, collections.abc.Mapping, collections.abc.Callable

Data container with a formula to calculate its own values.

Cells are created by `new_cells` method or its variant methods of the containing space, or by function definitions with `defcells` decorator.

Methods

clear(*args, **kwargs)	Clear all the values.
clear_formula()	Clear the formula.
copy([space, name])	Make a copy of itself and return it.
get(k[,d])	
items()	
keys()	
match(*args, **kwargs)	Returns the best matching args and their value.
node(*args, **kwargs)	Return a *CellNode* object for the given arguments.
preds(*args, **kwargs)	Return a list of predecessors of a cell.
set_formula(func)	Set formula from a function.
set_property(name, value)	Set property `name`
succs(*args, **kwargs)	Return a list of successors of a cell.
to_frame(*args)	Convert the cells itself into a Pandas DataFrame and return it.
to_series(*args)	Convert the cells itself into a Pandas Series and return it.
values()	

Attributes

allow_none	Whether a cells can have None as its value.
doc	Description string
formula	Property to get, set, delete formula.
frame	Alias of `to_frame()`.
fullname	Dotted name of the object.
model	The model this object belongs to.
name	Name of the object.
parameters	A tuple of parameter strings.
parent	The parent of this object.
properties	
series	Alias of `to_series()`.
value	Get, set, delete the scalar value.

match(**args*, ***kwargs*)

Returns the best matching args and their value.

If the cells returns None for the given arguments, continue to get a value by passing arguments masking the given arguments with Nones. The search of non-None value starts from the given arguments to the all None arguments in the lexicographical order. The masked arguments that returns non-None value first is returned with the value.

copy(*space=None*, *name=None*)

Make a copy of itself and return it.

clear(**args*, ***kwargs*)

Clear all the values.

to_series(**args*)
Convert the cells itself into a Pandas Series and return it.

series
Alias of `to_series()`.

to_frame(**args*)
Convert the cells itself into a Pandas DataFrame and return it.

if no *args* are passed, the returned DataFrame contains as many values as the cells have.

if A sequence of arguments to the cells is passed as *args*, the returned DataFrame contains values only for the specified *args*.

Parameters **args** – A sequence or iterable of arguments to the cells.

Returns a DataFrame with a column named after the cells, with indexes named after the parameters of the cells.

frame
Alias of `to_frame()`.

formula
Property to get, set, delete formula.

parameters
A tuple of parameter strings.

set_formula(*func*)
Set formula from a function. Deprecated since version 0.0.5. Use formula property instead.

clear_formula()
Clear the formula. Deprecated since version 0.0.5. Use formula property instead.

value
Get, set, delete the scalar value. The cells must be a scalar cells.

node(**args*, ***kwargs*)
Return a *CellNode* object for the given arguments.

preds(**args*, ***kwargs*)
Return a list of predecessors of a cell.

This method returns a list of CellNode objects, whose elements are predecessors of (i.e. referenced in the formula of) the cell specified by the given arguments.

succs(**args*, ***kwargs*)
Return a list of successors of a cell.

This method returns a list of CellNode objects, whose elements are successors of (i.e. referencing in their formulas) the cell specified by the given arguments.

_baseattrs
A dict of members expressed in literals

_to_attrdict(*attrs=None*)
Get extra attributes

allow_none
Whether a cells can have None as its value.

This is a property of Model, Space and Cells. If `allow_none` of a cells is False, the cells cannot have None as its value. Assigning None to the cells or its formula returning None raises an Error. If True, the

cells can have None as their value. If set to None, `allow_none` of its parent is looked up, and the search continues until True or False is found.

Returns True if the cells can have None, False if it cannot, or None if a default value from the parent is to be used.

doc
Description string

When models or spaces are imported from modules, taken from modules docstring. For cells, set to its formula's docstring.

fullname
Dotted name of the object.

Names joined by dots, such as 'Model1.Space1.Cells1', each element in the string is the name of the parent object of the next one joined by a dot.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

model
The model this object belongs to.

This is a property of Model, Space and Cells. For models, this property is themselves.

name
Name of the object.

parent
The parent of this object. None for models.

The parent object of a cells is a space that contains the cells. The parent object of a space is either a model or another space that contains the space.

set_property(*name: str*, *value*)
Set property `name`

Set `value` to property `name` of an interface. Equivalent to `x.name = value`, where x is a Model/Space/Cells object.

values() → an object providing a view on D's values

SpaceView

class modelx.core.space.**SpaceView**(*data*)
Bases: modelx.core.base.BaseView

A mapping of space names to space objects.

Methods

__init__(data)	Initialize self.
get(k[,d])	
items()	
keys()	

Continued on next page

Table 11 – continued from previous page

values()

_baseattrs
A dict of members expressed in literals

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

values() → an object providing a view on D's values

CellsView

class modelx.core.space.**CellsView**(*data*, *keys=None*)
Bases: modelx.core.base.SelectedView

A mapping of cells names to cells objects.

CellsView objects are returned by *UserSpace.cells* property. When *UserSpace.cells* is called without subscription(`[]` operator), the returned CellsView contains all the cells in the space.

CellsView supports a normal subscription(`[]`) operation with one argument to retrieve a cells object from its name, but it also supports multiple arguments to indicate the names of cells to select, and returns another CellsView containing only the selected cells.

For example, if `space` contains 3 cells `foo`, `bar` and `baz`:

```
>> space.cells
{foo,
 bar,
 baz}

>> space.cells['bar', 'baz']
{bar,
 baz}
```

Methods

__init__(data[, keys])	Initialize self.
get(k[,d])	
items()	
keys()	
to_frame(*args)	Convert the cells in the view into a DataFrame object.
values()	

to_frame(**args*)
Convert the cells in the view into a DataFrame object.

If `args` is not given, this method returns a DataFrame that has an Index or a MultiIndex depending of the number of cells parameters and columns each of which corresponds to each cells included in the view.

`args` can be given to calculate cells values and limit the DataFrame indexes to the given arguments.

The cells in this view may have different number of parameters, but parameters shared among multiple cells must appear in the same position in all the parameter lists. For example, Having `foo()`, `bar(x)` and `baz(x, y=1)` is okay because the shared parameter `x` is always the first parameter, but this method does not work if the view has `quz(x, z=2, y=1)` cells in addition to the first three cells, because `y` appears in different positions.

Parameters **`args`** (*`optional`*) – multiple arguments, or an iterator of arguments to the cells.

`_baseattrs`
A dict of members expressed in literals

`get`(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

`items`() → a set-like object providing a view on D's items

`keys`() → a set-like object providing a view on D's keys

`values`() → an object providing a view on D's values

CellNode

`class` `modelx.core.cells.`**`CellNode`**(*node*)
Bases: `object`

A combination of a cells, its args and its value.

Methods

`__init__`(node)	Initialize self.

Attributes

`args`	Return a tuple of the cells' arguments.
`cells`	Return the Cells object
`has_value`	Return `True` if the cell has a value.
`preds`	A list of nodes that this node refers to.
`succs`	A list of nodes that refer to this node.
`value`	Return the value of the cells.

`cells`
Return the Cells object

`args`
Return a tuple of the cells' arguments.

`has_value`
Return `True` if the cell has a value.

`value`
Return the value of the cells.

`preds`
A list of nodes that this node refers to.

`succs`
A list of nodes that refer to this node.

_baseattrs
A dict of members expressed in literals

10.6.3 Miscellaneous

Technical Guide

modelx embodies software engineering concepts in Object-oriented programming (OOP) paradigm, such as composition and inheritance. However, modelx is not a programming language, rather it is a system of objects. So, some of the OOP terminologies are redefined for the purpose of describing modelx specifications below, although there is strong resemblance. In reading the specifications, readers should be aware that the meanings of the terms in modelx may differ from those in the context of OOP.

Core modelx Types

Model, Space, and Cells are the core modelx types exposed to users for manipulation. Models are the top level objects that contain spaces. Spaces can contain other spaces, so a tree of spaces originating a model can be formed in the model. See '*Space Composition*' section for more details.

Spaces are intermediate objects that come between models and cells in the object composition hierarchy of modelx. Spaces can contain cells, other spaces and references.

A cells can have a formula that calculates the cells' values, just like spreadsheet cells can have formulas. Cells values can be either calculated by the formula or assigned as input.

Space Composition

Parent and Child spaces

As mentioned above, spaces can contain other spaces. A space that contains other spaces is called the *parent* space of the other spaces, and the other spaces are called *child* spaces of the parent space.

By recursively creating child spaces in another space's child spaces, you can create a tree of spaces. A tree of spaces originating a space are called *descendant* spaces of the space. In turn *ascendant* spaces of a space are those that have the space as their descendant space.

Child spaces can not 'outlive' their parent space. In other words, a parent space owns its child spaces i.e. when the parent space is deleted, its child spaces, if there are any, are deleted too.

Note: In OOP contexts, the terms 'parent' and 'child' are sometimes used interchangeably with 'base' and 'sub' respectively. The readers should be aware that here in this reference for modelx, we use the terms 'parent' and 'child' in composition contexts exclusively, and the terms 'sub' and 'base' exclusively in inheritance contexts.

Space Inheritance

Base and Sub spaces

If a space inherits other spaces, the inherited spaces are called *base* spaces of the inheriting space, and the inheriting space is called a *sub* space of the inherited spaces. Multiple inheritance is allowed, i.e. a space can have more than one base spaces.

Since a space that inherits base spaces can in turn be the base space of other spaces, you can create a directional graph of the inheritance relationship. We use the term base spaces of a space to mean not just those spaces that are directly inherited by the space, but also those that are inherited indirectly, through chains of inheritance. In turn, if a space is a base space of another space, then the other space is a sub space of the space either directly or indirectly.

When a space inherits another space that has descendant spaces, descendant spaces are created in the sub space. The descendant spaces in the sub space compose the same tree hierarchy as that in the base space. Each descendant space in the sub space inherits the corresponding space in the space tree of the base space.

A space cannot inherit its descendant spaces. Later in this document, the distinction between static and dynamic spaces is introduced. Static spaces cannot be inherited by their descendant spaces, as that would form circular inheritance. Dynamic spaces can be inherited by their descendant spaces.

Note: The space inheritance concept explained above is analogous to that in OOP. However, the term 'derived' has a special meaning in modelx context. In OOP contexts, 'a class deriving another class' is synonymous with 'the other class inheriting the class'.

Defined and Derived spaces

If a space inherits another space, the child spaces, cells and refs of the base space are derived in the sub space. Furthermore, the descendant spaces and their members of the descendant spaces are derived. The derived spaces have the original corresponding spaces in the base space as their base spaces.

Every static space is either a defined space or a derived space.

Overriding members

Derived cells and refs are overridden when new cells and refs are defined with the same name as the derived cells and refs.

Derived spaces cannot be overridden, however, members of derived spaces can be added or overridden. When members of a derived space are added or overridden, the derived space and its derived ascendant spaces become defined.

Spaces that are directly contained in a model, i.e. spaces that are not child spaces of any other spaces, are always defined spaces.

Static and Dynamic spaces

Every space is either a static space or a dynamic space.

Static spaces are those that are created explicitly by calling their parents' methods, or automatically by the space inheritance mechanism. Spaces that are directly contained in a model, i.e. spaces that are not child spaces of any other spaces, are always static spaces. Since they are always defined, they are always defined and static spaces.

Dynamic spaces, a.k.a parametrized spaces, are those that are created upon the first call or subscription operations on their parent spaces. Such parent spaces must have associated formulas that define parameters of the dynamic spaces and return arguments to be passed to the dynamic spaces for their initialization.

Dynamic spaces can have child spaces just like static spaces, either by calling their methods, or automatically by inheriting base spaces. A dynamic space and its descendants are collectively called a dynamic space tree.

A dynamic space can also have dynamic spaces in its dynamic space tree.

Spaces that are in dynamic space trees cannot be base spaces of other spaces.

Dynamic spaces are not inherited, i.e. if a static ascendant space of dynamic spaces are inherited, no derived spaces in the sub space are created, that correspond to the dynamic spaces in the base space.

Model Structure

There are three types of relationships between spaces, namely composition, inheritance, and dependency. The diagram above only depicts composition and inheritance relationships, but does not show dependency relationships. Spaces that each module depends on are listed in the *ref* section on the page of each source module.

Composition

Lines with the filled diamond arrowheads denote that the spaces on the arrowhead ends contain(and owns) the spaces on the other ends of the lines. In the example diagram below, Space A contains Space B, i.e. Space A is the parent of the Space B, and in tern Space B is a child of Space A. Note that spaces can be directly under their model, in which case the parent of the spaces is the model.

Inheritance

Lines with the hollow triangle arrowheads denote that the spaces on the ends without the arrowheads are derived from the spaces pointed by the arrowheads. In the example diagram below, Space B is derived from Space A, which means copies of all the cells, spaces and refs in Space A are included in Space B.

The Space A above is drawn as a dotted rectangular to indicate that the space acts solely as a base space of others, and it's not meant to be directly accessed by the user.

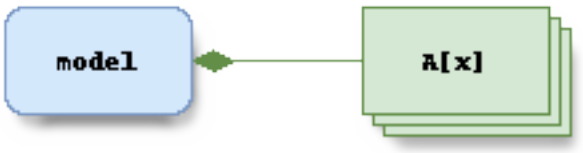
Dependency

When Space B is dependent on Space A, then cells in Space B refer to members of Space A to calculate their values by their formulas. Dependency is not necessarily the relationship between spaces, but it could be the cells

Dynamic Spaces

Dynamic spaces are drawn as a stacked rectangular shape. Dynamic spaces are, in fact, a 'normal' space with its child spaces dynamically created when accessed via subscription(`[]`) or call(`()`) operator. In the example diagram below, Spaces `A[x]` are dynamic spaces. Space A is a normal space and it has a `x` parameter. If A is accessed by, for

example `A[1]`, then a dynamic child space is created under Space A, and in the dynamic child space `A[1]`, Variable `x` is available in the child space and it is set to `1`.



- genindex
- search

Python Module Index

m

Index

Symbols

A

O

P

R

S

T

U

V

W