## Contents

1. **Contents** 3
   1.1 Getting started 3
   1.2 ModelManager API 6
   1.3 Model step template APIs 8
   1.4 Data management templates 20
   1.5 Shared utilities 24
   1.6 Development guide 31

Python Module Index 33

Index 35
UrbanSim Templates provides building blocks for Orca-based simulation models. It’s part of the Urban Data Science Toolkit (UDST).

The library contains templates for common types of model steps, plus a tool called ModelManager that runs as an extension to the Orca task orchestrator. ModelManager can register template-based model steps with the orchestrator, save them to disk, and automatically reload them for future sessions.

v0.2.dev7, released July 22, 2019
1.1 Getting started

1.1.1 Intro

UrbanSim Templates is a Python library that provides building blocks for Orca-based simulation models. It’s part of the Urban Data Science Toolkit (UDST).

The library contains templates for common types of model steps, plus a tool called ModelManager that runs as an extension to the Orca task orchestrator. ModelManager can register template-based model steps with the orchestrator, save them to disk, and automatically reload them for future sessions. The package was developed to make it easier to set up new simulation models — model step templates reduce the need for custom code and make settings more portable between models.

UrbanSim Templates is hosted on Github with a BSD 3-Clause open source license. The code repository includes some material not found in this documentation: a change log, a contributor’s guide, and instructions for running the tests, updating the documentation, and creating a new release.

Another useful resource is the issues and pull requests on Github, which include detailed feature proposals and other discussions.

UrbanSim Templates was created in 2018 by Sam Maurer (maurer@urbansim.com), who remains the lead developer, with contributions from Paul Waddell, Max Gardner, Eddie Janowicz, Arezoo Besharati Zadeh, Xavier Gitiaux, and others.

1.1.2 Installation

UrbanSim Templates is tested with Python versions 2.7, 3.5, 3.6, and 3.7.

As of Feb. 2019, there is an installation problem in Python 3.7 when using Pip (because of an issue with Orca’s PyTables dependency). Conda should work.
**Note:** It can be helpful to set up a dedicated Python environment for each project you work on. This lets you use a stable and replicable set of libraries that won’t be affected by other projects. Here are some good environment settings for UrbanSim Templates projects.

### Production releases

UrbanSim Templates can be installed using the Pip or Conda package managers. With Conda, you (currently) need to install UrbanSim separately; Pip will handle this automatically.

```
pip install urbansim_templates
conda install urbansim_templates --channel conda-forge
conda install urbansim --channel udst
```

Dependencies include NumPy, Pandas, and Statsmodels, plus two other UDST libraries: Orca and ChoiceModels. These will be included automatically when you install UrbanSim Templates.

Certain less-commonly-used templates require additional packages: currently, PyLogit and Scikit-learn. You’ll need to install these manually to use the associated templates.

When new production releases of UrbanSim Templates come out, you can upgrade like this:

```
pip install urbansim_templates --upgrade
conda update urbansim_templates --channel conda-forge
```

### Developer pre-releases

Developer pre-releases of UrbanSim Templates can be installed using the Github URL. These versions sometimes require having a developer release of ChoiceModels as well. Information about the developer releases can be found in Github pull requests.

```
pip install git+git://github.com/udst/choicemodels.git
pip install git+git://github.com/udst/urbansim_templates.git
```

You can use the same command to upgrade.

### Cloning the repository

If you’ll be modifying the code, you can install UrbanSim Templates by cloning the Github repository:

```
git clone https://github.com/udst/urbansim_templates.git
cd urbansim_templates
python setup.py develop
```

Update it with `git pull`.

### 1.1.3 Basic usage
Initializing ModelManager

To get started, import and initialize ModelManager. This makes sure there’s a directory set up to store any template-based model steps that are generated within the script or notebook.

```python
from urbansim_templates import modelmanager
modelmanager.initialize()
```

The default file location is a `configs` folder located in the current working directory; you can provide an alternate path if needed. If ModelManager finds existing saved objects in the directory, it will load them and register them with Orca.

**Note:** It can be helpful to add a cell to your notebook that reports which version of UrbanSim Templates is installed, particularly if you’re using development releases!

```python
In [2]: import urbansim_templates
   print(urbansim_templates.__version__)
Out[2]: '0.2.dev0'
```

Creating a model step

Now we can choose a template and use it to build a model step. The templates are Python classes that contain logic for setting up and running different kinds of model logic — currently focusing on OLS regressions and discrete choice models.

A template takes a variety of arguments, which can either be passed as parameters or set as object properties after an instance of the template is created.

```python
from urbansim_templates.models import OLSRegressionStep
m = OLSRegressionStep()
m.name = 'price-prediction'
m.tables = 'buildings'
m.model_expression = 'sale_price ~ residential_sqft'
```

This sets up `m` as an instance of the OLS regression template. The `tables` and `model_expression` arguments refer to data that needs to be registered separately with Orca. So let’s load the data before trying to estimate the model:

```python
import orca
import pandas as pd
url = 'https://www.dropbox.com/s/vxg5pdfxzrh6osz/buildings-demo.csv?dl=1'
df = pd.read_csv(url).dropna()
orca.add_table('buildings', df)
```

Fitting the statistical model

Now we can fit the building price model:

```python
m.fit()
```
This will print a summary table describing the estimation results.

Now that we have a fitted model, we can use it to predict sale prices for other buildings. UrbanSim forecasting models consist of many interconnected steps like this, iteratively predicting real estate prices, household moves, construction, and other urban dynamics.

### Registering the step

Now we can register the model step:

```python
modelmanager.register(m)
```

ModelManager parses the step, saves a copy to disk, and registers a runnable version of it as a standard Orca step, so that it can be invoked as part of a sequence of other steps:

```python
orca.run(['price-prediction', 'household-moves', 'residential-development'])
```

In real usage, some additional parameters would be set to specify which data to use for prediction, and where to store the output.

### Making changes

ModelManager also includes some interactive functionality. Previously registered steps can be retrieved as template objects, which can be modified and re-registered as needed. This also works with model steps loaded from disk.

```python
modelmanager.list_steps()

m2 = modelmanager.get_step('price-prediction')
...

m2.name = 'better-price-prediction'
modelmanager.register(m2)
modelmanager.remove_step('price-prediction')
```

If you take a look in the `configs` folder, you’ll see a yaml file representing the saved model step. It includes the settings we provided, plus the fitted coefficients and anything else generated by the internal logic of the template.

### 1.2 ModelManager API

ModelManager runs as an extension to the Orca task orchestrator. ModelManager can register template-based model steps with the orchestrator, save them to disk, and automatically reload them for future sessions.

The recommended way to load ModelManager is like this:

```python
from urbansim_templates import modelmanager
modelmanager.initialize()
```

#### 1.2.1 Core operations

```python
urbansim_templates.modelmanager.initialize(path='configs')
```

Load saved model steps from disk. Each file in the directory will be checked for compliance with the ModelManager YAML format and then loaded into memory.
If run multiple times, steps will be cleared from memory and re-loaded.

**Parameters**
- **path** *(str)* – Path to config directory, either absolute or relative to the Python working directory

**urbansim_templates.modelmanager.register** *(step, save_to_disk=True)*

Register a model step with ModelManager and Orca. This includes saving it to disk, optionally, so it can be automatically loaded in the future.

Registering a step will overwrite any previously loaded step with the same name. If a name has not yet been assigned, one will be generated from the template name and a timestamp.

If the model step includes an attribute ‘autorun’ that’s set to True, the step will run after being registered.

**Parameters**
- **step** *(object)* –

**Returns** None

**urbansim_templates.modelmanager.list_steps** *

Return a list of registered steps, with name, template, and tags for each.

**Returns** list of dicts, ordered by name

**urbansim_templates.modelmanager.get_step** *(name)*

Return the class representation of a registered step, by name.

**Parameters**
- **name** *(str)* –

**Returns** instance of a template class

**urbansim_templates.modelmanager.remove_step** *(name)*

Remove a model step, by name. It will immediately be removed from ModelManager and from disk, but will remain registered in Orca until the current Python process terminates.

**Parameters**
- **name** *(str)* –

### 1.2.2 Internal functionality

These functions are the building blocks of ModelManager. You probably won’t need to use them directly, but they could be useful for debugging or for extending ModelManager’s functionality.

**urbansim_templates.modelmanager.template** *(cls)*

This is a decorator for ModelManager-compliant template classes. Place `@modelmanager.template` on the line before a class definition.

This makes the class available to ModelManager (e.g. for reading saved steps from disk) whenever it’s imported.

**urbansim_templates.modelmanager.build_step** *(d)*

Build a model step object from a saved dictionary. This includes loading supplemental objects from disk.

**Parameters**
- **d** *(dict)* – Representation of a model step.

**Returns**

**Return type** object

**urbansim_templates.modelmanager.save_step_to_disk** *(step)*

Save a model step to disk, over-writing the previous file. The file will be named ‘model-name.yaml’ and will be saved to the initialization directory.
urbansim_templates.modelmanager.load_supplemental_object(step_name, name, content_type, required=True)

Load a supplemental object from disk.

Parameters

- **step_name** (str) – Name of the associated model step.
- **name** (str) – Name of the supplemental object.
- **content_type** (str) – Currently supports ‘pickle’.
- **required** (bool, optional) – Whether the supplemental object is required (not yet supported).

Returns

Return type: object

urbansim_templates.modelmanager.save_supplemental_object(step_name, name, content, content_type, required=True)

Save a supplemental object to disk.

Parameters

- **step_name** (str) – Name of the associated model step.
- **name** (str) – Name of the supplemental object.
- **content** (obj) – Object to save.
- **content_type** (str) – Currently supports ‘pickle’.
- **required** (bool, optional) – Whether the supplemental object is required (not yet supported).

urbansim_templates.modelmanager.remove_supplemental_object(step_name, name, content_type)

Remove a supplemental object from disk.

Parameters

- **step_name** (str) – Name of the associated model step.
- **name** (str) – Name of the supplemental object.
- **content_type** (str) – Currently supports ‘pickle’.

urbansim_templates.modelmanager.get_config_dir()

Return the config directory, for other services that need to interoperate.

Returns

Return type: str

### 1.3 Model step template APIs

The following templates are included in the core package. ModelManager can also work with templates defined elsewhere, as long as they follow the specifications described in the design guidelines.
1.3.1 OLS Regression

class urbansim_templates.models.OLSRegressionStep(tables=None,
model_expression=None,
filters=None,
out_tables=None,
out_column=None,
out_transform=None,
out_filters=None,
nname=None,
tags=[])

A class for building OLS (ordinary least squares) regression model steps. This extends TemplateStep, where some common functionality is defined. Estimation and simulation are handled by urbansim.models.RegressionModel().

Expected usage: - create a model object - specify some parameters - run the fit() method - iterate as needed

Then, for simulation: - specify some simulation parameters - use the run() method for interactive testing - use modelmanager.register() to save the model to Orca and disk - registered steps can be accessed via ModelManager and Orca

All parameters listed in the constructor can be set directly on the class object, at any time.

Parameters

• tables (str or list of str, optional) – Name(s) of Orca tables to draw data from. The first table is the primary one. Any additional tables need to have merge relationships (“broadcasts”) specified so that they can be merged unambiguously onto the first table. Among them, the tables must contain all variables used in the model expression and filters. The left-hand-side variable should be in the primary table. The tables parameter is required for fitting a model, but it does not have to be provided when the object is created.

• model_expression (str, optional) – Patsy formula containing both the left- and right-hand sides of the model expression: http://patsy.readthedocs.io/en/latest/formulas.html This parameter is required for fitting a model, but it does not have to be provided when the object is created.

• filters (str or list of str, optional) – Filters to apply to the data before fitting the model. These are passed to pd.DataFrame.query(). Filters are applied after any additional tables are merged onto the primary one. Replaces the fit_filters argument in UrbanSim.

• out_tables (str or list of str, optional) – Name(s) of Orca tables to use for simulation. If not provided, the tables parameter will be used. Same guidance applies: the tables must be able to be merged unambiguously, and must include all columns used in the right-hand-side of the model expression and in the out_filters.

• out_column (str, optional) – Name of the column to write predicted values to. If it does not already exist in the primary output table, it will be created. If not provided, the left-hand-side variable from the model expression will be used. Replaces the out_fname argument in UrbanSim.

• out_transform (str, optional) – Element-wise transformation to apply to the predicted values, for example to reverse a transformation of the left-hand-side variable in the model expression. This should be provided as a string containing a function name. Supports anything from NumPy or Python’s built-in math library, for example ‘np.exp’ or ‘math.floor’. Replaces the ytransform argument in UrbanSim.

• out_filters (str or list of str, optional) – Filters to apply to the data before simulation. If not provided, no filters will be applied. Replaces the predict_filters argument in UrbanSim.
• **name** *(str, optional)* – Name of the model step, passed to ModelManager. If none is provided, a name is generated each time the `fit()` method runs.

• **tags** *(list of str, optional)* – Tags, passed to ModelManager.

`fit()`

Fit the model; save and report results.

This currently uses the `RegressionModel` class from core UrbanSim. We save the model object for prediction and interactive use (`model`, with type `urbansim.models.regression.RegressionModel`).

For example, you can use this to get a latex version of the summary table using `m.model.model_fit.summary().as_latex()`. This may change in the future if we refactor the template to use StatsModels directly.

`classmethod from_dict(d)`

Create an object instance from a saved dictionary representation.

Parameters

- **d** *(dict)* –

Returns

- **Return type** `OLSRegressionStep`

`run()`

Run the model step: calculate predicted values, transform them as specified, and use them to update a column.

The pre-transformation predicted values are saved to the class object for diagnostic use (`predicted_values` with type `pd.Series`). The post-transformation predicted values are written to Orca.

`to_dict()`

Create a dictionary representation of the object.

Returns

- **Return type** `dict`

### 1.3.2 Binary Logit

**class urbansim_templates.models.BinaryLogitStep** *(tables=None,
model_expression=None,
filters=None,
out_tables=None,
out_column=None,
out_filters=None,
out_value_true=1,
out_value_false=0,
name=None, tags=[])*

A class for building binary logit model steps. This extends TemplateStep, where some common functionality is defined. Estimation is handled by Statsmodels and simulation is handled within this class.

Expected usage: - create a model object - specify some parameters - run the `fit()` method - iterate as needed

Then, for simulation: - specify some simulation parameters - use the `run()` method for interactive testing - use `modelmanager.register()` to save the model to Orca and disk - registered steps can be accessed via ModelManager and Orca

All parameters listed in the constructor can be set directly on the class object, at any time.

Parameters

• **tables** *(str or list of str, optional)* – Name(s) of Orca tables to draw data from. The first table is the primary one. Any additional tables need to have merge relationships (“broadcasts”) specified so that they can be merged unambiguously onto the first table.
Among them, the tables must contain all variables used in the model expression and filters. The left-hand-side variable should be in the primary table. The \textit{tables} parameter is required for fitting a model, but it does not have to be provided when the object is created.

- \textbf{model\_expression}(str, \textit{optional}) – Patsy formula containing both the left- and right-hand sides of the model expression: \url{http://patsy.readthedocs.io/en/latest/formulas.html} This parameter is required for fitting a model, but it does not have to be provided when the object is created.

- \textbf{filters}(str or list of str, \textit{optional}) – Filters to apply to the data before fitting the model. These are passed to \textit{pd.DataFrame.query()}. Filters are applied after any additional tables are merged onto the primary one. Replaces the \textit{fit\_filters} argument in UrbanSim.

- \textbf{out\_tables}(str or list of str, \textit{optional}) – Name(s) of Orca tables to use for simulation. If not provided, the \textit{tables} parameter will be used. Same guidance applies: the tables must be able to be merged unambiguously, and must include all columns used in the right-hand-side of the model expression and in the \textit{out\_filters}.

- \textbf{out\_column}(str, \textit{optional}) – Name of the column to write simulated choices to. If it does not already exist in the primary output table, it will be created. If not provided, the left-hand-side variable from the model expression will be used. Replaces the \textit{out\_fname} argument in UrbanSim.

# TO DO - auto-generation not yet working; column must exist in the primary table

- \textbf{out\_filters}(str or list of str, \textit{optional}) – Filters to apply to the data before simulation. If not provided, no filters will be applied. Replaces the \textit{predict\_filters} argument in UrbanSim.

- \textbf{out\_value\_true}(numeric or str, \textit{optional}) – Value to save to the output column corresponding to an affirmative choice. Default is 1 (int). Use keyword ‘nothing’ to leave values unchanged.

- \textbf{out\_value\_false}(numeric or str, \textit{optional}) – Value to save to the output column corresponding to a negative choice. Default is 0 (int). Use keyword ‘nothing’ to leave values unchanged.

- \textbf{name}(str, \textit{optional}) – Name of the model step, passed to ModelManager. If none is provided, a name is generated each time the \textit{fit()} method runs.

- \textbf{tags}(list of str, \textit{optional}) – Tags, passed to ModelManager.

\textbf{fit()}

Fit the model; save and report results. This currently uses the Statsmodels Logit class with default estimation settings. (It will shift to ChoiceModels once more infrastructure is in place.)

The \textit{fit()} method can be run as many times as desired. Results will not be saved with Orca or ModelManager until the \textit{register()} method is run.

\textbf{Parameters} None –

\textbf{Returns}

\textbf{Return type} None

\textbf{classmethod from\_dict}(d)

Create an object instance from a saved dictionary representation.

\textbf{Parameters} \textit{d}(dict) –

\textbf{Returns}
Return type  *BinaryLogitStep*

**run**

Run the model step: calculate simulated choices and use them to update a column.

For binary logit, we calculate predicted probabilities and then perform a weighted random draw to determine the simulated binary outcomes. This is done directly from the fitted parameters, because we can’t conveniently regenerate a Statsmodels results object from a dictionary representation.

The predicted probabilities and simulated choices are saved to the class object for interactive use (*probabilities* and *choices*, with type pd.Series) but are not persisted in the dictionary representation of the model step.

**Parameters**  
None

**Returns**  
Return type  None

**to_dict**

Create a dictionary representation of the object.

**Returns**  
Return type  dict

### 1.3.3 Small Multinomial Logit

**class urbansim_templates.models.SmallMultinomialLogitStep** ([tables=None](#),  
*model_expression=None*,  
*model_labels=None*,  
*choice_column=None*,  
*initial_coefs=None*,  
*filters=None*,  
*out_tables=None*,  
*out_column=None*,  
*out_filters=None*,  
*name=None, tags=[])*

A class for building multinomial logit model steps where the number of alternatives is “small”. Estimation is handled by PyLogit via the ChoiceModels API. Simulation is handled by PyLogit (probabilities) and ChoiceModels (simulation draws).

Multinomial logit models can involve a range of different specification and estimation mechanics. For now these are separated into two templates. What’s the difference?

“Small” MNL: - data is in a single table (choosers) - each alternative can have a different model expression - all the alternatives are available to all choosers - estimation and simulation use the PyLogit engine (via ChoiceModels)

“Large” MNL: - data is in two tables (choosers and alternatives) - each alternative has the same model expression - N alternatives are sampled for each chooser - estimation and simulation use the ChoiceModels engine (formerly UrbanSim MNL)

TO DO: - Add support for specifying availability of alternatives - Add support for sampling weights - Add support for on-the-fly interaction calculations (e.g. distance)

**Parameters**

- **tables**(str or list of str, optional) – Name(s) of Orca tables to draw data from. The first table is the primary one. Any additional tables need to have merge relationships (“broadcasts”) specified so that they can be merged unambiguously onto the first table.
Among them, the tables must contain all variables used in the model expression and filters. The index of the primary table should be a unique ID. The `tables` parameter is required for fitting a model, but it does not have to be provided when the object is created. Reserved column names: `_obs_id`, `_alt_id`, `_chosen`.

- **model_expression** *(OrderedDict, optional)* – PyLogit model expression. This parameter is required for fitting a model, but it does not have to be provided when the object is created.

- **model_labels** *(OrderedDict, optional)* – PyLogit model labels.

- **choice_column** *(str, optional)* – Name of the column indicating observed choices, for model estimation. The column should contain integers matching the alternatives in the model expression. This parameter is required for fitting a model, but it does not have to be provided when the object is created.

- **initial_coefs** *(list of numerics, optional)* – Starting values for the parameter estimation algorithm, passed to PyLogit. Length must be equal to the number of parameters being estimated. If this is not provided, zeros will be used.

- **filters** *(str or list of str, optional)* – Filters to apply to the data before fitting the model. These are passed to `pd.DataFrame.query()`. Filters are applied after any additional tables are merged onto the primary one. Replaces the `fit_filters` argument in UrbanSim.

- **out_tables** *(str or list of str, optional)* – Name(s) of Orca tables to use for simulation. If not provided, the `tables` parameter will be used. Same guidance applies: the tables must be able to be merged unambiguously, and must include all columns used in the model expression and in the `out_filters`.

- **out_column** *(str, optional)* – Name of the column to write simulated choices to. If it does not already exist in the primary output table, it will be created. If not provided, the `choice_column` will be used. Replaces the `out_fname` argument in UrbanSim.

- **out_filters** *(str or list of str, optional)* – Filters to apply to the data before simulation. If not provided, no filters will be applied. Replaces the `predict_filters` argument in UrbanSim.

- **name** *(str, optional)* – Name of the model step, passed to ModelManager. If none is provided, a name is generated each time the `fit()` method runs.

- **tags** *(list of str, optional)* – Tags, passed to ModelManager.

**fit()**

Fit the model; save and report results. This uses PyLogit via ChoiceModels.

The `fit()` method can be run as many times as desired. Results will not be saved with Orca or ModelManager until the `register()` method is run.

**classmethod from_dict(d)**

Create an object instance from a saved dictionary representation.

**Parameters**

- **d** *(dict)*

**Returns**

**Return type** *SmallMultinomialLogitStep*

**run()**

Run the model step: calculate simulated choices and use them to update a column.

Alternatives that appear in the estimation data but not in the model expression will not be available for simulation.
Predicted probabilities come from PyLogit. Monte Carlo simulation of choices is performed directly. (This functionality will move to ChoiceModels.)

The predicted probabilities and simulated choices are saved to the class object for interactive use (probabilities with type pd.DataFrame, and choices with type pd.Series) but are not persisted in the dictionary representation of the model step.

to_dict()
Create a dictionary representation of the object.

Returns
Return type dict

1.3.4 Large Multinomial Logit

class urbansim_templates.models.LargeMultinomialLogitStep(choosers=None, alternatives=None, model_expression=None, choice_column=None, chooser_filters=None, chooser_sample_size=None, alt_filters=None, alt_sample_size=None, out_choosers=None, out_alternatives=None, out_column=None, out_chooser_filters=None, out_alt_filters=None, constrained_choices=False, alt_capacity=None, chooser_size=None, max_iter=None, name=None, tags=[])  

Class for building standard multinomial logit model steps where alternatives are interchangeable and all have the same model expression. Supports random sampling of alternatives.

Estimation and simulation are performed using ChoiceModels.

Parameters

- **choosers** (str or list of str, optional) – Name(s) of Orca tables to draw choice scenario data from. The first table is the primary one. Any additional tables need to have merge relationships (“broadcasts”) specified so that they can be merged unambiguously onto the first table. The index of the primary table should be a unique ID. In this template, the ‘choosers’ and ‘alternatives’ parameters replace the ‘tables’ parameter. Both are required for fitting a model, but do not have to be provided when the object is created. Reserved column names: ‘chosen’.

- **alternatives** (str or list of str, optional) – Name(s) of Orca tables containing data about alternatives. The first table is the primary one. Any additional tables need to have merge relationships (“broadcasts”) specified so that they can be merged unambiguously onto the first table. The index of the primary table should be a unique ID. In this template, the ‘choosers’ and ‘alternatives’ parameters replace the ‘tables’ parameter. Both are required for fitting a model, but do not have to be provided when the object is created. Reserved column names: ‘chosen’.
• **model_expression** *(str, optional)* – Patsy-style right-hand-side model expression representing the utility of a single alternative. Passed to `choicemodels.MultinomialLogit()`. This parameter is required for fitting a model, but does not have to be provided when the object is created.

• **choice_column** *(str, optional)* – Name of the column indicating observed choices, for model estimation. The column should contain integers matching the id of the primary `alternatives` table. This parameter is required for fitting a model, but it does not have to be provided when the object is created. Not required for simulation.

• **chooser_filters** *(str or list of str, optional)* – Filters to apply to the chooser data before fitting the model. These are passed to `pd.DataFrame.query()`. Filters are applied after any additional tables are merged onto the primary one. Replaces the `fit_filters` argument in UrbanSim.

• **chooser_sample_size** *(int, optional)* – Number of choosers to sample, for faster model fitting. Sampling is random and may vary between model runs.

• **alt_filters** *(str or list of str, optional)* – Filters to apply to the alternatives data before fitting the model. These are passed to `pd.DataFrame.query()`. Filters are applied after any additional tables are merged onto the primary one. Replaces the `fit_filters` argument in UrbanSim. Choosers whose chosen alternative is removed by these filters will not be included in the model estimation.

• **alt_sample_size** *(int, optional)* – Number of alternatives to sample for each choice scenario. For now, only random sampling is supported. If this parameter is not provided, we will use a sample size of one less than the total number of alternatives. (Choice-Models codebase currently requires sampling.) The same sample size is used for estimation and prediction.

• **out_choosers** *(str or list of str, optional)* – Name(s) of Orca tables to draw choice scenario data from, for simulation. If not provided, the `choosers` parameter will be used. Same guidance applies. Reserved column names: ‘chosen’, ‘join_index’, ‘observation_id’.

• **out_alternatives** *(str or list of str, optional)* – Name(s) of Orca tables containing data about alternatives, for simulation. If not provided, the `alternatives` parameter will be used. Same guidance applies. Reserved column names: ‘chosen’, ‘join_index’, ‘observation_id’.

• **out_column** *(str, optional)* – Name of the column to write simulated choices to. If it does not already exist in the primary `out_choosers` table, it will be created. If not provided, the `choice_column` will be used. If the column already exists, choices will be cast to match its data type. If the column is generated on the fly, it will be given the same data type as the index of the alternatives table. Replaces the `out_fname` argument in UrbanSim.

• **out_chooser_filters** *(str or list of str, optional)* – Filters to apply to the chooser data before simulation. If not provided, no filters will be applied. Replaces the `predict_filters` argument in UrbanSim.

• **out_alt_filters** *(str or list of str, optional)* – Filters to apply to the alternatives data before simulation. If not provided, no filters will be applied. Replaces the `predict_filters` argument in UrbanSim.

• **constrained_choices** *(bool, optional)* – “True” means alternatives have limited capacity. “False” (default) means that alternatives can accommodate an unlimited number of choosers.

• **alt_capacity** *(str, optional)* – Name of a column in the `out_alternatives` table that expresses the capacity of alternatives. If not provided and `constrained_choices` is True,
each alternative is interpreted as accommodating a single chooser.

- **chooser_size** (str, optional) – Name of a column in the out_choosers table that expresses the size of choosers. Choosers might have varying sizes if the alternative capacities are amounts rather than counts – e.g. square footage. Chooser sizes must be in the same units as alternative capacities. If not provided and constrained_choices is True, each chooser has a size of 1.

- **max_iter** (int or None, optional) – Maximum number of choice simulation iterations. If None (default), the algorithm will iterate until all choosers are matched or no alternatives remain.

- **name** (str, optional) – Name of the model step, passed to ModelManager. If none is provided, a name is generated each time the \texttt{fit()} method runs.

- **tags** (list of str, optional) – Tags, passed to ModelManager.

\textbf{All parameters can also be get and set as properties. The following attributes should be treated as read-only.}

**choices**

Available after the model step is run. List of chosen alternative id’s, indexed with the chooser id. Does not persist when the model step is reloaded from storage.

\hspace{1cm} Type \texttt{pd.Series}

**mergedchoicetable**

Table built for estimation or simulation. Does not persist when the model step is reloaded from storage. Not available if choices have capacity constraints, because multiple choice tables are generated iteratively.

\hspace{1cm} Type \texttt{choicemodels.tools.MergedChoiceTable}

**model**

Available after a model has been fit. Persists when reloaded from storage.

\hspace{1cm} Type \texttt{choicemodels.MultinomialLogitResults}

**probabilities**

Available after the model step is run – but not if choices have capacity constraints, which requires probabilities to be calculated multiple times. Provides list of probabilities corresponding to the sampled alternatives, indexed with the chooser and alternative id’s. Does not persist when the model step is reloaded from storage.

\hspace{1cm} Type \texttt{pd.Series}

**fit** (*mct=None*)

Fit the model; save and report results. This uses the ChoiceModels estimation engine (originally from UrbanSim MNL).

The \texttt{fit()} method can be run as many times as desired. Results will not be saved with Orca or ModelManager until the \texttt{register()} method is run.

After sampling alternatives for each chooser, the merged choice table is saved to the class object for diagnostic use (\texttt{mergedchoicetable} with type \texttt{choicemodels.tools.MergedChoiceTable}).

\textbf{Parameters} \texttt{mct} (\texttt{choicemodels.tools.MergedChoiceTable}) – This parameter is a temporary backdoor allowing us to pass in a more complicated choice table than can be generated within the template, for example including sampling weights or interaction terms.

\textbf{Returns}

\hspace{1cm} Return type None
classmethod from_dict(d)
Create an object instance from a saved dictionary representation.

Parameters d (dict) –

Returns

Return type LargeMultinomialLogitStep
un(chooser_batch_size=None, interaction_terms=None)
Run the model step: simulate choices and use them to update an Orca column.

The simulated choices are saved to the class object for diagnostics. If choices are unconstrained, the choice table and the probabilities of sampled alternatives are saved as well.

Parameters

• chooser_batch_size (int) – This parameter gets passed to choicemodels.tools.simulation.iterative_lottery_choices and is a temporary workaround for dealing with memory issues that arise from generating massive merged choice tables for simulations that involve large numbers of choosers, large numbers of alternatives, and large numbers of predictors. It allows the user to specify a batch size for simulating choices one chunk at a time.

• interaction_terms (pandas.Series, pandas.DataFrame, or list of either, optional) – Additional column(s) of interaction terms whose values depend on the combination of observation and alternative, to be merged onto the final data table. If passed as a Series or DataFrame, it should include a two-level MultiIndex. One level’s name and values should match an index or column from the observations table, and the other should match an index or column from the alternatives table.

Returns

Return type None

to_dict()
Create a dictionary representation of the object.

Returns

Return type dict

1.3.5 Segmented Large Multinomial Logit

class urbansim_templates.models.SegmentedLargeMultinomialLogitStep (defaults=None, segmentation_column=None, name=None, tags=[]) 

This template automatically generates a set of LargeMultinomialLogitStep submodels corresponding to “segments” or categories of choosers. The submodels can be directly accessed and edited.

Running ‘build_submodels()’ will create a submodel for each category of choosers identified in the segmentation column. The submodels are implemented using filter queries.

Once they are generated, the ‘submodels’ property contains a dict of LargeMultinomialLogitStep objects, identified by category name. You can edit their properties as needed, fit them individually, etc.

Editing a property in the ‘defaults’ object will update all the submodels at once, while leaving customizations to other properties intact.

Parameters
• **defaults** *(LargeMultinomialLogitStep, optional)* – Object containing initial parameter values for the submodels. Values for ‘choosers’, ‘alternatives’, and ‘choice_column’ are required to generate submodels, but do not have to be provided when the object is created.

• **segmentation_column** *(str, optional)* – Name of a column of categorical values in the ‘defaults.choosers’ table. Any data that can be interpreted by Pandas as categorical is valid. This is required to generate submodels, but does not have to be provided when the object is created.

• **name** *(str, optional)* – Name of the model step.

• **tags** *(list of str, optional)* – Tags associated with the model step.

`build_submodels()`
Create a submodel for each category of choosers identified in the segmentation column. Only categories with at least one observation remaining after applying chooser and alternative filters will be included.

Running this method will overwrite any previous submodels.

`fit_all(mct=None)`
Fit all the submodels. Build the submodels first, if they don’t exist yet. This method can be run as many times as desired.

Parameters **mct** *(choicemodels.tools.MergedChoiceTable)* – This parameter is a temporary backdoor allowing us to pass in a more complicated choice table than can be generated within the template, for example including sampling weights or interaction terms.

`classmethod from_dict(d)`
Create an object instance from a saved dictionary representation.

Parameters **d** *(dict)* –

Returns **Return type** *SegmentedLargeMultinomialLogitStep*

`get_segmentation_column()`
Get the column of segmentation values from Orca. Chooser and alternative filters are applied to identify valid observations.

Returns **Return type** *pd.Series*

`run()`
Convenience method (required by template spec) that invokes `run_all()`.

`run_all(interaction_terms=None)`
Run all the submodels.

Parameters **interaction_terms** *(pandas.Series, pandas.DataFrame, or list of either, optional)* – Additional column(s) of interaction terms whose values depend on the combination of observation and alternative, to be merged onto the final data table. If passed as a Series or DataFrame, it should include a two-level MultiIndex. One level’s name and values should match an index or column from the observations table, and the other should match an index or column from the alternatives table.

`to_dict()`
Create a dictionary representation of the object.

Returns **Return type** *dict*
update_submodels(param, value)

Updates a property across all the submodels. This method is bound to the defaults object and runs automatically when one of its properties is changed.

Note that the chooser_filters and alt_filters properties cannot currently be updated this way, because they can affect the model segmentation. If you are confident the changes are valid, you can edit the submodels directly. Otherwise, you can regenerate them using updated defaults by running build_submodels().

Parameters

- **param** (str) – Property name.
- **value** (anything) –

1.3.6 Template Step parent class
class urbansim_templates.models.TemplateStep(tables=None, model_expression=None, filters=None, out_tables=None, out_column=None, out_transform=None, out_filters=None, name=None, tags=[])  

Shared functionality for the template classes.

Parameters

- **tables** (str or list of str, optional) – Required to fit a model, but doesn’t have to be provided at initialization.
- **model_expression** (str, optional) – Required to fit a model, but doesn’t have to be provided at initialization.
- **filters** (str or list of str ?, optional) – Replaces fit_filters argument.
- **out_tables** (str or list of str, optional) –
- **out_column** (str, optional) – Replaces out_fname argument.
- **out_transform** (callable, optional) – Replaces ytransform argument.
- **out_filters** (str or list of str ?, optional) – Replaces predict_filters argument.
- **name** (str, optional) – For ModelManager.
- **tags** (list of str, optional) – For ModelManager.

classmethod from_dict(d)

Create an object instance from a saved dictionary representation.

Child classes will need to override this method to implement loading of custom parameters and estimation results.

Parameters **d** (dict) –

Returns

Return type TemplateStep
to_dict()

Create a dictionary representation of the object.

Child classes will need to override this method to implement saving of custom parameters and estimation results.

Returns
1.4 Data management templates

1.4.1 Usage

Data templates help you load tables into Orca, create columns of derived data, or save tables or subsets of tables to disk.

```python
from urbansim_templates.data import LoadTable

t = LoadTable()
t.table = 'buildings'  # a name for the Orca table
t.source_type = 'csv'
t.path = 'buildings.csv'
t.csv_index_cols = 'building_id'
t.name = 'load_buildings'  # a name for the model step that sets up the table
```

You can run this directly using `t.run()`, or register the configured template to be part of a larger workflow:

```python
from urbansim_templates import modelmanager

modelmanager.register(t)
```

Registration does two things: (a) it saves the configured template to disk as a yaml file, and (b) it creates a model step with logic for loading the table. Running the model step is equivalent to running the configured template object:

```python

t.run()
# equivalent:
import orca
orca.run(['load_buildings'])
```

Strictly speaking, running the model step doesn’t load the data, it just sets up an Orca table with instructions for loading the data when it’s needed. (This is called lazy evaluation.)

```python
orca.run(['load_buildings'])  # now an Orca table named 'buildings' is registered
orca.get_table('buildings').to_frame()  # now the data is read from disk
```

Because “running” the table-loading step is costless, it’s done automatically when you register a configured template. It’s also done automatically when you initialize a ModelManager session and table-loading configs are read from yaml. (If you’d like to disable this for a particular table, you can set `t.autorun == False`.)

**Recommended data schemas**

The `LoadTable` template will work with any data that can be loaded into a Pandas DataFrame. But we highly recommend following stricter data schema rules:

1. Each table should include a unique, named index column (a.k.a. primary key) or set of columns (multi-index, a.k.a composite key).
2. If a column is meant to be a join key for another table, it should have the same name as the index of that table.
3. Duplication of column names across tables (except for the join keys) is discouraged, for clarity.
If you follow these rules, tables can be automatically merged on the fly, for example to assemble estimation data or calculate indicators.

You can use `validate_table()` or `validate_all_tables()` to check whether these expectations are met. When templates merge tables on the fly, they use `merge_tables()`.

These utility functions work with any Orca table that meets the schema expectations, whether or not it was created with a template.

**Compatibility with Orca**

From Orca’s perspective, tables set up using the `LoadTable` template are equivalent to tables that are registered using `orca.add_table()` or the `@orca.table` decorator. Technically, they are `orca.TableFuncWrapper` objects.

Unlike the templates, Orca relies on user-specified “broadcast” relationships to perform automatic merging of tables. `LoadTable` does not register any broadcasts, because they’re not needed if tables follow the schema rules above. So if you use these tables in non-template model steps, you may need to add broadcasts separately.

### 1.4.2 Data loading API

```python
def LoadTable():
    # Template for registering data tables from local CSV or HDF files.
```

**Parameters**

- `table` *(str, optional)* – Name of the Orca table to be created. Must be provided before running the step.
- `source_type` *(‘csv’ or ‘hdf’, optional)* – Source type. Must be provided before running the step.
- `path` *(str, optional)* – Local file path to load data from, either absolute or relative to the ModelManager config directory. Please provide a Unix-style path (this will work on any platform, but a Windows-style path won’t, and they’re hard to normalize automatically).
- `url` *(str, optional – NOT YET IMPLEMENTED)* – Remote url to download file from.
- `csv_index_cols` *(str or list of str, optional)* – Required for tables loaded from csv.
- `extra_settings` *(dict, optional)* – Additional arguments to pass to `pd.read_csv()` or `pd.read_hdf()`. For example, you could automatically extract csv data from a gzip file using `{'compression': ‘gzip’}`, or specify the table identifier within...
a multi-object hdf store using `{key}: `table-name`{}`. See Pandas documentation for additional settings.

- **orca_test_spec**(dict, optional - NOT YET IMPLEMENTED) – Data characteristics to be tested when the table is validated.

- **cache**(bool, default True) – Passed to orca.table(). Note that the default is True, unlike in the underlying general-purpose Orca function, because tables read from disk should not need to be regenerated during the course of a model run.

- **cache_scope** ('step', 'iteration', or 'forever', default 'forever') – Passed to orca.table(). Default is 'forever', as in Orca.

- **copy_col**(bool, default True) – Passed to orca.table(). Default is True, as in Orca.

- **name**(str, optional) – Name of the model step.

- **tags**(list of str, optional) – Tags, passed to ModelManager.

- **autorun**(bool, default True) – Automatically run the step whenever it’s registered with ModelManager.

**classmethod from_dict**(d)
Create an object instance from a saved dictionary representation.

Parameters d(dict) –

Returns

Return type Table

**run**()
Register a data table with Orca.

Requires values to be set for `table`, `source_type`, and `path`. CSV data also requires `csv_index_cols`.

Returns

Return type None

**to_dict**()
Create a dictionary representation of the object.

Returns

Return type dict

### 1.4.3 Column creation API

<table>
<thead>
<tr>
<th>ColumnFromExpression([meta, data, output])</th>
<th>Template to register a column of derived data with Orca, based on an expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExpressionSettings([table, expression])</td>
<td>Stores custom parameters used by the ColumnFromExpression template.</td>
</tr>
</tbody>
</table>

class urbansim_templates.data.ColumnFromExpression (meta=None, data=None, output=None)

Template to register a column of derived data with Orca, based on an expression. Parameters may be passed to the constructor, but they are easier to set as attributes. The expression can refer to any columns in the same table, and will be evaluated using `df.eval()`. Values will be calculated lazily, only when the column is needed for
a specific operation.

Parameters

- **meta** (*CoreTemplateSettings*, optional) – Standard parameters. This template sets the default value of `meta.autorun` to `True`.
- **data** (*ExpressionSettings*, optional) – Special parameters for this template.
- **output** (*OutputColumnSettings*, optional) – Parameters for the column that will be generated. This template uses `data.table` as the default value for `output.table`.

```python
classmethod from_dict(d)
Create a class instance from a saved dictionary.

classmethod from_dict_0_2_dev5(d)
Converter to read saved data from 0.2.dev5 or earlier. Automatically invoked by `from_dict()` as needed.

run()
Run the template, registering a column of derived data with Orca. Requires values to be set for `data.table`, `data.expression`, and `output.column_name`.

to_dict()
Create a dictionary representation of the object.
```

```python
class urbansim_templates.data.ExpressionSettings(table=None, expression=None)
Stores custom parameters used by the `ColumnFromExpression` template. Parameters can be passed to the constructor or set as attributes.

Parameters

- **table** (*str*, optional) – Name of Orca table the expression will be evaluated on. Required before running then template.
- **expression** (*str*, optional) – String describing operations on existing columns of the table, for example “a/log(b+c)”. Required before running. Supports arithmetic and math functions including sqrt, abs, log, log1p, exp, and expm1 – see Pandas `df.eval()` documentation for further details.
```

## 1.4.4 Data output API

### SaveTable([table, columns, filters,…])
Template for saving Orca tables to local CSV or HDF5 files.

```python
class urbansim_templates.data.SaveTable(table=None, columns=None, filters=None, output_type=None, path=None, extra_settings=None, name=None, tags=[])
Template for saving Orca tables to local CSV or HDF5 files. Parameters can be passed to the constructor or set as attributes.

Parameters

- **table** (*str*, optional) – Name of the Orca table. Must be provided before running the step.
- **columns** (*str or list of str*, optional) – Names of columns to include. `None` will return all columns. Indexes will always be included.
- **filters** (*str or list of str*, optional) – Filters to apply to the data before
saving. Will be passed to `pd.DataFrame.query()`.

- **output_type** ("csv" or "hdf", optional) – Type of file to be created. Must be provided before running the step.

- **path** (str, optional) – Local file path to save the data to, either absolute or relative to the ModelManager config directory. Please provide a Unix-style path (this will work on any platform, but a Windows-style path won’t, and they’re hard to normalize automatically). For dynamic file names, you can include the characters “%RUN%”, “%ITER%”, or “%TS%”. These will be replaced by the run id, the model iteration value, or a timestamp when the output file is created.

- **extra_settings** (dict, optional) – Additional arguments to pass to `pd.to_csv()` or `pd.to_hdf()`. For example, you could automatically compress csv data using {'compression': 'gzip'}, or specify a custom table name for an hdf store using {'key': 'table-name’}. See Pandas documentation for additional settings.

- **name** (str, optional) – Name of the model step.

- **tags** (list of str, optional) – Tags, passed to ModelManager.

classmethod from_dict(d)

Create an object instance from a saved dictionary representation.

Parameters  

- **d** (dict) –

Returns

Return type  Table

get_dynamic_filepath()

Substitute run id, model iteration, and/or timestamp into the filename.

For the run id and model iteration, we look for Orca injectables named `run_id` and `iter_var`, respectively. If none is found, we use 0.

The timestamp is UTC, formatted as `YYYYMMDD-HHmmSS`.

Returns

Return type  str

run()

Save a table to disk.

Saving a table to an HDF store requires providing a key that will be used to identify the table in the store. We’ll use the Orca table name, unless you provide a different key in the `extra_settings`.

Returns

Return type  None

to_dict()

Create a dictionary representation of the object.

Returns

Return type  dict

1.5 Shared utilities

The utilities are mainly helper functions for templates.
1.5.1 General template tools API

**CoreTemplateSettings**([name, tags, notes, ...]) Stores standard parameters and logic used by all templates.

```python
class urbansim_templates.shared.CoreTemplateSettings(
    name=None, tags=[], notes=None, autorun=False, template=None, template_version=None)
```

Stores standard parameters and logic used by all templates. Parameters can be passed to the constructor or set as attributes.

**Parameters**

- **name** *(str, optional)* - Name of the configured template instance.
- **tags** *(list of str, optional)* - Tags associated with the configured template instance.
- **notes** *(str, optional)* - Notes associated with the configured template instance.
- **autorun** *(bool, optional)* - Whether to run the configured template instance automatically when it’s registered or loaded by ModelManager. The overall default is False, but the default can be overridden at the template level.
- **template** *(str)* - Name of the template class associated with a configured instance.
- **template_version** *(str)* - Version of the template class package.

**classmethod from_dict**(d) Create a class instance from a saved dictionary representation.

**Parameters**

- **d** *(dict)*

**Returns** *obj*

**Return type** *CoreTemplateSettings*

**to_dict**() Create a dictionary representation of the object.

**Returns** *d*

**Return type** *dict*

1.5.2 Column output tools API

**OutputColumnSettings**([column_name, table, ...]) Stores standard parameters used by templates that generate or modify columns.

```python
class urbansim_templates.shared.OutputColumnSettings(
    column_name=None, table=None, data_type=None, missing_values=None, cache=False, cache_scope='forever')
```

Stores standard parameters used by templates that generate or modify columns. Parameters can be passed to the constructor or set as attributes.
Parameters

- **column_name (str, optional)** – Name of the Orca column to be created or modified. Generally required before running a configured template.
- **table (str, optional)** – Name of Orca table the column will be associated with. Generally required before running the configured template.
- **data_type (str, optional)** – Python type or `numpy.dtype` to case the column’s values to.
- **missing_values (str or numeric, optional)** – Value to use for rows that would otherwise be missing.
- **cache (bool, default False)** – Whether to cache column values after they are calculated
- **cache_scope ('step', 'iteration', or 'forever', default 'forever')** – How long to cache column values for (ignored if cache is False).

**classmethod from_dict(d)**

Create a class instance from a saved dictionary representation.

**Parameters**

- **d (dict)** –

**Returns**

- **obj**

**Return type** *OutputColumnSettings*

**to_dict()**

Create a dictionary representation of the object.

**Returns**

- **d**

**Return type** *dict*

**urbansim_templates.shared.register_column(build_column, settings)**

Register a callable as an Orca column.

**Parameters**

- **build_column (callable)** – Callable should return a `pd.Series`.
- **settings (ColumnOutputSettings)** –

1.5.3 Table schemas and merging API

- **validate_table(table[, reciprocal])**
  Check some basic expectations about an Orca table:
- **validate_all_tables()**
  Validate all tables registered with Orca.
- **merge_tables(tables[, columns])**
  Merge two or more tables into a single DataFrame.

**urbansim_templates.utils.validate_table(table, reciprocal=True)**

Check some basic expectations about an Orca table:

- Confirm that it includes a unique, named index column (a.k.a. primary key) or set of columns (multi-index, a.k.a. composite key). If not, raise a ValueError.
- Confirm that none of the other columns in the table share names with the index(es). If they do, raise a ValueError.
- If the table contains columns whose names match the index columns of other tables registered with Orca, check whether they make sense as join keys. This prints a status message with the number of presumptive
foreign-key values that are found in the primary/composite key, for evaluation by the user.

- Perform the same check for columns in _other_ tables whose names match the index column(s) of _this_ table.
- It doesn’t currently compare indexes to indexes. (Maybe it should?)

Running this will trigger loading all registered Orca tables, which may take a while. Stand-alone columns will not be loaded unless their names match an index column.

Doesn’t currently incorporate orca_test validation, but it might be added.

**Parameters**

- **table**(str) – Name of Orca table to validate.
- **reciprocal**(bool, default True) – Whether to also check how columns of other tables align with this one’s index. If False, only check this table’s columns against other tables’ indexes.

**Returns**

**Return type** bool

urbansim_templates.utils.validate_all_tables()
Validate all tables registered with Orca. See validate_table() above.

**Returns**

**Return type** bool

urbansim_templates.utils.merge_tables(tables, columns=None)
Merge two or more tables into a single DataFrame.

All the data will eventually be merged onto the first table in the list. In each merge stage, we’ll refer to the right-hand table as the “source” and the left-hand one as the “target”.

Tables are merged using ModelManager schema rules: The source table must have a unique index, and the target table must have a column with a matching name, which will be used as the join key. Multi-indexes are fine, but all of the index columns need to be present in the target table.

The last table in the list is the initial source. The algorithm searches backward through the list for a table that qualifies as a target. The source table is left-joined onto the target, and then the algorithm continues with the second-to-last table as the new source.

Example 1: Tables A and B share join keys. Tables B and C share join keys. Merging [A, B, C] will left-join C onto B, and then left-join the result onto A.

Example 2: Tables A and B share join keys. Tables A and C also share join keys, but tables B and C don’t. Merging [A, B, C] will left-join C onto A, and then left-join B onto the result of the first join.

If you provide a list of columns, the output table will be limited to columns in this list. The index(es) of the left-most table will always be retained, but it’s a good practice to list them anyway. Column names not found will be ignored.

If two tables contain columns with identical names (other than join keys), they can’t be automatically merged. If the columns are just incidental and not needed in the final output, you can perform the merge by providing a columns list that excludes them.

A note about data types: They will be retained, but if NaN values need to be added (e.g. if some identifiers from the target table aren’t found in the source table), data may need to be cast to a type that allows missing values. For better control over this, see urbansim_templates.data.ColumnFromBroadcast().

**Parameters**
• **tables** *(list of str, orca.DataFrameWrapper, orca.TableFuncWrapper, or pd.DataFrame)* – Two or more tables to merge. Types can be mixed and matched.

• **columns** *(list of str, optional)* – Names of columns to retain in the final output.

**Returns**

**Return type** *pd.DataFrame*

### 1.5.4 Other helper functions API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>all_cols</strong> <em>(table)</em></td>
<td>Returns a list of all column names in a table, including index(es). Input can be an Orca table name, orca.DataFrameWrapper, orca.TableFuncWrapper, or pd.DataFrame.</td>
</tr>
<tr>
<td><strong>cols_in_expression</strong> <em>(expression)</em></td>
<td>Extract all possible column names from a df.eval()-style expression.</td>
</tr>
<tr>
<td><strong>get_data</strong> <em>(tables[, fallback_tables, filters, ...]</em>)</td>
<td>Generate a pd.DataFrame for model estimation or simulation.</td>
</tr>
<tr>
<td><strong>get_df</strong> <em>(table[, columns]</em>)</td>
<td>Returns a table as a pd.DataFrame.</td>
</tr>
<tr>
<td><strong>trim_cols</strong> <em>(df[, columns]</em>)</td>
<td>Limit a DataFrame to columns that appear in a list of names.</td>
</tr>
<tr>
<td><strong>to_list</strong> <em>(items)</em></td>
<td>In many places we accept either a single string or a list of strings.</td>
</tr>
<tr>
<td><strong>update_column</strong> <em>(table, column, data[, ...]</em>)</td>
<td>Update an Orca column.</td>
</tr>
<tr>
<td><strong>update_name</strong> <em>(template[, name]</em>)</td>
<td>Generate a name for a configured model step, based on its template class and the current timestamp.</td>
</tr>
</tbody>
</table>

#### urbansim_templates.utils.all_cols *(table)*

Returns a list of all column names in a table, including index(es). Input can be an Orca table name, orca.DataFrameWrapper, orca.TableFuncWrapper, or pd.DataFrame.

**Parameters**

- **table** *(str, orca.DataFrameWrapper, orca.TableFuncWrapper, or pd.DataFrame)* –

**Returns**

**Return type** *list of str*

#### urbansim_templates.utils.cols_in_expression *(expression)*

Extract all possible column names from a df.eval()-style expression.

This is achieved using regex to identify tokens in the expression that begin with a letter and contain any number of alphanumericcs or underscores, but do not end with an opening parenthesis. This excludes function names, but would not exclude constants (e.g. “pi”), which are semantically indistinguishable from column names.

**Parameters**

- **expression** *(str)* –

**Returns**

**cols** *list of str*

**Return type** *list of str*

#### urbansim_templates.utils.get_data *(tables, fallback_tables=None, filters=None, model_expression=None, extra_columns=None)*

Generate a pd.DataFrame for model estimation or simulation. Automatically loads tables from Orca, merges them, and removes columns not referenced in a model expression or data filter. Additional columns can be requested.
If filters are provided, the output will include only rows that match the filter criteria.

See `urbansim_templates.utils.merge_tables()` for a detailed description of how the merges are performed.

**Parameters**

- `tables` *(str or list of str)* – Orca table(s) to draw data from.
- `fallback_tables` *(str or list of str, optional)* – Table(s) to use if first parameter evaluates to `None`. (This option will be removed shortly when estimation and simulation settings are separated.)
- `filters` *(str or list of str, optional)* – Filter(s) to apply to the merged data, using `pd.DataFrame.query()`.
- `model_expression` *(str, optional)* – Model expression that will be evaluated using the output data. Only used to drop non-relevant columns. PyLogit format is not yet supported.
- `extra_columns` *(str or list of str, optional)* – Columns to include, in addition to any in the model expression and filters. (If this and the model_expression are both `None`, all columns will be included.)

**Returns**

**Return type** `pd.DataFrame`

`urbansim_templates.utils.get_df(table, columns=None)`

Returns a table as a `pd.DataFrame`. Input can be an Orca table name, `orca.DataFrameWrapper`, `orca.TableFuncWrapper`, or `pd.DataFrame`.

Optionally, columns can be limited to those that appear in a list of names. The list may contain duplicates or columns not in the table. Index(es) will always be retained, but it’s a good practice to list them anyway.

**Parameters**

- `table` *(str, orca.DataFrameWrapper, orca.TableFuncWrapper, or pd.DataFrame)* –
- `columns` *(list of str, optional)* –

**Returns**

**Return type** `pd.DataFrame`

`urbansim_templates.utils.trim_cols(df, columns=None)`

Limit a DataFrame to columns that appear in a list of names. List may contain duplicates or names not in the DataFrame. Index(es) of the DataFrame will always be retained, but it’s a good practice to list them anyway. If `columns` is `None`, all columns are retained. Returns the original DataFrame, not a copy.

**Parameters**

- `df` *(pd.DataFrame)* –
- `columns` *(list of str, optional)* –

**Returns**

**Return type** `pd.DataFrame`

`urbansim_templates.utils.to_list(items)`

In many places we accept either a single string or a list of strings. This function normalizes `None` -> `[None]`, `str` -> `[str]`, and leaves lists unchanged.

**Parameters**

- `items` *(str, list, or None)* –
Returns

Return type  list

`urbansim_templates.utils.update_column(table, column, data, fallback_table=None, fallback_column=None)`

Update an Orca column. If it doesn’t exist yet, add it to the wrapped DataFrame. Values will be aligned using
the indexes if possible.

Data types: If the column already exists, new values will be cast to match the existing data type. If the column
is new, it will retain the data type of the pd.Series that’s passed to this function – unless it doesn’t fully align
with the table’s index, in which case it may be cast to allow missing values (e.g. from int to float).

Parameters

- **table** *(str or list of str)* – Name of Orca table to update. If list, the first ele-
  ment will be used.
- **column** *(str)* – Name of existing column to update, or new column to create. Cannot be
  an index.
- **data** *(pd.Series)* – Column of data to update or add.
- **fallback_table** *(str or list of str)* – Name of Orca table to use if table
  evaluates to None.
- **fallback_column** *(str)* – Name of Orca column to use if column evaluates to None.

Returns

Return type  None

`urbansim_templates.utils.update_name(template, name=None)`

Generate a name for a configured model step, based on its template class and the current timestamp. But if
a custom name has already been provided, return that instead. (A name is judged to be custom if it does not
contain the class type.)

Parameters

- **template** *(str)* – Template class name.
- **name** *(str, optional)* – Existing name for the configured model step.

Returns

Return type  str

1.5.5 Spec validation API

| validate_template(cls) | Checks whether a template class meets the basic expec-
|                        | tations for working with ModelManager, to aid in devel-
|                        | opment and testing. |

`urbansim_templates.utils.validate_template(cls)`

Checks whether a template class meets the basic expectations for working with ModelManager, to aid in develop-
ment and testing.

Looks for ‘to_dict’, ‘from_dict’, and ‘run’ methods, and ‘name’, ‘tags’, ‘template’, and ‘template_version’
attributes. Checks that an object can be instantiated without arguments, plus some additional behaviors. See
documentation for a full description of ModelManager specs and guidelines.

There are many behaviors this does NOT check, because we don’t know what particular parameters are expected
and valid for a given template. For example, saving a configured model step and reloading it should produce an equivalent object, but this needs to be checked in template-specific unit tests.

**Parameters**

- **cls (class)** – Template class.

**Returns**

**Return type** bool

### 1.5.6 Version management API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>parse_version(v)</code></td>
<td>Parses a version string into its component parts.</td>
</tr>
<tr>
<td><code>version_greater_or_equal(a, b)</code></td>
<td>Tests whether version string 'a' is greater than or equal to version string 'b'.</td>
</tr>
</tbody>
</table>

**urbansim_templates.utils.parse_version(v)**

Parses a version string into its component parts. String is expected to follow the pattern “0.1.1.dev0”, which would be parsed into (0, 1, 1, 0). The first two components are required. The third is set to 0 if missing, and the fourth to None.

**Parameters**

- **v (str)** – Version string using syntax described above.

**Returns**

**Return type** tuple with format (int, int, int, int or None)

**urbansim_templates.utils.version_greater_or_equal(a, b)**

Tests whether version string ‘a’ is greater than or equal to version string ‘b’. Version syntax should follow the pattern described for `version_parse()`.

Note that ‘dev’ versions are pre-releases, so ‘0.2’ < ‘0.2.1.dev5’ < ‘0.2.1’.

**Parameters**

- **a (str)** – First version string, formatted as described in `version_parse()`.
- **b (str)** – Second version string, formatted as described in `version_parse()`.

**Returns**

**Return type** boolean

### 1.6 Development guide

Below are some strategies we’ve come up with for the templates. Technical contribution guidelines are in the Github repo.

#### 1.6.1 Design patterns for templates

A ModelManager-compliant template is a Python class that conforms to the following spec:

1. can save itself to a dict using a method named `to_dict()`
2. can rebuild itself from a dict using a method named `from_dict()`
3. can execute a configured version of itself using a method named `run()`
4. accepts parameters `name (str)` and `tags (list of str)`
5. uses the `@modelmanager.template` decorator

Running a configured model step executes logic and typically saves output to Orca.

Templates should try to use parameter names that are consistent or harmonious with other templates.

Tables and columns of data should be input as named Orca objects. Other inputs that are hard to store as strings (like callables) should probably be input as Orca objects as well; we’re still working on a solution for this.

All template inputs should be accepted either as constructor parameters or object properties, if feasible:

```python
m1 = TemplateStep(foo='yes')
m2 = TemplateStep()
m2.foo = 'yes'
```

It’s fine for templates to require interactive configuration, like fitting a statistical model. Also fine to require these actions to be completed before the model step can be saved or run.

Ideally, users should be able to edit object properties and re-run the interactive components whenever they like. Changes will not be saved until a an object is re-registered with ModelManager.

Lightweight intermediate outputs like summary tables and fitted parameters should be saved in an object’s dictionary representation if feasible.

Bigger intermediate outputs, like pickled copies of full fitted models, can be automatically stored to disk by providing an entry named `supplemental_objects` in a model’s dictionary representation. This should contain a list of dicts, each of which has parameters `name` (str), `content` (obj), and `content_type` (str, e.g. 'pickle').

To avoid dependency bloat, the default installation only includes the dependencies required for core model management and the most commonly used templates. Templates using additional libraries should check whether they’re installed before fitting or running a model step, and provide helpful error messages if not.
Python Module Index

U

urbansim_templates.modelmanager, 7
urbansim_templates.shared, 25
urbansim_templates.utils, 31
Index

A
all_cols() (in module urbansim_templates.utils), 28

B
BinaryLogitStep (class in urbansim_templates.models), 10
build_step() (in module urbansim_templates.modelmanager), 7
build_submodels() (urban_sim_templates.models.SegmentedLargeMultinomialLogitStep method), 18

C
choices(urbansim_templates.models.LargeMultinomialLogitStep attribute), 16
cols_in_expression() (in module urbansim_templates.data), 28
ColumnFromExpression (class in urbansim_templates.data), 22
CoreTemplateSettings (class in urbansim_templates.shared), 25

E
ExpressionSettings (class in urbansim_templates.data), 23

F
fit() (urbansim_templates.models.BinaryLogitStep method), 11
fit() (urbansim_templates.models.LargeMultinomialLogitStep method), 16
fit() (urbansim_templates.models.OLSRegressionStep method), 10
fit() (urbansim_templates.models.SmallMultinomialLogitStep method), 13
fit_all() (urbansim_templates.models.SegmentedLargeMultinomialLogitStep method), 18
from_dict() (urbansim_templates.data.ColumnFromExpression class method), 23
from_dict() (urbansim_templates.data.LoadTable class method), 22
from_dict() (urbansim_templates.data.SaveTable class method), 24
from_dict() (urban_sim_templates.models.BinaryLogitStep class method), 11
from_dict() (urban_sim_templates.models.LargeMultinomialLogitStep class method), 16
from_dict() (urban_sim_templates.models.OLSRegressionStep class method), 10
from_dict() (urban_sim_templates.models.SegmentedLargeMultinomialLogitStep class method), 18
from_dict() (urban_sim_templates.models.SmallMultinomialLogitStep class method), 13
from_dict() (urban_sim_templates.models.TemplateStep class method), 19
from_dict() (urban_sim_templates.shared.CoreTemplateSettings class method), 25
from_dict() (urban_sim_templates.shared.OutputColumnSettings class method), 26
from_dict_0_2_dev5() (urban_sim_templates.data.ColumnFromExpression class method), 23

G
g_config_dir() (in module urbansim_templates.modelmanager), 8
get_df() (in module urbansim_templates.utils), 29
g_get_dynamic_filepath() (urban_sim_templates.data.SaveTable method), 24
get_segmentation_column()  (urban-sim_templates.models.SegmentedLargeMultinomialLogitStep method), 18
get_step()  (in module urban-sim_templates.modelmanager), 7
initialize()  (in module urban-sim_templates.modelmanager), 6
LargeMultinomialLogitStep  (class in urban-sim_templates.models), 14
list_steps()  (in module urban-sim_templates.modelmanager), 7
load_supplemental_object()  (in module urban-sim_templates.modelmanager), 8
LoadTable  (class in urbansim_templates.data), 21
merge_tables()  (in module urban-sim_templates.utils), 27
mergedchoicetable  (urban-sim_templates.models.LargeMultinomialLogitStep attribute), 16
model  (urban-sim_templates.models.LargeMultinomialLogitStep attribute), 16
OLSRegressionStep  (class in urban-sim_templates.models), 9
OutputColumnSettings  (class in urban-sim_templates.shared), 25
parse_version()  (in module urban-sim_templates.utils), 31
probabilities  (urban-sim_templates.models.LargeMultinomialLogitStep attribute), 16
register()  (in module urban-sim_templates.modelmanager), 7
register_column()  (in module urban-sim_templates.shared), 26
remove_step()  (in module urban-sim_templates.modelmanager), 7
remove_supplemental_object()  (in module urban-sim_templates.modelmanager), 8
run()  (urbansim_templates.data.ColumnFromExpression method), 23
run()  (urbansim_templates.data.LoadTable method), 22
run()  (urbansim_templates.data.SaveTable method), 24
run()  (urban-sim_templates.models.BinaryLogitStep method), 12
run()  (urban-sim_templates.models.LargeMultinomialLogitStep method), 17
run()  (urban-sim_templates.models.OLSRegressionStep method), 10
run()  (urban-sim_templates.models.SegmentedLargeMultinomialLogitStep method), 18
run()  (urban-sim_templates.models.SmallMultinomialLogitStep method), 13
run_all()  (urban-sim_templates.models.SegmentedLargeMultinomialLogitStep method), 18
save_step_to_disk()  (in module urban-sim_templates.modelmanager), 7
save_supplemental_object()  (in module urban-sim_templates.modelmanager), 8
SaveTable  (class in urbansim_templates.data), 23
SegmentedLargeMultinomialLogitStep  (class in urban-sim_templates.models), 17
SmallMultinomialLogitStep  (class in urban-sim_templates.models), 12
template()  (in module urban-sim_templates.modelmanager), 7
to_dict()  (urbansim_templates.data.ColumnFromExpression method), 19
to_dict()  (urbansim_templates.data.LoadTable method), 23
to_dict()  (urbansim_templates.data.SaveTable method), 22
to_dict()  (urbansim_templates.data.SaveTable method), 24
to_dict()  (urban-sim_templates.models.BinaryLogitStep method), 12
to_dict()  (urban-sim_templates.models.LargeMultinomialLogitStep method), 17
to_dict()  (urban-sim_templates.models.OLSRegressionStep method), 10
to_dict()  (urban-sim_templates.models.SegmentedLargeMultinomialLogitStep method), 18
to_dict()  (urban-sim_templates.models.SmallMultinomialLogitStep method), 14
to_dict()  (urban-sim_templates.models.TemplateStep method), 19
to_dict()  (urban-sim_templates.shared.CoreTemplateSettings method), 25
to_dict()  (urban-sim_templates.shared.OutputColumnSettings method), 26
to_list() (in module urbansim_templates.utils), 29
trim_cols() (in module urbansim_templates.utils), 29

U
update_column() (in module urbansim_templates.utils), 30
update_name() (in module urbansim_templates.utils), 30
update_submodels() (urbansim_templates.models.SegmentedLargeMultinomialLogitStep method), 18
urbansim_templates.modelmanager (module), 6, 7
urbansim_templates.shared (module), 25
urbansim_templates.utils (module), 26, 28, 30, 31

V
validate_all_tables() (in module urbansim_templates.utils), 27
validate_table() (in module urbansim_templates.utils), 26
validate_template() (in module urbansim_templates.utils), 30
version_greater_or_equal() (in module urbansim_templates.utils), 31