Scenic

Daniel J. Fremont, Eric Vin, Edward Kim, Tommaso Dreossi, Shro

Jun 30, 2023

INTRODUCTION

Scenic is a domain-specific probabilistic programming language for modeling the environments of cyber-physical systems like robots and autonomous cars. A Scenic program defines a distribution over *scenes*, configurations of physical objects and agents; sampling from this distribution yields concrete scenes which can be simulated to produce training or testing data. Scenic can also define (probabilistic) policies for dynamic agents, allowing modeling scenarios where agents take actions over time in response to the state of the world.

Scenic was designed and implemented by Daniel J. Fremont, Eric Vin, Edward Kim, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia, with contributions from *many others*. For a description of the language and some of its applications, see our journal paper, which extends our PLDI 2019 paper on Scenic 1.x. Our *publications* page lists additional papers using Scenic.

Note: The syntax of Scenic 3.0 is not completely backwards-compatible with earlier versions of Scenic, which were used in our papers prior to 2023. See *What's New in Scenic* for a list of syntax changes and new features. Old code can likely be easily ported; you can also install older releases if necessary from GitHub.

If you have any problems using Scenic, please submit an issue to our GitHub repository or contact Daniel at dfremont@ucsc.edu.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Getting Started with Scenic

1.1.1 Installation

Scenic requires **Python 3.8** or newer. Run **`python --version`** to make sure you have a new enough version; if not, you can install one from the Python website or using pyenv (e.g. running **`pyenv install 3.11`**). If the version of Python you want to use is called something different than just `python` on your system, e.g. `python3.11`, use that name in place of `python` when creating a virtual environment below.

There are two ways to install Scenic:

- from our repository, which has the very latest features but may not be stable. The repository also contains example scenarios such as those used in the instructions below and our tutorials.
- from the Python Package Index (PyPI), which will get you the latest official release of Scenic but will not include example scenarios, etc.

If this is your first time using Scenic, we suggest installing from the repository so that you can try out the example scenarios.

Once you've decided which method you want to use, follow the instructions below for your operating system. If you encounter any errors, please see our *Notes on Installing Scenic* for suggestions.

macOS

Start by downloading Blender and OpenSCAD and installing them into your `Applications` directory.

Next, activate the virtual environment in which you want to install Scenic. To create and activate a new virtual environment called `venv`, you can run the following commands:

```
python3 -m venv venv
source venv/bin/activate
```

Once your virtual environment is activated, you no longer need to use a name like `python3` or `python3.11`; use just **`python`** to ensure you're running the copy of Python in your virtual environment.

Next, make sure your **`pip`** tool is up-to-date:

```
python -m pip install --upgrade pip
```

Now you can install Scenic either from the repository or from PyPI:

Repository

The following commands will clone the Scenic repository into a folder called `Scenic` and install Scenic from there. It is an "editable install", so if you later update the repository with **`git pull`** or make changes to the code yourself, you won't need to reinstall Scenic.

```
git clone https://github.com/BerkeleyLearnVerify/Scenic
cd Scenic
python -m pip install -e .
```

If you will be developing Scenic, you will want to use a variant of the last command to install additional development dependencies: see *Developing Scenic*.

PyPI

The following command will install the latest full release of Scenic from PyPI:

```
python -m pip install scenic
```

Note that this command skips experimental alpha and beta releases, preferring stable versions. If you want to get the very latest version available on PyPI (which may still be behind the repository), run:

```
python -m pip install --pre scenic
```

You can also install specific versions with a command like:

```
python -m pip install scenic==2.0.0
```

Next, activate the virtual environment in which you want to install Scenic. To create and activate a new virtual environment called `venv`, you can run the following commands:

```
python3 -m venv venv
source venv/bin/activate
```

Once your virtual environment is activated, you no longer need to use a name like `python3` or `python3.11`; use just **`python`** to ensure you're running the copy of Python in your virtual environment.

Next, make sure your **`pip`** tool is up-to-date:

```
python -m pip install --upgrade pip
```

Now you can install Scenic either from the repository or from PyPI:

Repository

The following commands will clone the Scenic repository into a folder called `Scenic` and install Scenic from there. It is an "editable install", so if you later update the repository with **`git pull`** or make changes to the code yourself, you won't need to reinstall Scenic.

```
git clone https://github.com/BerkeleyLearnVerify/Scenic
cd Scenic
python -m pip install -e .
```

If you will be developing Scenic, you will want to use a variant of the last command to install additional development dependencies: see *Developing Scenic*.

PyPI

The following command will install the latest full release of Scenic from PyPI:

```
python -m pip install scenic
```

Note that this command skips experimental alpha and beta releases, preferring stable versions. If you want to get the very latest version available on PyPI (which may still be behind the repository), run:

```
python -m pip install --pre scenic
```

You can also install specific versions with a command like:

```
python -m pip install scenic==2.0.0
```

Linux

Start by installing the Python-Tk interface, Blender, and OpenSCAD. You can likely use your system's package manager; e.g. on Debian/Ubuntu run:

```
sudo apt-get install python3-tk blender openscad
```

For other Linux distributions or if you need to install from source, see the download pages for Blender and OpenSCAD.

Next, activate the virtual environment in which you want to install Scenic. To create and activate a new virtual environment called `venv`, you can run the following commands:

```
python3 -m venv venv
source venv/bin/activate
```

Once your virtual environment is activated, you no longer need to use a name like `python3` or `python3.11`; use just **`python`** to ensure you're running the copy of Python in your virtual environment.

Next, make sure your **`pip`** tool is up-to-date:

```
python -m pip install --upgrade pip
```

Now you can install Scenic either from the repository or from PyPI:

Repository

The following commands will clone the Scenic repository into a folder called `Scenic` and install Scenic from there. It is an "editable install", so if you later update the repository with **`git pull`** or make changes to the code yourself, you won't need to reinstall Scenic.

```
git clone https://github.com/BerkeleyLearnVerify/Scenic
cd Scenic
python -m pip install -e .
```

If you will be developing Scenic, you will want to use a variant of the last command to install additional development dependencies: see *Developing Scenic*.

PyPI

The following command will install the latest full release of Scenic from PyPI:

```
python -m pip install scenic
```

Note that this command skips experimental alpha and beta releases, preferring stable versions. If you want to get the very latest version available on PyPI (which may still be behind the repository), run:

```
python -m pip install --pre scenic
```

You can also install specific versions with a command like:

```
python -m pip install scenic==2.0.0
```

Next, activate the virtual environment in which you want to install Scenic. To create and activate a new virtual environment called `venv`, you can run the following commands:

```
python3 -m venv venv
source venv/bin/activate
```

Once your virtual environment is activated, you no longer need to use a name like `python3` or `python3.11`; use just **`python`** to ensure you're running the copy of Python in your virtual environment.

Next, make sure your **`pip`** tool is up-to-date:

```
python -m pip install --upgrade pip
```

Now you can install Scenic either from the repository or from PyPI:

Repository

The following commands will clone the Scenic repository into a folder called `Scenic` and install Scenic from there. It is an "editable install", so if you later update the repository with **`git pull`** or make changes to the code yourself, you won't need to reinstall Scenic.

```
git clone https://github.com/BerkeleyLearnVerify/Scenic
cd Scenic
python -m pip install -e .
```

If you will be developing Scenic, you will want to use a variant of the last command to install additional development dependencies: see *Developing Scenic*.

PyPI

The following command will install the latest full release of Scenic from PyPI:

```
python -m pip install scenic
```

Note that this command skips experimental alpha and beta releases, preferring stable versions. If you want to get the very latest version available on PyPI (which may still be behind the repository), run:

```
python -m pip install --pre scenic
```

You can also install specific versions with a command like:

```
python -m pip install scenic==2.0.0
```

Windows

These instructions cover installing Scenic natively on Windows; if you are using the Windows Subsystem for Linux (on Windows 10 and newer), see the WSL tab instead.

Start by downloading and running the installers for Blender and OpenSCAD.

Next, activate the virtual environment in which you want to install Scenic. To create and activate a new virtual environment called `venv`, you can run the following commands:

Next, activate the virtual environment in which you want to install Scenic. To create and activate a new virtual environment called `venv`, you can run the following commands:

```
python -m venv venv
venv\Scripts\activate.bat
```

Once your virtual environment is activated, you no longer need to use a name like `python3` or `python3.11`; use just **`python`** to ensure you're running the copy of Python in your virtual environment.

Next, make sure your **`pip`** tool is up-to-date:

```
python -m pip install --upgrade pip
```

Now you can install Scenic either from the repository or from PyPI:

Repository

The following commands will clone the Scenic repository into a folder called `Scenic` and install Scenic from there. It is an "editable install", so if you later update the repository with **`git pull`** or make changes to the code yourself, you won't need to reinstall Scenic.

```
git clone https://github.com/BerkeleyLearnVerify/Scenic
cd Scenic
python -m pip install -e .
```

If you will be developing Scenic, you will want to use a variant of the last command to install additional development dependencies: see *Developing Scenic*.

PyPI

The following command will install the latest full release of Scenic from PyPI:

```
python -m pip install scenic
```

Note that this command skips experimental alpha and beta releases, preferring stable versions. If you want to get the very latest version available on PyPI (which may still be behind the repository), run:

```
python -m pip install --pre scenic
```

You can also install specific versions with a command like:

```
python -m pip install scenic==2.0.0
```

Once your virtual environment is activated, you no longer need to use a name like `python3` or `python3.11`; use just **`python`** to ensure you're running the copy of Python in your virtual environment.

Next, make sure your **`pip`** tool is up-to-date:

```
python -m pip install --upgrade pip
```

Now you can install Scenic either from the repository or from PyPI:

Repository

The following commands will clone the Scenic repository into a folder called `Scenic` and install Scenic from there. It is an "editable install", so if you later update the repository with **`git pull`** or make changes to the code yourself, you won't need to reinstall Scenic.

```
git clone https://github.com/BerkeleyLearnVerify/Scenic
cd Scenic
python -m pip install -e .
```

If you will be developing Scenic, you will want to use a variant of the last command to install additional development dependencies: see *Developing Scenic*.

PyPI

The following command will install the latest full release of Scenic from PyPI:

```
python -m pip install scenic
```

Note that this command skips experimental alpha and beta releases, preferring stable versions. If you want to get the very latest version available on PyPI (which may still be behind the repository), run:

```
python -m pip install --pre scenic
```

You can also install specific versions with a command like:

```
python -m pip install scenic==2.0.0
```

WSL

These instructions cover installing Scenic on the Windows Subsystem for Linux (WSL).

If you haven't already installed WSL, you can do that by running **`wsl --install`** (in either Command Prompt or PowerShell) and restarting your computer. Then open a WSL terminal and run the following commands to install Python, the Python-Tk interface, Blender, and OpenSCAD:

```
sudo apt-get update
sudo apt-get install python3 python3-tk blender openscad
```

Next, activate the virtual environment in which you want to install Scenic. To create and activate a new virtual environment called `venv`, you can run the following commands:

Next, activate the virtual environment in which you want to install Scenic. To create and activate a new virtual environment called `venv`, you can run the following commands:

```
python3 -m venv venv
source venv/bin/activate
```

If you get an error about needing a package like `python3.10-venv`, run

```
sudo apt-get install python3.10-venv
```

(putting in the appropriate Python version) and try the commands above again.

Once your virtual environment is activated, you no longer need to use a name like `python3` or `python3.11`; use just **`python`** to ensure you're running the copy of Python in your virtual environment.

Next, make sure your **`pip`** tool is up-to-date:

```
python -m pip install --upgrade pip
```

Now you can install Scenic either from the repository or from PyPI:

Repository

The following commands will clone the Scenic repository into a folder called `Scenic` and install Scenic from there. It is an "editable install", so if you later update the repository with **`git pull`** or make changes to the code yourself, you won't need to reinstall Scenic.

```
git clone https://github.com/BerkeleyLearnVerify/Scenic
cd Scenic
python -m pip install -e .
```

If you will be developing Scenic, you will want to use a variant of the last command to install additional development dependencies: see *Developing Scenic*.

PyPI

The following command will install the latest full release of Scenic from PyPI:

```
python -m pip install scenic
```

Note that this command skips experimental alpha and beta releases, preferring stable versions. If you want to get the very latest version available on PyPI (which may still be behind the repository), run:

```
python -m pip install --pre scenic
```

You can also install specific versions with a command like:

```
python -m pip install scenic==2.0.0
```

Once your virtual environment is activated, you no longer need to use a name like `python3` or `python3.11`; use just **`python`** to ensure you're running the copy of Python in your virtual environment.

Next, make sure your **`pip`** tool is up-to-date:

```
python -m pip install --upgrade pip
```

Now you can install Scenic either from the repository or from PyPI:

Repository

The following commands will clone the Scenic repository into a folder called `Scenic` and install Scenic from there. It is an "editable install", so if you later update the repository with **`git pull`** or make changes to the code yourself, you won't need to reinstall Scenic.

```
git clone https://github.com/BerkeleyLearnVerify/Scenic
cd Scenic
python -m pip install -e .
```

If you will be developing Scenic, you will want to use a variant of the last command to install additional development dependencies: see *Developing Scenic*.

PyPI

The following command will install the latest full release of Scenic from PyPI:

```
python -m pip install scenic
```

Note that this command skips experimental alpha and beta releases, preferring stable versions. If you want to get the very latest version available on PyPI (which may still be behind the repository), run:

```
python -m pip install --pre scenic
```

You can also install specific versions with a command like:

```
python -m pip install scenic==2.0.0
```

You can now verify that Scenic is properly installed by running the command:

```
scenic --version
```

This should print out a message like `Scenic 3.0.0` showing which version of Scenic is installed. If you get an error (or got one earlier when following the instructions above), please see our *Notes on Installing Scenic* for suggestions.

Note: If a feature described in this documentation seems to be missing, your version of Scenic may be too old: take a look at *What's New in Scenic* to see when the feature was added.

To help read Scenic code, we suggest you install a syntax highlighter plugin for your text editor. Plugins for Sublime Text and Visual Studio Code can be installed from within those tools; for other editors supporting the TextMate grammar format, the grammar is available here.

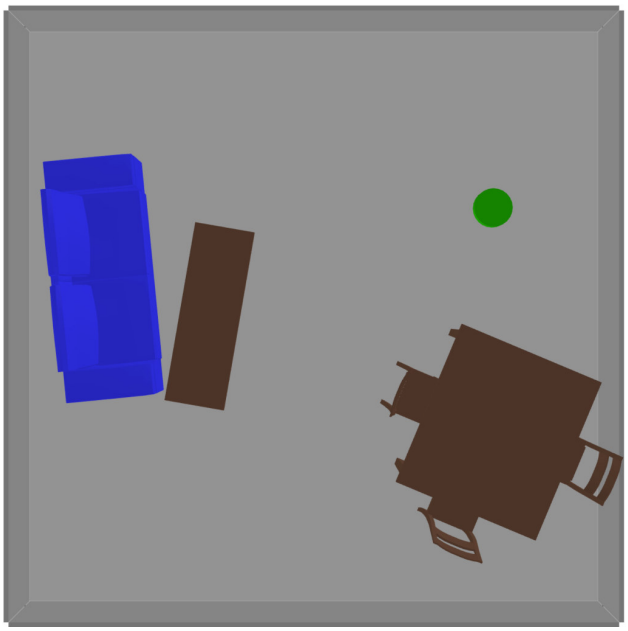
1.1.2 Trying Some Examples

The Scenic repository contains many example scenarios, found in the `examples` directory. They are organized in various directories with the name of the simulator, abstract application domain, or visualizer they are written for. For example, `gta` and `webots` for the GTA (Grand Theft Auto V) and Webots simulators; the `driving` directory for the abstract *driving domain*; and the `visualizer` directory for the built in Scenic visualizer.

Each simulator has a specialized Scenic interface which requires additional setup (see *Supported Simulators*); however, for convenience Scenic provides an easy way to visualize scenarios without running a simulator. Simply run **`scenic`**, giving a path to a Scenic file:

```
scenic examples/webots/vacuum/vacuum_simple.scenic
```

This will compile the Scenic program and sample from it (which may take several seconds), displaying a schematic of the resulting scene. Since this is a simple scenario designed to evaluate the performance of a robot vacuum, you should get something like this:



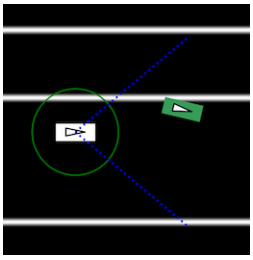
The green cylinder is the vacuum, surrounded by various pieces of furniture in a room. You can adjust the camera angle by clicking and dragging, and zoom in and out using the mouse wheel. If you close the window or press `q`, Scenic will sample another scene from the same scenario and display it. This will repeat until you kill the generator (`Control-c` in the terminal on Linux; `Command-q` in the viewer window on MacOS).

Some scenarios were written for older versions of Scenic, which were entirely 2D. Those scenarios should be run using the `--2d` command-line option, which will enable 2D backwards-compatibility mode. Information about whether or not the `--2d` flag should be used can be found in the `README` of each example directory.

One such scenario is the badly-parked car example from our GTA case study, which can be run with the following command:

```
scenic --2d examples/gta/badlyParkedCar2.scenic
```

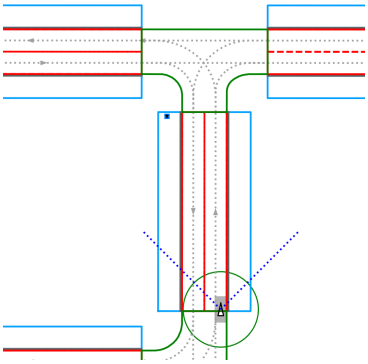
This will open Scenic's 2D viewer, and should look something like this:

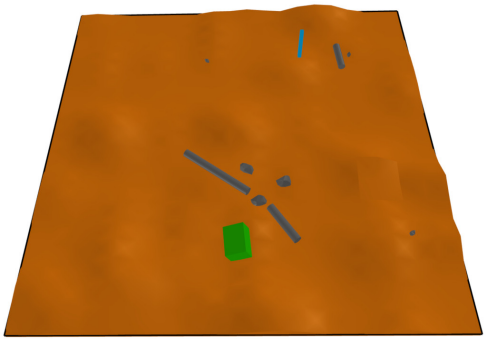


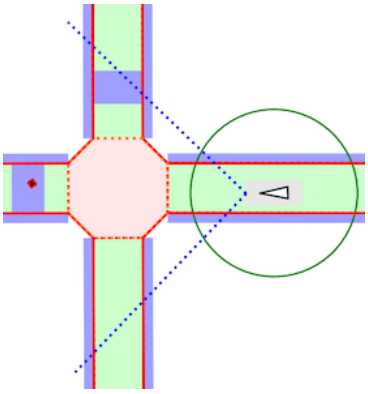
Here the circled rectangle is the ego car; its view cone extends to the right, where we see another car parked rather poorly at the side of the road (the white lines are curbs). (Note that on MacOS, scene generation with the 2D viewer is stopped differently than with the 3D viewer: right-click on its icon in the Dock and select Quit.)

Scenarios for the other simulators can be viewed in the same way. Here are a few for different simulators:

```
scenic --2d examples/driving/pedestrian.scenic
scenic examples/webots/mars/narrowGoal.scenic
scenic --2d examples/webots/road/crossing.scenic
```







The **scenic** command has options for setting the random seed, running dynamic simulations, printing debugging information, etc.: see *Command-Line Options*.

1.1.3 Learning More

Depending on what you'd like to do with Scenic, different parts of the documentation may be helpful:

- If you want to start learning how to write Scenic programs, see *Scenic Fundamentals*.
- If you want to learn how to write dynamic scenarios in Scenic, see *Dynamic Scenarios*.
- If you want to use Scenic with a simulator, see *Supported Simulators* (which also describes how to interface Scenic to a new simulator, if the one you want isn't listed).
- If you want to control Scenic from Python rather than using the command-line tool (for example if you want to collect data from the generated scenarios), see *Using Scenic Programmatically*.
- If you want to add a feature to the language or otherwise need to understand Scenic's inner workings, see our pages on *Developing Scenic* and *Scenic Internals*.

1.2 Notes on Installing Scenic

This page describes common issues with installing Scenic and suggestions for fixing them.

1.2.1 All Platforms

Missing Python Version

If when running **pip** you get an error saying that your machine does not have a compatible version, this means that you do not have Python 3.8 or later on your PATH. Install a newer version of Python, either directly from the Python website or using pyenv (e.g. running **pyenv install 3.10.4**). Then use that version of Python when creating a virtual environment before installing Scenic.

"setup.py" not found

This error indicates that you are using too old a version of `pip`: you need at least version 21.3. Run **python -m pip install --upgrade pip** to upgrade.

Dependency Conflicts

If you install Scenic using `pip`, you might see an error message like the following:

> ERROR: pip's dependency resolver does not currently take into account all the packages that are installed.
> This behaviour is the source of the following dependency conflicts.

This error means that in order to install Scenic, `pip` had to break the dependency constraints of some package you had previously installed (the error message will indicate which one). So while Scenic will work correctly, something else may now be broken. This won't happen if you install Scenic into a fresh virtual environment.

Cannot Find Scenic

If when running the **scenic** command you get a "command not found" error, or when trying to import the `scenic` module you get a `ModuleNotFoundError`, then Scenic has not been installed where your shell or Python respectively can find it. The most likely problem is that you installed Scenic for one copy of Python but are now using a different one: for example, if you installed Scenic in a Python virtual environment (which we highly recommend), you may have forgotten to activate that environment, and so are using your system Python instead. See the virtual environment tutorial for instructions.

Scene Schematics Don't Appear (2D)

If no window appears when you ask Scenic to generate and display a scene using the *--2d* flag (as in the example commands in *Getting Started with Scenic*), this means that Matplotlib has no interactive backend installed. On Linux, try installing the `python3-tk` package (e.g. **sudo apt-get install python3-tk**).

Missing SDL

If you get an error about SDL being missing, you may need to install it. On Linux (or Windows with *WSL*), install the `libsdl2-dev` package (e.g. **`sudo apt-get install libsdl2-dev`**); on macOS, if you use Homebrew you can run **`brew install sdl2`**. For other platforms, see the SDL website.

Using a Local Scenic Version with VerifAI

If you are using Scenic as part of the VerifAI toolkit, the VerifAI installation process will automatically install Scenic from PyPI. However, if you need to use your own fork of Scenic or some features which have not yet been released on PyPI, you will need to install Scenic manually in VerifAI's virtual environment. The easiest way to do this is as follows:

1. Install VerifAI in a virtual environment of your choice.
2. Activate the virtual environment.
3. Change directory to your clone of the Scenic repository.
4. Run **`pip install -e .`**

You can test that this process has worked correctly by going back to the VerifAI repo and running the Scenic part of its test suite with **`pytest tests/test_scenic.py`**.

Note: Installing Scenic in this way bypasses dependency resolution for VerifAI. If your local version of Scenic requires different versions of some of VerifAI's dependencies, you may get errors from **`pip`** about dependency conflicts. Such errors do not actually prevent Scenic from being installed; however you may get unexpected behavior from VerifAI at runtime. If you are developing forks of Scenic and VerifAI, a more stable approach would be to modify VerifAI's `pyproject.toml` to point to your fork of Scenic instead of the `scenic` package on PyPI.

1.2.2 MacOS

Installing python-fcl on Apple silicon

If on an Apple-silicon machine you get an error related to pip being unable to install `python-fcl`, it can be installed manually using the following steps:

1. Clone the python-fcl repository.
2. Navigate to the repository.
3. Install dependencies using Homebrew with the following command: **`brew install fcl eigen octomap`**
4. Activate your virtual environment if you haven't already.
5. Install the package using pip with the following command: **`CPATH=$(brew --prefix)/include:$(brew --prefix)/include/eigen3 LD_LIBRARY_PATH=$(brew --prefix)/lib python -m pip install .`**

1.2.3 Windows

Using WSL

For greatest ease of installation, we recommend using the Windows Subsystem for Linux (WSL, a.k.a. "Bash on Windows") on Windows 10 and newer.

Some WSL users have reported encountering the error `no display name and no $DISPLAY environmental variable`, but have had success applying the techniques outlined here.

It is possible to run Scenic natively on Windows; however, in the past there have been issues with some of Scenic's dependencies either not providing wheels for Windows or requiring manual installation of additional libraries.

Problems building Shapely

In the past, the `shapely` package did not install properly on Windows. If you encounter this issue, try installing it manually following the instructions here.

1.3 What's New in Scenic

This page describes what new features have been added in each version of Scenic, as well as any syntax changes which break backwards compatibility. Scenic uses semantic versioning, so a program written for Scenic 2.1 should also work in Scenic 2.5, but not necessarily in Scenic 3.0. You can run **`scenic --version`** to see which version of Scenic you are using.

1.3.1 Scenic 3.x

The Scenic 3.x series adds native support for 3D geometry, precise modeling of the shapes of objects, and temporal requirements. It also features a new parser enabling clearer error messages, greater language extensibility, and various improvements to the syntax.

See porting to Scenic 3 for tools to help migrate existing 2D scenarios.

Scenic 3.0.0

Backwards-incompatible syntax changes:

- Objects must be explicitly created using the *new* keyword, e.g. `new Object at (1, 2)` instead of the old `Object at (1, 2)`. This removes an ambiguity in the Scenic grammar, and makes non-creation uses of class names like `myClasses = [Car, Bicycle, Pedestrian]` clearer.
- *Monitor definitions* must include a parenthesized list of arguments, like behaviors: you should write `monitor MyMonitor():` for example instead of the old `monitor MyMonitor:`. Furthermore, monitors are no longer automatically enforced in the scenario where they are defined: you must explicitly instantiate them with the new *require monitor* statement.
- As the `heading` property is now derived from the 3D `orientation` (see below), it can no longer be set directly. Classes providing a default value for `heading` should instead provide a default value for `parentOrientation`. Code like `with heading 30 deg` should be replaced with the more idiomatic `facing 30 deg`.

Backwards-incompatible semantics changes:

- Objects are no longer required to be visible from the *ego* by default. (The `requireVisible` property is now *False* by default.)

- Visibility checks take occlusion into account by default (see below). The visible regions of objects are now 3D regions.
- Checking containment of objects in regions is now precise (previously, Scenic only checked if all of the corners of the object were contained in the region).
- While evaluating a precondition or invariant of a behavior or scenario, code that would cause the simulation to be rejected (such as sampling from an empty list) is now considered as simply making the precondition/invariant false.
- The *left of* Object specifier and its variants now correctly take into account the dimensions of both the object being created *and* the given object (the implementation previously did not account for the latter, despite the documentation saying otherwise).
- The offset by specifier now optionally specifies parentOrientation.

Backwards-incompatible API changes:

- The **maxIterations** argument of *Simulator.simulate* now has default value 1, rather than 100. A default value of 1 is the most reasonable in general since it means that when a simulation is rejected, a new scene will have to be generated (instead of trying many simulations from the same starting scene, which might well fail in the same way).
- For simulator interface writers: the *Simulator.createSimulation* and *Simulation* APIs have changed; initial creation of objects is now done automatically, and other initialization must be done in the new *Simulation.setup* method. See *scenic.core.simulators* for details.

Major new features:

- Scenic uses 3D geometry. Vectors now have 3 coordinates: if a third coordinate is not provided, it is assumed to be zero, so that scenarios taking place entirely in the z=0 plane will continue to work as before. Orientations of objects in space are represented by a new orientation property (internally a quaternion), which is computed by applying intrinsic yaw, pitch, and roll rotations, given by new properties by those names. These rotations are applied to the object's parentOrientation, which by default aligns with the Scenic global coordinate system but is optionally specified by *left of* and similar specifiers; this makes it easy to orient an object with respect to another object. See the relevant section of the *tutorial* for examples.
- Scenic models the precise shapes of objects, rather than simply using bounding boxes for collision detection and visibility checks. Objects have a new shape property (an instance of the *Shape* class) representing their shape; shapes can be created from standard 3D mesh formats such as STL.
- Visibility checks now take occlusion into account as well as precise shapes of objects. This is done using raytracing: the number of rays can be controlled on a per-object basis using viewRayDensity and related properties.
- The *require* statement accepts arbitrary properties in Linear Temporal Logic (not just the require always and require eventually forms previously allowed).
- Sampled *Scene* objects can now be serialized to short sequences of bytes and restored later. Similarly, executed *Simulation* objects can be saved and replayed. See *Storing Scenes/Simulations for Later Use* for details.
- Scenic syntax highlighters for Sublime Text, Visual Studio Code, and other TextMate-compatible editors are now available: see *Getting Started with Scenic*. For users of Pygments, the scenic package automatically installs a Pygments lexer (and associated style) for Scenic.

Minor new features:

- It is no longer necessary to define an *ego* object. If no *ego* is defined, the egoObject attribute of a sampled *Scene* is None.
- Syntax errors should now always indicate the correct part of the source code.

1.3.2 Scenic 2.x

The Scenic 2.x series is a major new version of Scenic which adds native support for dynamic scenarios, scenario composition, and more.

Scenic 2.1.0

Major new features:

- Modular scenarios and ways to compose them together, introduced as a prototype in 2.0.0, are now finalized, with many fixes and improvements. See *Composing Scenarios* for an overview of the new syntax.
- The `record` statement for recording values at every step of dynamic simulations (or only at the start/end).
- A built-in Newtonian physics simulator for debugging dynamic scenarios without having to install an external simulator (see `scenic.simulators.newtonian`).
- The interface to the Webots simulator has been greatly generalized, and now supports dynamic scenarios (see `scenic.simulators.webots`).

Minor new features:

- You can now write `require expr as name` to give a name to a requirement; similarly for `require always`, termination conditions, etc.
- Compatibility with Python 3.7 is restored. Scenic 2 now supports all versions of Python from 3.7 to 3.11.

Scenic 2.0.0

Backwards-incompatible syntax changes:

- The interval notation `(low, high)` for uniform distributions has been removed: use `Range(low, high)` instead. As a result of this change, the usual Python syntax for tuples is now legal in Scenic.
- The `height` property of *Object*, measuring its extent along the Y axis, has been renamed `length` to better match its intended use. The name `height` will be used again in a future version of Scenic with native support for 3D geometry.

Major new features:

- Scenic now supports writing and executing dynamic scenarios, where agents take actions over time according to behaviors specified in Scenic. See *Dynamic Scenarios* for an overview of the new syntax.
- An abstract *Driving Domain* allowing traffic scenarios to be written in a platform-agnostic way and executed in multiple simulators (in particular, both CARLA and LGSVL). This library includes functionality to parse road networks from standard formats (currently OpenDRIVE) and expose information about them for use in Scenic scenarios.
- A much generalized and improved interface to CARLA. (Many thanks to the CARLA team for contributing this.)
- An interface to the LGSVL driving simulator. (Many thanks to the LG team for helping develop this interface.)

Minor new features:

- Operators and specifiers which take vectors as arguments will now accept tuples and lists of length 2; for example, you can write `Object at (1, 2)`. The old syntax `Object at 1@2` is still supported.
- The `model` statement allows a scenario to specify which world model it uses, while being possible to override from the command line with the `--model` option.

- Global parameters can be overridden from the command line using the `--param` option (e.g. to specify a different map to use for a scenario).
- The unpacking operator `*` can now be used with `Uniform` to select a random element of a random list/tuple (e.g. `lane = Uniform(*network.lanes); sec = Uniform(*lane.sections)`).
- The Python built-in function `filter` is now supported, and can be used along with unpacking as above to select a random element of a random list satisfying a given condition (see *filter* for an example).

(Many other minor features didn't make it into this list.)

1.4 Scenic Fundamentals

This tutorial motivates and illustrates the main features of Scenic, focusing on aspects of the language that make it particularly well-suited for describing geometric scenarios. We begin by walking through Scenic's core features from first principles, using simple toy examples displayed in Scenic's built-in visualizer. We then consider discuss two case studies in depth: using Scenic to generate traffic scenes to test and train autonomous cars (as in [F22], [F19]), and testing a motion planning algorithm for a Mars rover able to climb over rocks. These examples show Scenic interfacing with actual simulators, and demonstrate how it can be applied to real problems.

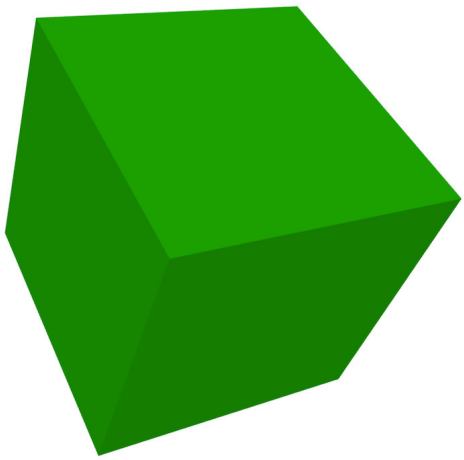
We'll focus here on the *spatial* aspects of scenarios; for adding *temporal* dynamics to a scenario, see our page on *Dynamic Scenarios*.

1.4.1 Objects, Geometry, and Specifiers

To start with, we'll construct a very basic Scenic program:

```
ego = new Object
```

Running this program should cause a window to pop up, looking like this:



You can rotate and move the camera of the visualizer around using the mouse. The only *Object* currently present is the one we created using the `new` keyword (rendered as a green box). Since we assigned this object to the *ego* name, it has special significance to Scenic, as we'll see later. For now it only has the effect of highlighting the object green in Scenic's visualizer. Pressing `w` will render all objects as wireframes, which will allow you to see the coordinate axes in the center of the ego object (at the origin).

Since we didn't provide any additional information to Scenic about this object, its **properties** like `position`, `orientation`, `width`, etc. were assigned default values from the object's class: here, the built-in class *Object*, representing a physical object. So we end up with a generic cube at the origin. To define the properties of an object, Scenic provides a flexible system of **specifiers** based on the many ways one can describe the position and orientation of an object in natural language. We can see a few of these specifiers in action in the following slightly more complex program (see the *Syntax Guide* for a summary of all the specifiers, and the *Specifiers Reference* for detailed definitions):

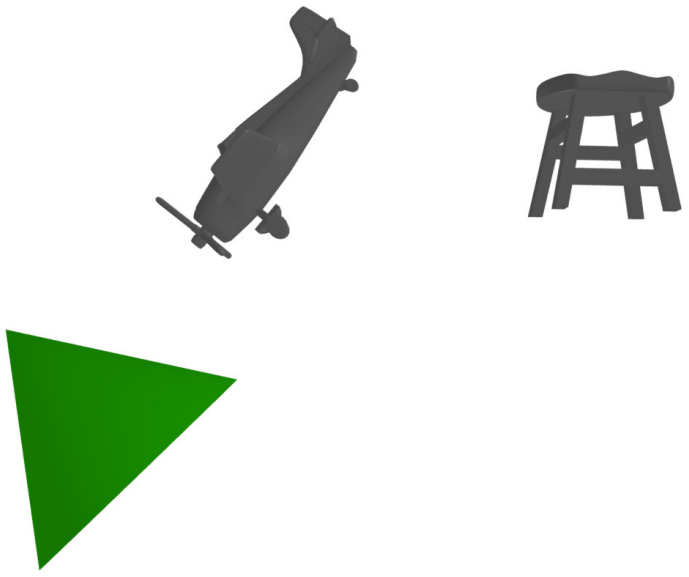
```
ego = new Object with shape ConeShape(),
        with width 2,
        with length 2,
        with height 1.5,
        facing (-90 deg, 45 deg, 0)

chair = new Object at (4,0,2),
            with shape MeshShape.fromFile(localPath("meshes/chair.obj"),
                initial_rotation=(0,90 deg,0), dimensions=(1,1,1))

plane_shape = MeshShape.fromFile(path=localPath("meshes/plane.obj"))

plane = new Object left of chair by 1,
            with shape plane_shape,
            with width 2,
            with length 2,
            with height 1,
            facing directly toward ego
```

This should generate the following scene:



The first object we create, the *ego*, has a cone shape. Scenic provides several built-in shapes like this (see *Shape* for a list). We then set the object's dimensions using the `with` specifier, which can set any property (even properties not built into Scenic, which you might access in your own code or which a particular simulator might understand). Finally, we set the object's global orientation (its `orientation` property) using the `facing` specifier. The tuple after `facing` contains the Euler angles of the desired orientation (yaw, pitch, roll).

The second object we create is first placed at a specific point in space using the `at` specifier (setting the object's

position property). We then set its shape to one imported from a mesh file, using the *MeshShape* class, applying an initial rotation to tell Scenic which side of the chair is its front. We also set default dimensions of the shape, which the object will then automatically inherit. If we hadn't set these default dimensions, Scenic would automatically infer the dimensions from the mesh file.

On line 11 we load a shape from a file, specifically to highlight that since Scenic is built on top of Python, we can write arbitrary Python expressions in Scenic (with some exceptions).

For our third and final object, we use the `left of` specifier to place it to the left of `chair` (the second object) by 1 unit. We set its shape and dimensions, similar to before, and then orient it to face directly toward the ego object using the `facing directly toward` specifier. This gives a first hint of the power of specifiers, with Scenic automatically working out how to compute the object's `orientation` so that it faces the *ego* regardless of how we specified its `position` (in fact, we could move the `left of` specifier to be after the `facing directly toward` and the code would still work).

Scenic will automatically reject scenarios that don't make physical sense, for instance when objects intersect each other[1]. For an example of this, try changing the code above to have a much larger ego object, to the point where it would intersect with the plane. While this isn't too important in the scenarios we've seen so far, it becomes very useful when we start constructing *random* scenarios.

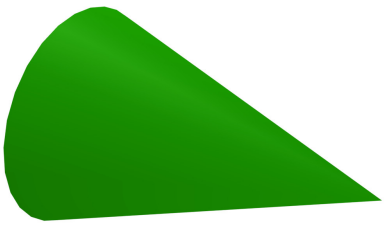
1.4.2 Randomness and Regions

So far all of our Scenic programs have defined concrete scenes, i.e. they uniquely define all the aspects of a scene, so every time we run the program we'll get the same scene. This is because so far we haven't introduced any *randomness*. Scenic is a *probabilistic programming language*, meaning a single Scenic program can in fact define a probability distribution over many possible scenes.

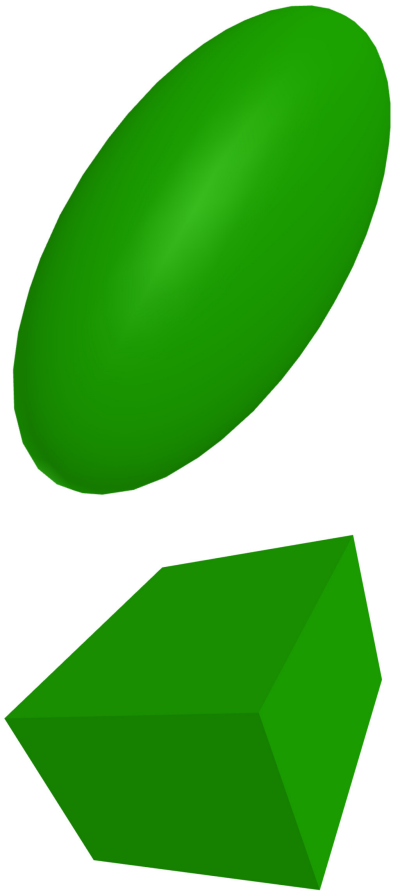
Let's look at a simple Scenic program with some random elements:

```
ego = new Object with shape Uniform(BoxShape(), SpheroidShape(), ConeShape()),
                 with width Range(1,2),
                 with length Range(1,2),
                 with height Range(1,3),
                 facing (Range(0,360) deg, Range(0,360) deg, Range(0,360) deg)
```

This will generate an object with a shape that is either a box, a spheroid, or a cone (each with equal probability). It will have a random width, length, and height within the ranges specified, and uniformly random rotation angles. Some examples:



[1] Although collisions can be allowed on a per-object basis: see the `allowCollisions` property of *Object*.



Random values can be used almost everywhere in Scenic; the major exception is that control flow (e.g. `if` statements and `for` loops) cannot depend on random values. This restriction enables more efficient sampling (see [F19]) and can often be worked around: for example it is still possible to select random elements satisfying desired criteria from lists (see *filter*).

Another key construct in Scenic is a *Region*, which represents a set of points in space. Having defined a region of interest, for example a lane of a road, you can then sample points from it, check whether objects are contained in it, etc. You can also use a region to define the **workspace**, a designated region which all objects in the scenario must be contained in (useful, for example, if the simulated world has fixed obstacles that Scenic objects should not intersect). For example, the following code:

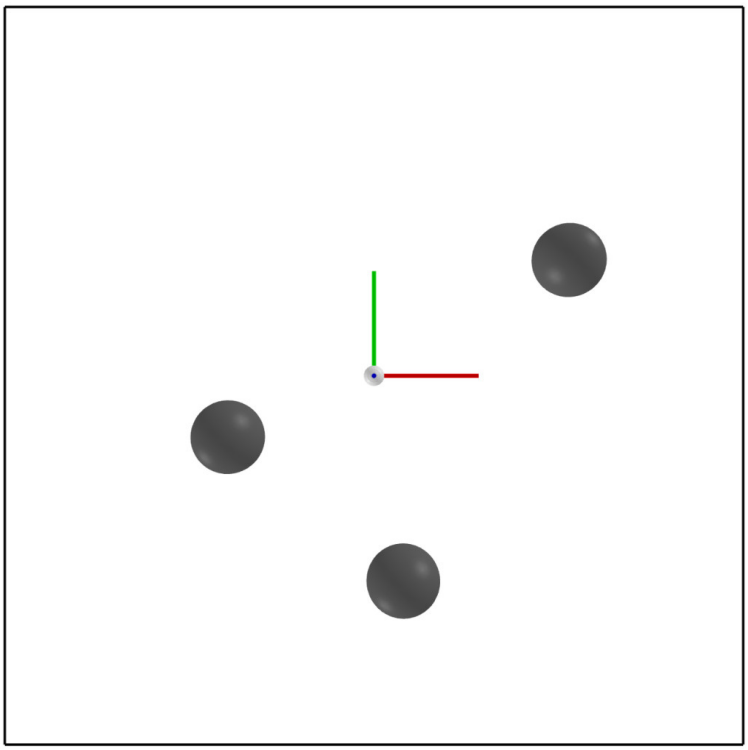
```
region = RectangularRegion((0,0,0), 0, 10, 10)
workspace = Workspace(region)

new Object in region, with shape SpheroidShape()
new Object in region, with shape SpheroidShape()
new Object in region, with shape SpheroidShape()
```

should generate a scene similar to this:

Note that in this scene the coordinate axes in the center are displayed due to the **`--axes`** flag, which can help clarify orientation.

We first create a 10-unit square *`RectangularRegion`*, and set it as the scenario's workspace. *`RectangularRegion`* is a 2D region, meaning it does not have a volume and therefore can't really contain objects. It is still a valid workspace, however, since for containment checks involving 2D regions, Scenic automatically uses the region's *footprint*, which extends infinitely in the positive and negative Z directions. We then create 3 spherical objects and place them using the



in specifier, which sets the position of an object (its center) to a uniformly-random point in the given region.

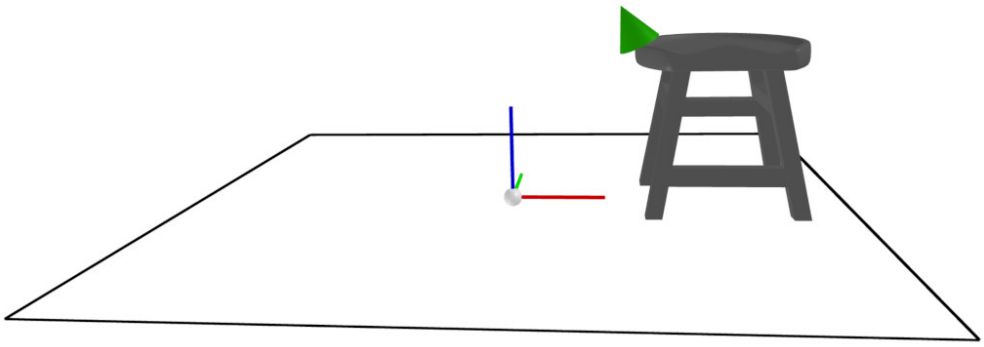
Similarly, we can use the on specifier to place the *base* of an object uniformly at random in a region, where the base is by default the center of the bottom side of its bounding box. The on specifier is also overloaded to work on objects, by default extracting the top surface of the object's mesh and placing the object on that. This can lead to very compact syntax for randomly placing objects on others, as seen in the following example:

```
workspace = Workspace(RectangularRegion((0,0,0), 0, 4, 4))
floor = workspace

chair = new Object on floor,
            with shape MeshShape.fromFile(path=localPath("meshes/chair.obj"),
                dimensions=(1,1,1), initial_rotation=(0, 90 deg, 0))

ego = new Object on chair,
            with shape ConeShape(dimensions=(0.25,0.25,0.25))
```

which might generate something like this:



1.4.3 Orientations in Depth

Notice how in the last example the cone is oriented to be tangent with the curved surface of the chair, even though we never set an orientation with `facing`. To explain this behavior, we need to look deeper into Scenic's orientation system. All objects have an `orientation` property, which is their orientation in *global* coordinates[2]. If you just want to set the orientation by giving explicit angles in global coordinates, you can use the `facing` specifier as we saw above. However, it's often useful to specify the orientation of an object in terms of *some other* coordinate system, for instance that of another object. To support such use cases, Scenic does not allow directly setting the value of `orientation` using `with`: instead, its value is *derived* from the values of 4 other properties, `parentOrientation`, `yaw`, `pitch`, and `roll`. The `parentOrientation` property defines the **parent orientation** of the object, which is the orientation with respect to which the (intrinsic Euler) angles `yaw`, `pitch`, and `roll` are interpreted. Specifically, `orientation` is obtained as follows:

1. start from `parentOrientation`;
2. apply a yaw (a CCW (counter-clockwise) rotation around the positive Z axis) of `yaw`;
3. apply a pitch (a CCW rotation around the resulting positive X axis) of `pitch`;
4. apply a roll (a CCW rotation around the resulting positive Y axis) of `roll`.

By default, `parentOrientation` is aligned with the global coordinate system, so that `yaw` for example is just the angle by which to rotate the object around the Z axis (this corresponds to the `heading` property in older versions of Scenic). But by setting `parentOrientation` to the `orientation` of another object, we can easily compose rotations together: "face the same way as the plane, but upside-down" could be implemented `with parentOrientation plane.orientation, with roll 180 deg`.

In fact it is often unnecessary to set `parentOrientation` yourself, since many of Scenic's specifiers do so automatically when there is a natural choice of orientation to use. This includes all specifiers which position one object in terms of another: if we write `new Object ahead of plane by 100`, the `ahead of` specifier specifies `position` to be 100 meters ahead of the plane but *also* specifies `parentOrientation` to be `plane.orientation`. So by default the new object will be oriented the same way as the plane; to implement the "upside-down" part, we could simply write `new Object ahead of plane by 100, with roll 180 deg`. Importantly, the `ahead of` specifier here only specifies `parentOrientation` *optionally*, giving it a new default value: if you want a different value, you can override that default by explicitly writing `with parentOrientation value`. (We'll return to how Scenic manages default values and "optional" specifications later.)

Another case where a specifier sets `parentOrientation` automatically is our cone-on-a-chair example above: in the code `new Object on chair`, the `on` specifier not only specifies `position` to be a random point on the top surface

[2] Represented as an instance of the *Orientation* class, which internally uses quaternions (although you shouldn't need to worry about that). In the rare case where you need to manipulate orientations beyond what Scenic's operators provide, see the documentation for *Orientation*.

of the chair but also specifies `parentOrientation` to be an orientation tangent to the surface at that point. Thus the cone lies flat on the surface by default without our needing to specify its orientation; we could even add code like `with roll 45 deg` to rotate the cone while keeping it tangent with the surface.

In general, the `on region` specifier specifies `parentOrientation` whenever the region in question has a preferred orientation: a *Vector Field* (another primitive Scenic type) which defines an orientation at each point in the region. The class `MeshSurfaceRegion`, used to represent surfaces of an object, has a default preferred orientation which is tangent to the surface, allowing us to easily place objects on irregular surfaces as we've seen. Preferred orientations can also be convenient for modeling the nominal driving direction on roads, for example (we'll return to this use case below).

1.4.4 Points, Oriented Points, and Classes

We've seen that Scenic has a built-in class *Object* for representing physical objects, and that individual objects are instantiated using the `new` keyword. *Object* is actually the bottom class in a hierarchy of built-in Scenic classes that support this syntax: its superclass is *OrientedPoint*, whose superclass in turn is *Point*. The base class *Point* provides the `position` property, while its subclass *OrientedPoint* adds `orientation` (plus `parentOrientation`, `yaw`, etc.). These two classes do not represent physical objects and aren't included in scenes generated by Scenic, but they provide a convenient way to use specifier syntax to construct positions and orientations for later use without creating actual objects. A *Point* can be used anywhere where a vector is expected (e.g. `at point`), and an *OrientedPoint* can also be used anywhere where an orientation is expected. With both a position and an orientation, an *OrientedPoint* defines a local coordinate system, and so can be used with specifiers like `ahead of` to position objects:

```
spot = new OrientedPoint on curb
new Object left of spot by 0.25
```

Here, suppose `curb` is a region with a preferred orientation aligned with the plane of the road and along the curb; then the first line creates an *OrientedPoint* at a uniformly-random position on the curb, oriented along the curb. So the second line then creates an *Object* offset 0.25 meters into the road, regardless of which direction the road happens to run in the global coordinate system.

Scenic also allows users to define their own classes. In our earlier example placing spheres in a region, we explicitly wrote out the specifiers for each object we created even though they were all identical. Such repetition can often be avoided by using functions and loops, and by defining a class of object providing new default values for properties of interest. Our example could be equivalently written:

```
workspace = Workspace(RectangularRegion((0,0,0), 0, 10, 10))

class SphereObject:
    position: new Point in workspace
    shape: SpheroidShape()

for i in range(3):
    new SphereObject
```

Here we define the `SphereObject` class, providing new default values for the `position` and `shape` properties, overriding those inherited from *Object* (the default superclass if none is explicitly given). So for example the default `position` for a `SphereObject` is the expression `new Point in workspace`, which creates a *Point* that can be automatically interpreted as a position. This gives us a way to get the convenience of specifiers in class definitions. Note that this is a random expression, and it is evaluated independently each time a `SphereObject` is defined; so the loop creates 3 objects which will all have different positions (and as usual Scenic will ensure they do not overlap). We can still override the default value as needed: adding the line `new SphereObject at (0,0,5)` would create a `SphereObject` which still used the default value of `shape` but whose `position` is exactly `(0,0,5)`.

In addition to the special syntax seen above for defining properties of a class and instantiating an instance of a class, Scenic classes support inheritance and methods in the same way as Python:

```
class Vehicle:
    pass
class Taxicab(Vehicle):
    magicNumber: 42

    def myMethod(self, x):
        return self.width + self.magicNumber + x

ego = new Taxicab with magicNumber 1729
y = ego.myMethod(3.14)
```

1.4.5 Models and Simulators

For the next part of this tutorial, we'll move beyond the internal Scenic visualizer to an actual simulator. Specifically, we will consider examples from our case study using Scenic to generate traffic scenes in GTA V to test and train autonomous cars ([F19], [F22]).

To start, suppose we want scenes of one car viewed from another on the road. We can write this very concisely in Scenic:

```
1 from scenic.simulators.gta.model import Car
2 ego = new Car
3 new Car
```

Line 1 imports the GTA world model, a Scenic library defining everything specific to our GTA interface. This includes the definition of the class `Car`, as well as information about the road geometry that we'll see later. We'll suppress this `import` statement in subsequent examples.

Line 2 then creates a `Car` and assigns it to the special variable `ego` specifying the *ego object*, which we've seen before. This is the reference point for the scenario: our simulator interfaces typically use it as the viewpoint for rendering images, and many of Scenic's geometric operators use `ego` by default when a position is left implicit[3].

Finally, line 3 creates a second `Car`. Compiling this scenario with Scenic, sampling a scene from it, and importing the scene into GTA V yields an image like this:

Note that both the `ego` car (where the camera is located) and the second car are both located on the road and facing along it, despite the fact that the code above does not specify the position or any other properties of the two cars. This is because reasonable default values for these properties have already been defined in the `Car` definition (shown here slightly simplified):

```
1 class Car:
2     position: new Point on road
3     heading: roadDirection at self.position    # note: can only set `heading` in 2D mode
4     width: self.model.width
5     length: self.model.length
6     model: CarModel.defaultModel()      # a distribution over several car models
7     requireVisible: True    # so all cars appear in the rendered images
```

Here `road` is a region defined in the `gta` model to specify which points in the workspace are on a road. Similarly, `roadDirection` is a *Vector Field* specifying the nominal traffic direction at such points. The operator `F at X` simply

[3] In fact, since `ego` is a variable and can be reassigned, we can set `ego` to one object, build a part of the scene around it, then reassign `ego` and build another part of the scene.

Fig. 1: A scene sampled from the simple car scenario, rendered in GTA V.

gets the direction of the field *F* at point *X*, so line 3 sets a `Car`'s default heading to be the road direction at its `position`. The default `position`, in turn, is a `new Point on road`, which means a uniformly random point on the road. Thus, in our simple scenario above both cars will be placed on the road facing a reasonable direction, without our having to specify this explicitly.

One further point of interest in the code above is that the default value for `heading` depends on the value of `position`, and the default values of `width` and `length` depend on `model`. Scenic allows default value expressions to use the special syntax `self.property` to refer to the value of another property of the object being defined: Scenic tracks the resulting dependencies and evaluates the expressions in an appropriate order (or raises an error if there are any cyclic dependencies). This capability is also frequently used by specifiers, as we explain next.

1.4.6 Specifiers in Depth

Why Specifiers?

The syntax `left of X` and `facing Y` for specifying positions and orientations may seem unusual compared to typical constructors in object-oriented languages. There are two reasons why Scenic uses this kind of syntax: first, readability. The second is more subtle and based on the fact that in natural language there are many ways to specify positions and other properties, some of which interact with each other. Consider the following ways one might describe the location of a car:

1. "is at position *X*" (an absolute position)
2. "is just left of position *X*" (a position based on orientation)
3. "is 3 m West of the taxi" (a relative position)
4. "is 3 m left of the taxi" (a local coordinate system)
5. "is one lane left of the taxi" (another local coordinate system)
6. "appears to be 10 m behind the taxi" (relative to the line of sight)
7. "is 10 m along the road from the taxi" (following a potentially-curving vector field)

These are all fundamentally different from each other: for example, (4) and (5) differ if the taxi is not parallel to the lane.

Furthermore, these specifications combine other properties of the object in different ways: to place the object "just left of" a position, we must first know the object's `orientation`; whereas if we wanted to face the object "towards" a location, we must instead know its `position`. There can be chains of such *dependencies*: for example, the description "the car is 0.5 m left of the curb" means that the *right edge* of the car is 0.5 m away from the curb, not its center, which is what the car's `position` property stores. So the car's `position` depends on its `width`, which in turn depends on its `model`. In a typical object-oriented language, these dependencies might be handled by first computing values for `position` and all other properties, then passing them to a constructor. For "a car is 0.5 m left of the curb" we might write something like:

```
# hypothetical Python-like language (not Scenic)
model = Car.defaultModelDistribution.sample()
pos = curb.offsetLeft(0.5 + model.width / 2)
car = Car(pos, model=model)
```

Notice how `model` must be used twice, because `model` determines both the model of the car and (indirectly) its position. This is inelegant, and breaks encapsulation because the default model distribution is used outside of the `Car` constructor. The latter problem could be fixed by having a specialized constructor or factory function:

```
# hypothetical Python-like language (not Scenic)
car = CarLeftOfBy(curb, 0.5)
```

However, such functions would proliferate since we would need to handle all possible combinations of ways to specify different properties (e.g. do we want to require a specific model? Are we overriding the width provided by the model for this specific car?). Instead of having a multitude of such monolithic constructors, Scenic uses specifiers to factor the definition of objects into potentially-interacting but syntactically-independent parts:

```
new Car left of curb by 0.5,
        with model CarModel.models['BUS']
```

Here the specifiers `left of` *`X`* `by` *`D`* and `with model` *`M`* do not have an order, but *together* specify the properties of the car. Scenic works out the dependencies between properties (here, `position` is provided by `left of`, which depends on `width`, whose default value depends on `model`) and evaluates them in the correct order. To use the default model distribution we would simply omit line 2; keeping it affects the `position` of the car appropriately without having to specify `BUS` more than once.

Dependencies and Modifying Specifiers

In addition to explicit dependencies when one specifier uses a property defined by another, Scenic also tracks dependencies which arise when an expression implicitly refers to the properties of the object being defined. For example, suppose we wanted to elaborate the scenario above by saying the car is oriented up to 5° off of the nominal traffic direction. We can write this using the `roadDirection` vector field and Scenic's general operator *`X`* `relative to` *`Y`*, which can interpret vectors and orientations as being in a variety of local coordinate systems:

```
new Car left of curb by 0.5,
        facing Range(-5, 5) deg relative to roadDirection
```

Notice that since `roadDirection` is a vector field, it defines a different local coordinate system at each point in space: at different points on the map, roads point different directions! Thus an expression like `15 deg relative to field` does not define a unique heading. The example above works because Scenic knows that the expression `Range(-5, 5) deg relative to roadDirection` depends on a reference position, and automatically uses the `position` of the `Car` being defined.

Another kind of dependency arises from **modifying specifiers**, which are specifiers that can take an *already-specified* value for a property and modify it (thereby in a sense both depending on that property and specifying it). The main example is the `on region` specifier, which in addition to the usage we saw above for placing an object randomly within a region, also can be used as a modifying specifier: if the `position` property has already been specified, then `on region` *projects* that position onto the region. So for example the code `new Object ahead of plane by 100, on ground` does not raise an error even though both `ahead of` and `on` specify `position`: Scenic first computes a position 100 m ahead of the plane, and then projects that position down onto the ground.

Specifier Priorities

As we've discussed previously, specifiers can specify multiple properties, and they can specify some properties *optionally*, allowing other specifiers to override them. In fact, when a specifier specifies a property it does so with a **priority** represented by a positive integer. A property specified with priority 1 cannot be overridden; increasingly large integers represent lower priorities, so a priority-2 specifier overrides one with priority 3. This system enables more-specific specifiers to naturally take precedence over more general specifiers while reducing the amount of boilerplate code you need to write. Consider for example the following sequence of object creations, where we provide progressively more information about the object:

- In `new Object ahead of plane by 100`, the `ahead of` specifier specifies `parentOrientation` with priority 3, so that the new object is aligned with the plane (a reasonable default since we're positioning the object with respect to the plane).
- In `new Object ahead of plane by 100, on ground`, the `on ground` specifies `parentOrientation` with priority 2, so it takes precedence and the object is aligned with the ground rather than the plane (which makes more sense since "on ground" implies the object likely lies flat on the ground).
- Finally, in `new Object ahead of plane by 100, on ground, with parentOrientation (0, 90 deg, 0)`, the `with` specifier specifies `parentOrientation` with priority 1, so it takes precedence and Scenic uses the explicit orientation the user provided.

As these examples show, specifier priorities enable concise specifications of objects to have intuitive default behavior when no explicit information is given, while at the same time overriding this behavior remains straightforward.

For a more thorough look at the specifier system, including which specifiers specify which properties and at which priorities, consult the *Specifiers Reference*.

1.4.7 Declarative Hard and Soft Constraints

Notice that in the scenarios above we never explicitly ensured that two cars will not intersect each other. Despite this, Scenic will never generate such scenes. This is because Scenic enforces several *default requirements*, as mentioned above:

- All objects must be contained in the workspace, or a particular specified region (its container). For example, we can define the `Car` class so that all of its instances must be contained in the region `road` by default.
- Objects must not intersect each other (unless explicitly allowed).

Scenic also allows the user to define custom requirements checking arbitrary conditions built from various geometric predicates. For example, the following scenario produces a car headed roughly towards the camera, while still facing the nominal road direction:

```
ego = new Car on road
car2 = new Car offset by (Range(-10, 10), Range(20, 40)), with viewAngle 30 deg
require car2 can see ego
```

Here we have used the `X can see Y` predicate, which in this case is checking that the ego car is inside the 30° view cone of the second car.

Requirements, called *observations* in other probabilistic programming languages, are very convenient for defining scenarios because they make it easy to restrict attention to particular cases of interest. Note how difficult it would be to write the scenario above without the `require` statement: when defining the ego car, we would have to somehow specify those positions where it is possible to put a roughly-oncoming car 20–40 meters ahead (for example, this is not possible on a one-way road). Instead, we can simply place *ego* uniformly over all roads and let Scenic work out how to condition the distribution so that the requirement is satisfied[4]. As this example illustrates, the ability to declaratively impose constraints gives Scenic greater versatility than purely-generative formalisms. Requirements also improve encapsulation by allowing us to restrict an existing scenario without altering it. For example:

```
from myScenarioLib import genericTaxiScenario    # import another Scenic scenario
fifthAvenue = ...              # extract a Region from a map here
require genericTaxiScenario.taxi in fifthAvenue
```

The constraints in our examples above are *hard requirements* which must always be satisfied. Scenic also allows imposing *soft requirements* that need only be true with some minimum probability:

```
require[0.5] car2 can see ego   # condition only needs to hold with prob. >= 0.5
```

Such requirements can be useful, for example, in ensuring adequate representation of a particular condition when generating a training set: for instance, we could require that at least 90% of generated images have a car driving on the right side of the road.

1.4.8 Mutations

A common testing paradigm is to randomly generate *variations* of existing tests. Scenic supports this paradigm by providing syntax for performing mutations in a compositional manner, adding variety to a scenario without changing its code. For example, given a complex scenario involving a taxi, we can add one additional line:

```
from bigScenario import taxi
mutate taxi
```

The `mutate` statement will add Gaussian noise to the `position` and `orientation` properties of `taxi`, while still enforcing all built-in and custom requirements. The standard deviation of the noise can be scaled by writing, for example, `mutate taxi by 2` (which adds twice as much noise), and in fact can be controlled separately for `position` and `orientation` (see *scenic.core.object_types.Mutator*).

1.4.9 A Worked Example

We conclude with a larger example of a Scenic program which also illustrates the language's utility across domains and simulators. Specifically, we consider the problem of testing a motion planning algorithm for a Mars rover able to climb over hills and rocks. Such robots can have very complex dynamics, with the feasibility of a motion plan depending on exact details of the robot's hardware and the geometry of the terrain. We can use Scenic to write a scenario generating challenging cases for a planner to solve in simulation. Some of the specifiers and operators we'll use have not been discussed before in the tutorial; as usual, information about them can be found in the *Syntax Guide*.

We will write a scenario representing a hilly field of rocks and pipes with a bottleneck between the rover and its goal that forces the path planner to consider climbing over a rock. First, we import a small Scenic library for the Webots robotics simulator and a mars specific library which defines the (empty) workspace and several types of objects: the `Rover` itself, the `Goal` (represented by a flag), the `MarsGround` and `MarsHill` classes which are used to create the hilly terrain, and debris classes `Rock`, `BigRock`, and `Pipe`. `Rock` and `BigRock` have fixed sizes, and the rover can

[4] On the other hand, Scenic may have to work hard to satisfy difficult constraints. Ultimately Scenic falls back on rejection sampling, which in the worst case will run forever if the constraints are inconsistent (although we impose a limit on the number of iterations: see *Scenario.generate*).

climb over them; `Pipe` cannot be climbed over, and can represent a pipe of arbitrary length, controlled by the `length` property (which corresponds to Scenic's *Y* axis).

```
model scenic.simulators.webots.mars.model
from mars_lib import *
```

Here we've used the *model* statement to select the world model for the scenario: it is equivalent to `from scenic.simulators.webots.model import *` except that the choice of model can be overridden from the command line when compiling the scenario (using the *--model* option). This is useful for scenarios that use one of Scenic's *Abstract Domains*: the scenario can be written once in a simulator-agnostic manner, then used with different simulators by selecting the appropriate simulator-specific world model.

Now we can start to create objects. The first object we will create will be the hilly ground. To do this, we use the `MarsGround` which has a `terrain` property which should be set to a collection of `MarsHill` classes, each of which adds a gaussian hill to the ground. Note that the `MarsGround` object has `allowCollisions` set to `True`, allowing objects to intersect and be slightly embedded in the ground. In the following code we create a ground object with 60 small hills (which are allowed to stack on top of each other):

```
ground = new MarsGround on (0,0,0), with terrain [new MarsHill for _ in range(60)]
```

We next create the rover at a fixed position and the goal at a random position on the other side of the workspace, ensuring both are on the ground:

```
ego = new Rover at (0, -3), on ground, with controller 'sojourner'
goal = new Goal at (Range(-2, 2), Range(2, 3)), on ground, facing (0,0,0)
```

Next we pick a position for the bottleneck, requiring it to lie roughly on the way from the robot to its goal, and place a rock there. Here we use the simple form of `facing` which takes a scalar argument, effectively setting the yaw of the object in the global coordinate system (so that `0 deg` is due North, for example, and `90 deg` is due West).

```
bottleneck = new OrientedPoint at ego offset by Range(-1.5, 1.5) @ Range(0.5, 1.5),␣
↪facing Range(-30, 30) deg
require abs((angle to goal) - (angle to bottleneck)) <= 10 deg
new BigRock at bottleneck, on ground
```

Note how we define `bottleneck` as an *OrientedPoint*, with a range of possible orientations: this is to set up a local coordinate system for positioning the pipes making up the bottleneck. Specifically, we position two pipes of varying lengths on either side of the bottleneck, projected onto the ground, with their ends far enough apart for the robot to be able to pass between. Note that we explicitly specify `parentOrientation` to be the global coordinate system, which prevents the pipes from lying tangent to the ground as we want them flat and partially embedded in the ground.

```
gap = 1.2 * ego.width
halfGap = gap / 2

leftEdge = new OrientedPoint left of bottleneck by halfGap,
    facing Range(60, 120) deg relative to bottleneck.heading
rightEdge = new OrientedPoint right of bottleneck by halfGap,
    facing Range(-120, -60) deg relative to bottleneck.heading

new Pipe ahead of leftEdge, with length Range(1, 2), on ground, facing leftEdge, with␣
↪parentOrientation 0
new Pipe ahead of rightEdge, with length Range(1, 2), on ground, facing rightEdge, with␣
↪parentOrientation 0
```

Finally, to make the scenario slightly more interesting, we add several additional obstacles, positioned either on the far side of the bottleneck or anywhere at random (recalling that Scenic automatically ensures that no objects will overlap).

```
new Pipe on ground, with parentOrientation 0
new BigRock beyond bottleneck by Range(0.25, 0.75) @ Range(0.75, 1), on ground
new BigRock beyond bottleneck by Range(-0.75, -0.25) @ Range(0.75, 1), on ground
new Rock on ground
new Rock on ground
new Rock on ground
```

This completes the scenario, which can also be found in the Scenic repository under `examples/webots/mars/narrowGoal.scenic`. Scenes generated from the scenario, and visualized in Scenic's internal visualizer and Webots, are shown below.

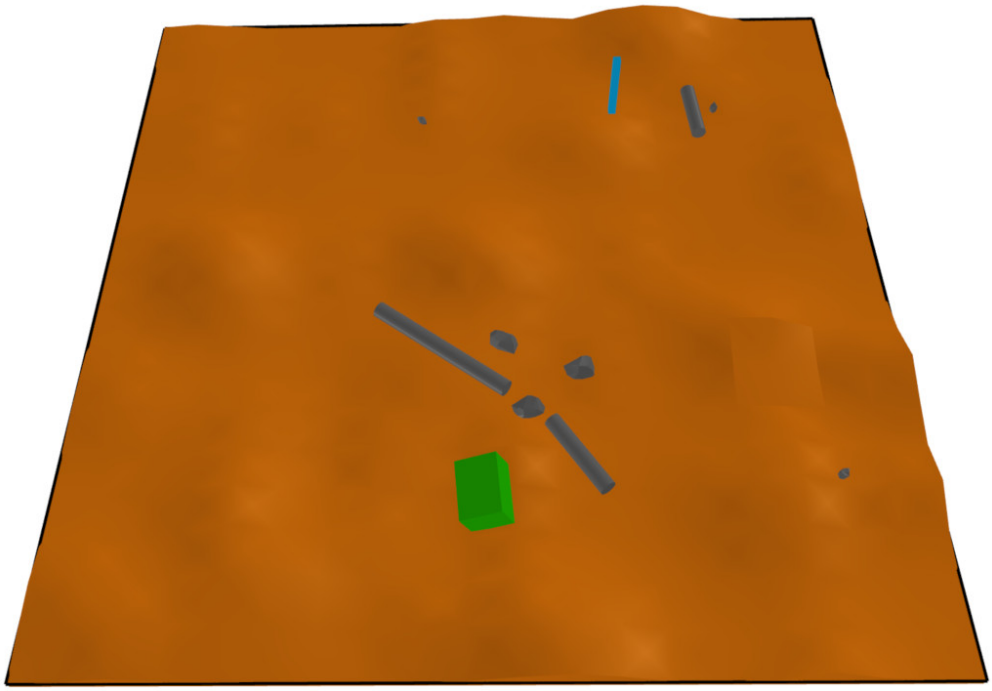


Fig. 2: A scene sampled from the Mars rover scenario, rendered in Scenic's internal visualizer.

1.4.10 Further Reading

This tutorial illustrated the syntax of Scenic through several simple examples. Much more complex scenarios are possible, such as the platoon and bumper-to-bumper traffic GTA V scenarios shown below. For many further examples using a variety of simulators, see the `examples` folder, as well as the links in the *Supported Simulators* page.

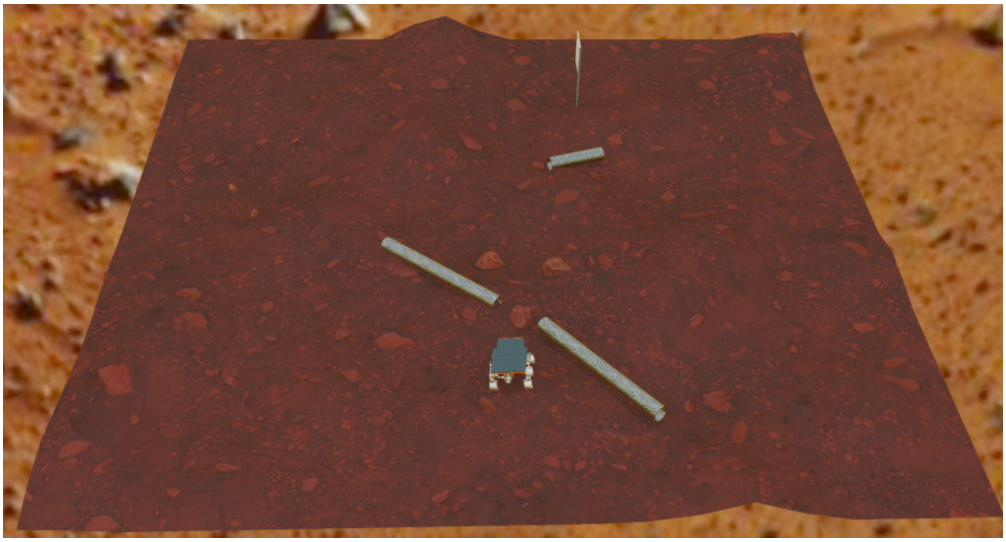


Fig. 3: A scene sampled from the Mars rover scenario, rendered in Webots.

Our tutorial on *Dynamic Scenarios* describes how to define scenarios with dynamic agents that move or take other actions over time. We also have a tutorial on *Composing Scenarios*: defining scenarios in a modular, reusable way and combining them to build up more complex scenarios.

For a comprehensive overview of Scenic's syntax, including details on all specifiers, operators, distributions, statements, and built-in classes, see the *Language Reference*. Our *Syntax Guide* summarizes all of these language constructs in convenient tables with links to the detailed documentation.

References

1.5 Dynamic Scenarios

The *Scenic Fundamentals* described how Scenic can model scenarios like "a badly-parked car" by defining spatial relationships between objects. Here, we'll cover how to model *temporal* aspects of scenarios: for a scenario like "a badly-parked car, which pulls into the road as the ego car approaches", we need to specify not only the initial position of the car but how it behaves over time.

1.5.1 Agents, Actions, and Behaviors

In Scenic, we call objects which take actions over time *dynamic agents*, or simply *agents*. These are ordinary Scenic objects, so we can still use all of Scenic's syntax for describing their initial positions, orientations, etc. In addition, we specify their dynamic behavior using a built-in property called `behavior`. Here's an example using one of the built-in behaviors from the *Driving Domain*:

```
model scenic.domains.driving.model
new Car with behavior FollowLaneBehavior
```

A behavior defines a sequence of *actions* for the agent to take, which need not be fixed but can be probabilistic and depend on the state of the agent or other objects. In Scenic, an action is an instantaneous operation executed by an agent, like setting the steering angle of a car or turning on its headlights. Most actions are specific to particular application domains, and so different sets of actions are provided by different simulator interfaces. For example, the *Driving Domain* defines a `SetThrottleAction` for cars.

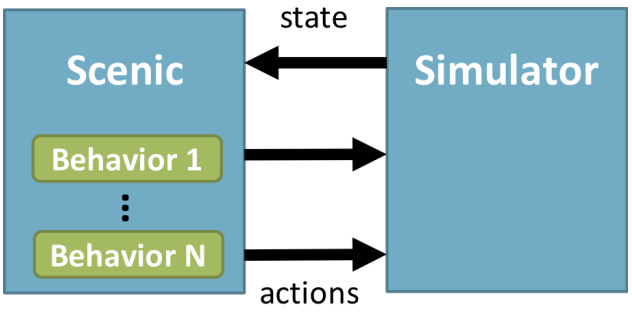
To define a behavior, we write a function which runs over the course of the scenario, periodically issuing actions. Scenic uses a discrete notion of time, so at each time step the function specifies zero or more actions for the agent to take. For example, here is a very simplified version of the `FollowLaneBehavior` above:

```
behavior FollowLaneBehavior():
    while True:
        throttle, steering = ...    # compute controls
        take SetThrottleAction(throttle), SetSteerAction(steering)
```

We intend this behavior to run for the entire scenario, so we use an infinite loop. In each step of the loop, we compute appropriate throttle and steering controls, then use the `take` statement to take the corresponding actions. When that

statement is executed, Scenic pauses the behavior until the next time step of the simulation, when the function resumes and the loop repeats.

When there are multiple agents, all of their behaviors run in parallel; each time step, Scenic sends their selected actions to the simulator to be executed and advances the simulation by one step. It then reads back the state of the simulation, updating the positions and other dynamic properties of the objects.



Behaviors can access the current state of the world to decide what actions to take:

```
behavior WaitUntilClose(threshold=15):
    while (distance from self to ego) > threshold:
        wait
    do FollowLaneBehavior()
```

Here, we repeatedly query the distance from the agent running the behavior (*self*) to the ego car; as long as it is above a threshold, we *wait*, which means take no actions. Once the threshold is met, we start driving by invoking the `FollowLaneBehavior` we saw above using the *do* statement. Since `FollowLaneBehavior` runs forever, we will never return to the `WaitUntilClose` behavior.

The example above also shows how behaviors may take arguments, like any Scenic function. Here, `threshold` is an argument to the behavior which has default value 15 but can be customized, so we could write for example:

```
ego = new Car
car2 = new Car visible, with behavior WaitUntilClose
car3 = new Car visible, with behavior WaitUntilClose(20)
```

Both `car2` and `car3` will use the `WaitUntilClose` behavior, but independent copies of it with thresholds of 15 and 20 respectively.

Unlike ordinary Scenic code, control flow constructs such as `if` and `while` are allowed to depend on random variables inside a behavior. Any distributions defined inside a behavior are sampled at simulation time, not during scene sampling. Consider the following behavior:

```
1 behavior Foo():
2     threshold = Range(4, 7)
3     while True:
4         if self.distanceToClosest(Pedestrian) < threshold:
5             strength = TruncatedNormal(0.8, 0.02, 0.5, 1)
6             take SetBrakeAction(strength), SetThrottleAction(0)
7         else:
8             take SetThrottleAction(0.5), SetBrakeAction(0)
```

Here, the value of `threshold` is sampled only once, at the beginning of the scenario when the behavior starts running. The value `strength`, on the other hand, is sampled every time control reaches line 5, so that every time step when the

car is braking we use a slightly different braking strength (0.8 on average, but with Gaussian noise added with standard deviation 0.02, truncating the possible values to between 0.5 and 1).

1.5.2 Interrupts

It is frequently useful to take an existing behavior and add a complication to it; for example, suppose we want a car that follows a lane, stopping whenever it encounters an obstacle. Scenic provides a concept of *interrupts* which allows us to reuse the basic `FollowLaneBehavior` without having to modify it:

```
behavior FollowAvoidingObstacles():
    try:
        do FollowLaneBehavior()
    interrupt when self.distanceToClosest(Object) < 5:
        take SetBrakeAction(1)
```

This *try-interrupt* statement has similar syntax to the Python try statement (and in fact allows `except` clauses just as in Python), and begins in the same way: at first, the code block after the `try:` (the *body*) is executed. At the start of every time step during its execution, the condition from each `interrupt` clause is checked; if any are true, execution of the body is suspended and we instead begin to execute the corresponding *interrupt handler*. In the example above, there is only one interrupt, which fires when we come within 5 meters of any object. When that happens, `FollowLaneBehavior` is paused and we instead apply full braking for one time step. In the next step, we will resume `FollowLaneBehavior` wherever it left off, unless we are still within 5 meters of an object, in which case the interrupt will fire again.

If there are multiple `interrupt` clauses, successive clauses take precedence over those which precede them. Furthermore, such higher-priority interrupts can fire even during the execution of an earlier interrupt handler. This makes it easy to model a hierarchy of behaviors with different priorities; for example, we could implement a car which drives along a lane, passing slow cars and avoiding collisions, along the following lines:

```
behavior Drive():
    try:
        do FollowLaneBehavior()
    interrupt when self.distanceToNextObstacle() < 20:
        do PassingBehavior()
    interrupt when self.timeToCollision() < 5:
        do CollisionAvoidance()
```

Here, the car begins by lane following, switching to passing if there is a car or other obstacle too close ahead. During *either* of those two sub-behaviors, if the time to collision gets too low, we switch to collision avoidance. Once the `CollisionAvoidance` behavior completes, we will resume whichever behavior was interrupted earlier. If we were in the middle of `PassingBehavior`, it will run to completion (possibly being interrupted again) before we finally resume `FollowLaneBehavior`.

As this example illustrates, when an interrupt handler completes, by default we resume execution of the interrupted code. If this is undesired, the *abort* statement can be used to cause the entire try-interrupt statement to exit. For example, to run a behavior until a condition is met without resuming it afterward, we can write:

```
behavior ApproachAndTurnLeft():
    try:
        do FollowLaneBehavior()
    interrupt when (distance from self to intersection) < 10:
        abort    # cancel lane following
    do WaitForTrafficLightBehavior()
    do TurnLeftBehavior()
```

This is a common enough use case of interrupts that Scenic provides a shorthand notation:

```
behavior ApproachAndTurnLeft():
    do FollowLaneBehavior() until (distance from self to intersection) < 10
    do WaitForTrafficLightBehavior()
    do TurnLeftBehavior()
```

Scenic also provides a shorthand for interrupting a behavior after a certain period of time:

```
behavior DriveForAWhile():
    do FollowLaneBehavior() for 30 seconds
```

The alternative form `do behavior for n steps` uses time steps instead of real simulation time.

Finally, note that when try-interrupt statements are nested, interrupts of the outer statement take precedence. This makes it easy to build up complex behaviors in a modular way. For example, the behavior `Drive` we wrote above is relatively complicated, using interrupts to switch between several different sub-behaviors. We would like to be able to put it in a library and reuse it in many different scenarios without modification. Interrupts make this straightforward; for example, if for a particular scenario we want a car that drives normally but suddenly brakes for 5 seconds when it reaches a certain area, we can write:

```
behavior DriveWithSuddenBrake():
    haveBraked = False
    try:
        do Drive()
    interrupt when self in targetRegion and not haveBraked:
        do StopBehavior() for 5 seconds
        haveBraked = True
```

With this behavior, `Drive` operates as it did before, interrupts firing as appropriate to switch between lane following, passing, and collision avoidance. But during any of these sub-behaviors, if the car enters the `targetRegion` it will immediately brake for 5 seconds, then pick up where it left off.

1.5.3 Stateful Behaviors

As the last example shows, behaviors can use local variables to maintain state, which is useful when implementing behaviors which depend on actions taken in the past. To elaborate on that example, suppose we want a car which usually follows the `Drive` behavior, but every 15-30 seconds stops for 5 seconds. We can implement this behavior as follows:

```
behavior DriveWithRandomStops():
    delay = Range(15, 30) seconds
    last_stop = 0
    try:
        do Drive()
    interrupt when simulation().currentTime - last_stop > delay:
        do StopBehavior() for 5 seconds
        delay = Range(15, 30) seconds
        last_stop = simulation().currentTime
```

Here `delay` is the randomly-chosen amount of time to run `Drive` for, and `last_stop` keeps track of the time when we last started to run it. When the time elapsed since `last_stop` exceeds `delay`, we interrupt `Drive` and stop for 5 seconds. Afterwards, we pick a new `delay` before the next stop, and save the current time in `last_stop`, effectively resetting our timer to zero.

Note: It is possible to change global state from within a behavior by using the Python global statement, for instance to communicate between behaviors. If using this ability, keep in mind that the order in which behaviors of different agents is executed within a single time step could affect your results. The default order is the order in which the agents were defined, but it can be adjusted by overriding the *Simulation.scheduleForAgents* method.

1.5.4 Requirements and Monitors

Just as you can declare spatial constraints on scenes using the *require* statement, you can also impose constraints on dynamic scenarios. For example, if we don't want to generate any simulations where `car1` and `car2` are simultaneously visible from the ego car, we could write:

```
require always not ((ego can see car1) and (ego can see car2))
```

Here, *always condition* is a *temporal operator* which can only be used inside a requirement, and which evaluates to true if and only if the condition is true at *every* time step of the scenario. So if the condition above is ever false during a simulation, the requirement will be violated, causing Scenic to reject that simulation and sample a new one. Similarly, we can require that a condition hold at *some* time during the scenario using the *eventually* operator:

```
require eventually ego in intersection
```

It is also possible to relate conditions at different time steps. For example, to require that `car1` enters the intersection no later than when `car2` does, we can use the *until* operator:

```
require car2 not in intersection until car1 in intersection
require eventually car2 in intersection
```

Temporal operators can be combined with Boolean operators to build up more complex requirements[1], e.g.:

```
require (always car.speed < 30) implies (always distance to car > 10)
```

See *Temporal Operators* for a complete list of the available operators and their semantics.

You can also use the ordinary *require* statement inside a behavior to require that a given condition hold at a certain point during the execution of the behavior. For example, here is a simple elaboration of the `WaitUntilClose` behavior we saw above which requires that no pedestrian comes close to *self* until the ego does (after which we place no further restrictions):

```
behavior WaitUntilClose(threshold=15):
    while distance from self to ego > threshold:
        require self.distanceToClosest(Pedestrian) > threshold
        wait
    do FollowLaneBehavior()
```

If you want to enforce a complex requirement that isn't conveniently expressible either using the temporal operators built into Scenic or by modifying a behavior, you can define a monitor. Like behaviors, monitors are functions which run in parallel with the scenario, but they are not associated with any agent and any actions they take are ignored (so you might as well only use the *wait* statement). Here is a monitor for requiring that a given car spends at most a certain amount of time in the intersection:

[1] For those familiar with temporal logic, you can encode any formula of Linear Temporal Logic.

```
1 monitor LimitTimeInIntersection(car, limit=100):
2     stepsInIntersection = 0
3     while True:
4         require stepsInIntersection <= limit
5         if car in intersection:
6             stepsInIntersection += 1
7         wait
```

We use the variable `stepsInIntersection` to remember how many time steps `car` has spent in the intersection; if it ever exceeds the limit, the requirement on line 4 will fail and we will reject the simulation. Note the necessity of the *wait* statement on line 7: if we omitted it, the loop could run forever without any time actually passing in the simulation.

Like behaviors, monitors can take parameters, allowing a monitor defined in a library to be reused in various situations. To instantiate a monitor in a scenario, use the *require monitor* statement:

```
require monitor LimitTimeInIntersection(ego)
require monitor LimitTimeInIntersection(taxi, limit=200)
```

1.5.5 Preconditions and Invariants

Even general behaviors designed to be used in multiple scenarios may not operate correctly from all possible starting states: for example, `FollowLaneBehavior` assumes that the agent is actually in a lane rather than, say, on a sidewalk. To model such assumptions, Scenic provides a notion of *guards* for behaviors. Most simply, we can specify one or more *preconditions*:

```
behavior MergeInto(newLane):
    precondition: self.lane is not newLane and self.road is newLane.road
    ...
```

Here, the precondition requires that whenever the `MergeInto` behavior is executed by an agent, the agent must not already be in the destination lane but should be on the same road. We can add any number of such preconditions; like ordinary requirements, violating any precondition causes the simulation to be rejected.

Since behaviors can be interrupted, it is possible for a behavior to resume execution in a state it doesn't expect: imagine a car which is lane following, but then swerves onto the shoulder to avoid an accident; naïvely resuming lane following, we find we are no longer in a lane. To catch such situations, Scenic allows us to define *invariants* which are checked at every time step during the execution of a behavior, not just when it begins running. These are written similarly to preconditions:

```
behavior FollowLaneBehavior():
    invariant: self in road
    ...
```

While the default behavior for guard violations is to reject the simulation, in some cases it may be possible to recover from a violation by taking some additional actions. To enable this kind of design, Scenic signals guard violations by raising a *GuardViolation* exception which can be caught like any other exception; the simulation is only rejected if the exception propagates out to the top level. So to model the lane-following-with-collision-avoidance behavior suggested above, we could write code like this:

```
behavior Drive():
    while True:
        try:
```

(continues on next page)

(continued from previous page)

```
            do FollowLaneBehavior()
        interrupt when self.distanceToClosest(Object) < 5:
            do CollisionAvoidance()
        except InvariantViolation:   # FollowLaneBehavior has failed
            do GetBackOntoRoad()
```

When any object comes within 5 meters, we suspend lane following and switch to collision avoidance. When the `CollisionAvoidance` behavior completes, `FollowLaneBehavior` will be resumed; if its invariant fails because we are no longer on the road, we catch the resulting *InvariantViolation* exception and run a `GetBackOntoRoad` behavior to restore the invariant. The whole `try` statement then completes, so the outermost loop iterates and we begin lane following once again.

1.5.6 Terminating the Scenario

By default, scenarios run forever, unless the *--time* option is used to impose a time limit. However, scenarios can also define termination criteria using the *terminate when* statement; for example, we could decide to end a scenario as soon as the ego car travels at least a certain distance:

```
start = new Point on road
ego = new Car at start
terminate when (distance to start) >= 50
```

Additionally, the *terminate* statement can be used inside behaviors and monitors: if it is ever executed, the scenario ends. For example, we can use a monitor to terminate the scenario once the ego spends 30 time steps in an intersection:

```
monitor StopAfterTimeInIntersection:
    totalTime = 0
    while totalTime < 30:
        if ego in intersection:
            totalTime += 1
        wait
    terminate
```

Note: In order to make sure that requirements are not violated, termination criteria are only checked *after* all requirements. So if in the same time step a monitor uses the *terminate* statement but another behavior uses *require* with a false condition, the simulation will be rejected rather than terminated.

1.5.7 Trying Some Examples

You can see all of the above syntax in action by running some of our examples of dynamic scenarios. We have examples written for the CARLA and LGSVL driving simulators, and those in `examples/driving` in particular are designed to use Scenic's abstract *driving domain* and so work in either of these simulators, as well as Scenic's built-in Newtonian physics simulator. The Newtonian simulator is convenient for testing and simple experiments; you can find details on how to install the more realistic simulators in our *Supported Simulators* page (they should work on both Linux and Windows, but not macOS, at the moment).

Let's try running `examples/driving/badlyParkedCarPullingIn.scenic`, which implements the "a badly-parked car, which pulls into the road as the ego car approaches" scenario we mentioned above. To start out, you can run it like any other Scenic scenario to get the usual schematic diagram of the generated scenes:

```
$ scenic examples/driving/badlyParkedCarPullingIn.scenic --2d
```

To run dynamic simulations, add the `--simulate` option (`-S` for short). Since this scenario is not written for a particular simulator, you'll need to specify which one you want by using the `--model` option (`-m` for short) to select the corresponding Scenic world model: for example, to use the Newtonian simulator we could add `--model scenic.simulators.newtonian.driving_model`. It's also a good idea to put a time bound on the simulations, which we can do using the `--time` option.

```
$ scenic examples/driving/badlyParkedCarPullingIn.scenic \
    --2d       \
    --simulate \
    --model scenic.simulators.newtonian.driving_model \
    --time 200
```

Running the scenario in CARLA is exactly the same, except we use the `--model scenic.simulators.carla.model` option instead (make sure to start CARLA running first). For LGSVL, the one difference is that this scenario specifies a map which LGSVL doesn't have built in; fortunately, it's easy to switch to a different map. For scenarios using the *driving domain*, the map file is specified by defining a global parameter `map`, and for the LGSVL interface we use another parameter `lgsvl_map` to specify the name of the map in LGSVL (the CARLA interface likewise uses a parameter `carla_map`). These parameters can be set at the command line using the `--param` option (`-p` for short); for example, let's pick the "BorregasAve" LGSVL map, an OpenDRIVE file for which is included in the Scenic repository. We can then run a simulation by starting LGSVL in "API Only" mode and invoking Scenic as follows:

```
$ scenic examples/driving/badlyParkedCarPullingIn.scenic \
    --2d       \
    --simulate \
    --model scenic.simulators.lgsvl.model \
    --time 200 \
    --param map assets/maps/LGSVL/borregasave.xodr \
    --param lgsvl_map BorregasAve
```

Try playing around with different example scenarios and different choices of maps (making sure that you keep the `map` and `lgsvl_map`/`carla_map` parameters consistent). For both CARLA and LGSVL, you don't have to restart the simulator between scenarios: just kill Scenic[2] and restart it with different arguments.

1.5.8 Further Reading

This tutorial illustrated most of Scenic's core syntax for dynamic scenarios. As with the rest of Scenic's syntax, these constructs are summarized in our *Syntax Guide*, with links to detailed documentation in the *Language Reference*. You may also be interested in some other sections of the documentation:

Composing Scenarios
: Building more complex scenarios out of simpler ones in a modular way.

Supported Simulators
: Details on which simulator interfaces support dynamic scenarios.

Execution of Dynamic Scenarios
: The gory details of exactly how behaviors run, monitors are checked, etc. (probably not worth reading unless you're having a subtle timing issue).

[2] Or use the `--count` option to have Scenic automatically terminate after a desired number of simulations.

1.6 Composing Scenarios

Scenic provides facilities for defining multiple scenarios in a single program and *composing* them in various ways. This enables writing a library of scenarios which can be repeatedly used as building blocks to construct more complex scenarios.

1.6.1 Modular Scenarios

The *scenario* statement defines a named, reusable scenario, optionally with tunable parameters: what we call a modular scenario. For example, here is a scenario which creates a parked car on the shoulder of the *ego*'s current lane (assuming there is one), using some APIs from the *Driving Domain*:

```
scenario ParkedCar(gap=0.25):
    precondition: ego.laneGroup._shoulder != None
    setup:
        spot = new OrientedPoint on visible ego.laneGroup.curb
        parkedCar = new Car left of spot by gap
```

The `setup` block contains Scenic code which executes when the scenario is instantiated, and which can define classes, create objects, declare requirements, etc. as in any ordinary Scenic scenario. Additionally, we can define preconditions and invariants, which operate in the same way as for *dynamic behaviors*.

Having now defined the `ParkedCar` scenario, we can use it in a more complex scenario, potentially multiple times:

```
scenario Main():
    setup:
        ego = new Car
    compose:
        do ParkedCar(), ParkedCar(0.5)
```

Here our `Main` scenario itself only creates the ego car; then its `compose` block orchestrates how to run other modular scenarios. In this case, we invoke two copies of the `ParkedCar` scenario in parallel, specifying in one case that the gap between the parked car and the curb should be 0.5 m instead of the default 0.25. So the scenario will involve three cars in total, and as usual Scenic will automatically ensure that they are all on the road and do not intersect.

1.6.2 Parallel and Sequential Composition

The scenario above is an example of *parallel* composition, where we use the *do* statement to run two scenarios at the same time. We can also use *sequential* composition, where one scenario begins after another ends. This is done the same way as in behaviors: in fact, the `compose` block of a scenario is executed in the same way as a monitor, and allows all the same control-flow constructs. For example, we could write a `compose` block as follows:

```
1 while True:
2     do ParkedCar(gap=0.25) for 30 seconds
3     do ParkedCar(gap=0.5) for 30 seconds
```

Here, a new parked car is created every 30 seconds,[1] with the distance to the curb alternating between 0.25 and 0.5 m. Note that without the `for 30 seconds` qualifier, we would never get past line 2, since the `ParkedCar` scenario does not define any termination conditions using *terminate when* (or *terminate* in a `compose` block) and so runs forever by default. If instead we want to create a new car only when the *ego* has passed the current one, we can use a *do-until* statement:

[1] In a real implementation, we would probably want to require that the parked car is not initially visible from the *ego*, to avoid the sudden appearance of cars out of nowhere.

```
while True:
    subScenario = ParkedCar(gap=0.25)
    do subScenario until (distance past subScenario.parkedCar) > 10
```

Note how we can refer to the `parkedCar` variable created in the `ParkedCar` scenario as a property of the scenario. Combined with the ability to pass objects as parameters of scenarios, this is convenient for reusing objects across scenarios.

1.6.3 Interrupts, Overriding, and Initial Scenarios

The *try-interrupt* statement used in behaviors can also be used in `compose` blocks to switch between scenarios. For example, suppose we already have a scenario where the *ego* is following a `leadCar`, and want to elaborate it by adding a parked car which suddenly pulls in front of the lead car. We could write a `compose` block as follows:

```
1 following = FollowingScenario()
2 try:
3     do following
4 interrupt when (distance to following.leadCar) < 10:
5     do ParkedCarPullingAheadOf(following.leadCar)
```

If the `ParkedCarPullingAheadOf` scenario is defined to end shortly after the parked car finishes entering the lane, the interrupt handler will complete and Scenic will resume executing `FollowingScenario` on line 3 (unless the *ego* is still within 10 m of the lead car).

Suppose that we want the lead car to behave differently while the parked car scenario is running; for example, perhaps the behavior for the lead car defined in `FollowingScenario` does not handle a parked car suddenly pulling in. To enable changing the `behavior` or other properties of an object in a sub-scenario, Scenic provides the *override* statement, which we can use as follows:

```
scenario ParkedCarPullingAheadOf(target):
    setup:
        override target with behavior FollowLaneAvoidingCollisions
        parkedCar = new Car left of ...
```

Here we override the `behavior` property of `target` for the duration of the scenario, reverting it back to its original value (and thereby continuing to execute the old behavior) when the scenario terminates. The *override* `object specifier, ...` statement takes a comma-separated list of specifiers like an *instance creation*, and can specify any properties of the object except for dynamic properties like `position` or `speed` which can only be indirectly controlled by taking actions.

In order to allow writing scenarios which can both stand on their own and be invoked during another scenario, Scenic provides a special conditional statement testing whether we are inside the *initial scenario*, i.e., the very first scenario to run. For instance:

```
scenario TwoLanePedestrianScenario():
    setup:
        if initial scenario:  # create ego on random 2-lane road
            roads = filter(lambda r: len(r.lanes) == 2, network.roads)
            road = Uniform(*roads)  # pick uniformly from list
            ego = new Car on road
        else:  # use existing ego car; require it is on a 2-lane road
            require len(ego.road.lanes) == 2
            road = ego.road
        new Pedestrian on visible road.sidewalkRegion, with behavior ...
```

1.6.4 Random Selection of Scenarios

For very general scenarios, like "driving through a city, encountering typical human traffic", we may want a variety of different events and interactions to be possible. We saw in the *Dynamic Scenarios* tutorial how we can write behaviors for individual agents which choose randomly between possible actions; Scenic allows us to do the same with entire scenarios. Most simply, since scenarios are first-class objects, we can write functions which operate on them, perhaps choosing a scenario from a list of options based on some complex criterion:

```
chosenScenario = pickNextScenario(ego.position, ...)
do chosenScenario
```

However, some scenarios may only make sense in certain contexts; for example, a red light runner scenario can take place only at an intersection. To facilitate modeling such situations, Scenic provides variants of the *do* statement which randomly choose scenarios to run amongst only those whose preconditions are satisfied:

```
1 do choose RedLightRunner, Jaywalker, ParkedCar(gap=0.5)
2 do choose {RedLightRunner: 2, Jaywalker: 1, ParkedCar(gap=0.5): 1}
3 do shuffle RedLightRunner, Jaywalker, ParkedCar
```

Here, line 1 checks the preconditions of the three given scenarios, then executes one (and only one) of the enabled scenarios. If for example the current road has no shoulder, then `ParkedCar` will be disabled and we will have a 50/50 chance of executing either `RedLightRunner` or `Jaywalker` (assuming their preconditions are satisfied). If *none* of the three scenarios are enabled, Scenic will reject the simulation. Line 2 shows a non-uniform variant, where `RedLightRunner` is twice as likely to be chosen as each of the other scenarios (so if only `ParkedCar` is disabled, we will pick `RedLightRunner` with probability 2/3; if none are disabled, 2/4). Finally, line 3 is a shuffled variant, where *all three* scenarios will be executed, but in random order.[2]

1.7 Syntax Guide

This page summarizes the syntax of Scenic, excluding the basic syntax of variable assignments, functions, loops, etc., which is identical to Python (see the Python Tutorial for an introduction). For more details, click the links for individual language constructs to go to the corresponding section of the *Language Reference*.

1.7.1 Primitive Data Types

Booleans	expressing truth values
Scalars	representing distances, angles, etc. as floating-point numbers
Vectors	representing positions and offsets in space
Headings	representing 2D orientations in the XY plane
Orientations	representing 3D orientations in space
Vector Fields	associating an orientation to each point in space
Regions	representing sets of points in space
Shapes	representing shapes (regions modulo similarity)

[2] Respecting preconditions, so in particular the simulation will be rejected if at some point none of the remaining scenarios to execute are enabled.

1.7.2 Distributions

Range(low, high)	uniformly-distributed real number in the interval
DiscreteRange(low, high)	uniformly-distributed integer in the (fixed) interval
Normal(mean, stdDev)	normal distribution with the given mean and standard deviation
TruncatedNormal(mean, stdDev, low, high)	normal distribution truncated to the given window
Uniform(value, ...)	uniform over a finite set of values
Discrete({value: weight, ...})	discrete with given values and weights
new Point in region	uniformly-distributed *Point* in a region

1.7.3 Statements

Compound Statements

Syntax	Meaning
class name*[*(superclass)*]:*	Defines a Scenic class.
behavior name(arguments)*:*	Defines a dynamic behavior.
monitor name(arguments)*:*	Defines a monitor.
scenario name(arguments)*:*	Defines a modular scenario.
try: ... *interrupt when* boolean*:*	Run code with interrupts inside a dynamic behavior or modular scenario.

Simple Statements

Syntax	Meaning
model name	Select the world model.
import module	Import a Scenic or Python module.
param name = value, ...	Define global parameters of the scenario.
require boolean	Define a hard requirement.
require[number] boolean	Define a soft requirement.
require LTL formula	Define a dynamic hard requirement.
require monitor monitor	Define a dynamic requirement using a monitor.
terminate when boolean	Define a termination condition.
terminate after scalar *(seconds | steps)*	Set the scenario to terminate after a given amount of time.
mutate identifier, ... *[by* number*]*	Enable mutation of the given list of objects.
record [initial | final] value *as* name	Save a value at every time step or only at the start/end of the simulation.

Dynamic Statements

These statements can only be used inside a dynamic behavior, monitor, or `compose` block of a modular scenario.

Syntax	Meaning
`take action, ...`	Take the action(s) specified.
`wait`	Take no actions this time step.
`terminate`	Immediately end the scenario.
`terminate simulation`	Immediately end the entire simulation.
`do behavior/scenario, ...`	Run one or more sub-behaviors/sub-scenarios until they complete.
`do behavior/scenario, ... until boolean`	Run sub-behaviors/scenarios until they complete or a condition is met.
`do behavior/scenario, ... for scalar (seconds | steps)`	Run sub-behaviors/scenarios for (at most) a specified period of time.
`do choose behavior/scenario, ...`	Run *one* choice of sub-behavior/scenario whose preconditions are satisfied.
`do shuffle behavior/scenario, ...`	Run several sub-behaviors/scenarios in a random order, satisfying preconditions.
`abort`	Break out of the current `try-interrupt` statement.
`override object specifier, ...`	Override properties of an object for the duration of the current scenario.

1.7.4 Objects

The syntax `new class specifier, ...` creates an instance of a Scenic class.

The Scenic class *Point* provides the basic position properties in the first table below; its subclass *OrientedPoint* adds the orientation properties in the second table. Finally, the class *Object*, which represents physical objects and is the default superclass of user-defined Scenic classes, adds the properties in the third table. See the *Objects and Classes Reference* for details.

Property	**Default**	**Meaning**
position[1]	(0, 0, 0)	position in global coordinates
visibleDistance	50	distance for the 'can see' operator
viewRayDensity	5	determines ray count (if ray count is not provided)
viewRayDistanceScaling	False	whether to scale number of rays with distance (if ray count is not provided)
viewRayCount	None	tuple of number of rays to send in each dimension.
mutationScale	0	overall scale of *mutations*
positionStdDev	(1,1,0)	mutation standard deviation for `position`

Properties added by *OrientedPoint*:

Property	**Default**	**Meaning**
yaw[1]	0	yaw in local coordinates
pitch[1]	0	pitch in local coordinates
roll[1]	0	roll in local coordinates
parentOrientation	global	basis for local coordinate system
viewAngles	(2,)	angles for visibility calculations
orientationStdDev	(5°, 0, 0)	mutation standard deviation for `orientation`

[1] These are dynamic properties, updated automatically every time step during dynamic simulations.

Properties added by *Object*:

Property	Default	Meaning
width	1	width of bounding box (X axis)
length	1	length of bounding box (Y axis)
height	1	height of bounding box (Z axis)
shape	*BoxShape*	shape of the object
allowCollisions	False	whether collisions are allowed
regionContainedIn	workspace	*Region* the object must lie within
baseOffset	(0, 0, -self.height/2)	offset determining the base of the object
contactTolerance	1e-4	max distance to be considered on a surface
sideComponentThresholds	(-0.5, 0.5) per side	thresholds to determine side surfaces
cameraOffset	(0, 0, 0)	position of camera for *can see*
requireVisible	False	whether object must be visible from ego
occluding	True	whether object occludes visibility
showVisibleRegion	False	whether to display the visible region
color	None	color of object
velocity[Page 45, 1]	from speed	initial (instantaneous) velocity
speed[Page 45, 1]	0	initial (later, instantaneous) speed
angularVelocity[Page 45, 1]	(0, 0, 0)	initial (instantaneous) angular velocity
angularSpeed[Page 45, 1]	0	angular speed (change in heading/time)
behavior	None	dynamic behavior, if any
lastActions	None	tuple of actions taken in last timestamp

1.7.5 Specifiers

The *with property value* specifier can specify any property, including new properties not built into Scenic. Additional specifiers for the position and orientation properties are listed below.

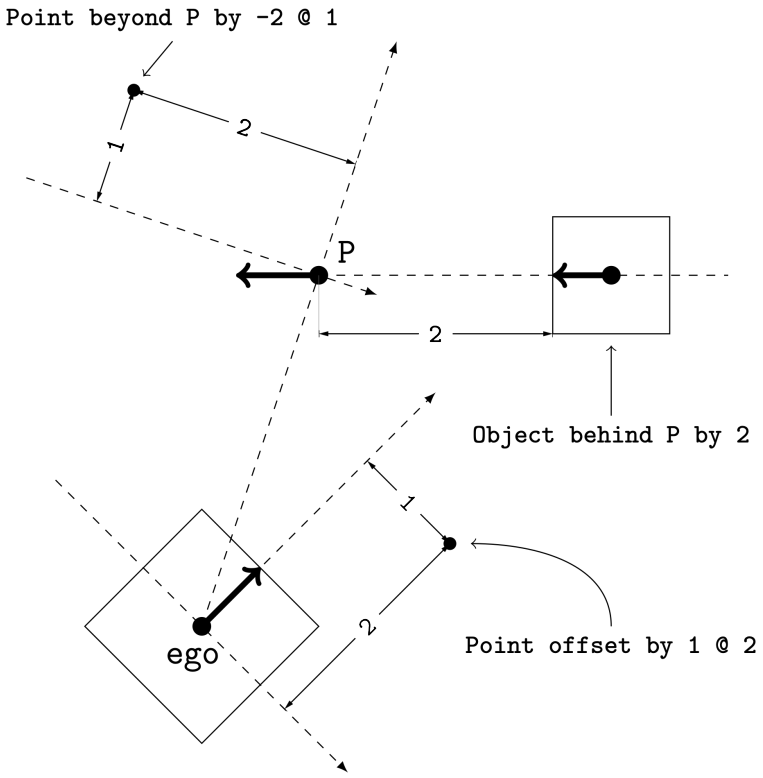


Fig. 4: Illustration of the `beyond`, `behind`, and `offset by` specifiers. Each *`OrientedPoint`* (e.g. `P`) is shown as a bold arrow.

Specifier for `position`	Meaning
`at` `vector`	Positions the object at the given global coordinates
`in` `region`	Positions the object uniformly at random in the given Region
`contained in` `region`	Positions the object uniformly at random entirely contained in the given Region
`on` `region`	Positions the base of the object uniformly at random in the given Region, or modifies the position so that the base is in the Region.
`offset by` `vector`	Positions the object at the given coordinates in the local coordinate system of ego (which must already be defined)
`offset along` `direction` *`by`* `vector`	Positions the object at the given coordinates, in a local coordinate system centered at ego and oriented along the given direction
`beyond` `vector` *`by`* `(vector | scalar)` *`[from`* `(vector | OrientedPoint)]`	Positions the object with respect to the line of sight from a point or the ego
`visible [from` `(Point | OrientedPoint)]`	Ensures the object is visible from the ego, or from the given Point/OrientedPoint if given, while optionally specifying position to be in the appropriate visible region.
`not visible [from` `(Point | OrientedPoint)]`	Ensures the object is not visible from the ego, or from the given Point/OrientedPoint if given, while optionally specifying position to be outside the appropriate visible region.
`(left | right) of` `(vector | OrientedPoint | Object)` *`[by`* `scalar]`	Positions the object to the left/right by the given scalar distance.
`(ahead of | behind)` `(vector | OrientedPoint | Object)` *`[by`* `scalar]`	Positions the object to the front/back by the given scalar distance
`(above | below)` `(vector | OrientedPoint | Object)` *`[by`* `scalar]`	Positions the object above/below by the given scalar distance
`following` `vectorField` *`[from`* `vector]` *`for`* `scalar`	Position by following the given vector field for the given distance starting from ego or the given vector

Specifier for `orientation`	Meaning
facing `orientation`	Orients the object along the given orientation in global coordinates
facing `vectorField`	Orients the object along the given vector field at the object's position
facing (toward | away from) `vector`	Orients the object toward/away from the given position (thereby depending on the object's position)
facing directly (toward | away from) `vector`	Orients the object *directly* toward/away from the given position (thereby depending on the object's position)
apparently facing `heading` *[from* `vector]`	Orients the object so that it has the given heading with respect to the line of sight from ego (or the given vector)

1.7.6 Operators

In the following tables, operators are grouped by the type of value they return.

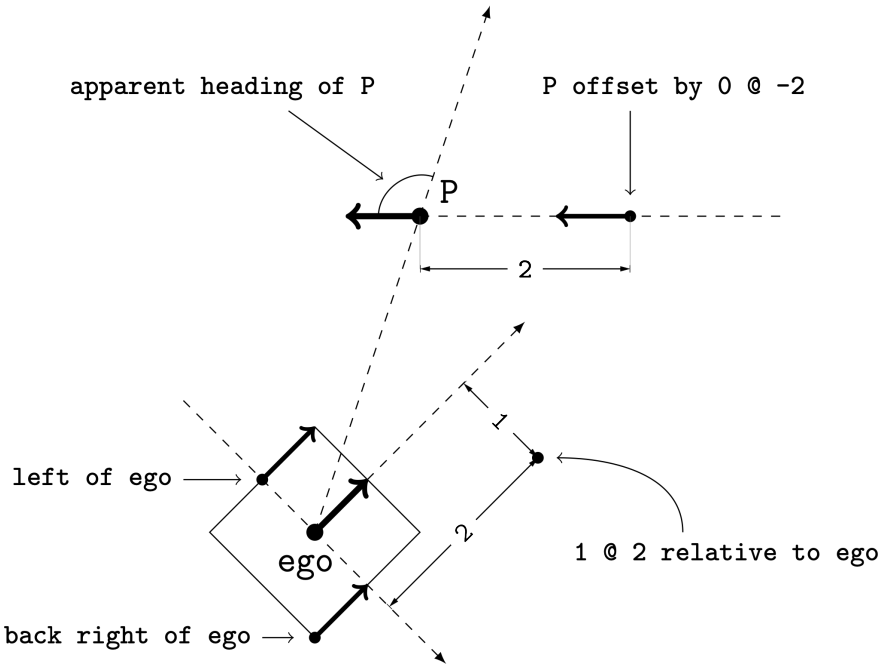


Fig. 5: Illustration of several operators. Each *OrientedPoint* (e.g. P) is shown as a bold arrow.

Scalar Operators	Meaning
relative heading of `heading` *[from* `heading]`	The relative heading of the given heading with respect to ego (or the `from` heading)
apparent heading of `OrientedPoint` *[from* `vector]`	The apparent heading of the *OrientedPoint*, with respect to the line of sight from ego (or the given vector)
distance [from `vector]` *to* `vector`	The distance to the given position from ego (or the `from` vector)
angle [from `vector]` *to* `vector`	The heading (azimuth) to the given position from ego (or the `from` vector)
altitude [from `vector]` *to* `vector`	The altitude to the given position from ego (or the `from` vector)

Boolean Operators	Meaning
`(Point | OrientedPoint) can see (vector | Object)`	Whether or not a position or *Object* is visible from a *Point* or *OrientedPoint*.
`(vector | Object) in region`	Whether a position or *Object* lies in the region

Orientation Operators	Meaning
`scalar deg`	The given angle, interpreted as being in degrees
`vectorField at vector`	The orientation specified by the vector field at the given position
`direction relative to direction`	The first direction (a heading, orientation, or vector field), interpreted as an offset relative to the second direction

Vector Operators	Meaning
`vector (relative to | offset by) vector`	The first vector, interpreted as an offset relative to the second vector (or vice versa)
`vector offset along direction by vector`	The second vector, interpreted in a local coordinate system centered at the first vector and oriented along the given direction

Region Operators	Meaning
`visible region`	The part of the given region visible from ego
`not visible region`	The part of the given region not visible from ego
`region visible from (Point | OrientedPoint)`	The part of the given region visible from the given *Point* or *OrientedPoint*.
`region not visible from (Point | OrientedPoint)`	The part of the given region not visible from the given *Point* or *OrientedPoint*.

OrientedPoint Operators	Meaning
`vector relative to OrientedPoint`	The given vector, interpreted in the local coordinate system of the OrientedPoint
`OrientedPoint offset by vector`	Equivalent to `vector relative to OrientedPoint` above
`(front | back | left | right) of Object`	The midpoint of the corresponding side of the bounding box of the Object, inheriting the Object's orientation.
`(front | back) (left | right) of Object`	The midpoint of the corresponding edge of the bounding box of the Object, inheriting the Object's orientation.
`(front | back) (left | right) of Object`	The midpoint of the corresponding edge of the bounding box of the Object, inheriting the Object's orientation.
`(top | bottom) (front | back) (left | right) of Object`	The corresponding corner of the bounding box of the Object, inheriting the Object's orientation.

Temporal Operators	Meaning
`always condition`	Require the condition to hold at every time step.
`eventually condition`	Require the condition to hold at some time step.
`next condition`	Require the condition to hold in the next time step.
`condition until condition`	Require the first condition to hold until the second becomes true.
`condition implies condition`	Require the second condition to hold if the first condition holds.

1.7.7 Built-in Functions

Function	Description
Misc Python functions	Various Python functions including `min`, `max`, `open`, etc.
filter	Filter a possibly-random list (allowing limited randomized control flow).
resample	Sample a new value from a distribution.
localPath	Convert a relative path to an absolute path, based on the current directory.
verbosePrint	Like `print`, but silent at low-enough verbosity levels.
simulation	Get the the current simulation object.

1.8 Language Reference

Language Constructs

These pages describe the syntax of Scenic in detail. For a one-page summary of Scenic's syntax, see the *Syntax Guide*. For details on the syntax for functions, loops, etc. inherited from Python, see the Python Language Reference.

1.8.1 General Notes on Syntax

Keywords

Keywords

The following words are reserved by Scenic and cannot be used as identifiers (i.e. as names of variables, functions, classes, properties, etc.).

```
False       break       except      lambda      require
None        by          finally     new         return
True        class       for         nonlocal    to
and         continue    from        not         try
as          def         global      of          until
assert      del         if          on          while
async       do          import      or          with
at          elif        in          pass        yield
await       else        is          raise
```

Soft Keywords

The following words have special meanings in Scenic in certain contexts, but are still available for use as identifiers. Users should take care not to use these names when doing so would introduce ambiguity. For example, consider the following code:

```
distance = 5  # not a good variable name to use here
new Object beyond A by distance from B
```

This might appear to use the three-argument form of the *beyond* specifier, creating the new object at distance 5 beyond `A` from the point of view of `B`. But in fact Scenic parses the code as `beyond A by (distance from B)`, because the interpretation of `distance` as being part of the *distance from* operator takes precedence.

To avoid confusion, we recommend not using `distance`, `angle`, `offset`, `altitude`, or `visible` as identifiers in code that uses Scenic operators or specifiers (inside pure-Python helper functions is fine).

_	behind	facing	mutate	see
abort	below	final	next	setup
above	beyond	follow	not	shuffle
additive	bottom	following	of	simulation
after	can	from	offset	simulator
ahead	case	front	override	steps
along	choose	heading	param	take
altitude	compose	implies	past	terminate
always	contained	initial	position	top
angle	deg	interrupt	precondition	toward
apparent	directly	invariant	record	visible
apparently	distance	left	relative	wait
away	dynamic	match	right	when
back	ego	model	scenario	workspace
behavior	eventually	monitor	seconds	

1.8.2 Data Types Reference

This page describes the primitive data types built into Scenic. In addition to these types, Scenic provides a class hierarchy for *points*, *oriented points*, and *objects*: see the *Objects and Classes Reference*.

Boolean

Booleans represent truth values, and can be `True` or `False`.

Note: These are equivalent to the Python `bool` type.

Scalar

Scalars represent distances, angles, etc. as floating-point numbers, which can be sampled from various distributions.

Note: These are equivalent to the Python `float` type; however, any context which accepts a scalar will also allow an `int` or a NumPy numeric type such as `numpy.single` (to be precise, any instance of `numbers.Real` is legal).

Vector

Vectors represent positions and offsets in space. They are constructed from coordinates using a length-3 list or tuple (`[X, Y, Z]` or `(X, Y, Z)`. Alternatively, they can be constructed with the syntax `X @ Y` (inspired by Smalltalk) or a length-2 list or tuple, with an implied z value of 0. By convention, coordinates are in meters, although the semantics of Scenic does not depend on this.

For convenience, instances of *Point* can be used in any context where a vector is expected: so for example if `P` is a *Point*, then `P offset by (1, 2)` is equivalent to `P.position offset by (1, 2)`.

Changed in version 3.0: Vectors are now 3 dimensional.

Heading

Headings represent yaw in the global XY plane. Scenic represents headings in radians, measured anticlockwise from North, so that a heading of 0 is due North and a heading of /2 is due West. We use the convention that the heading of a local coordinate system is the heading of its Y-axis, so that, for example, the vector `-2 @ 3` means 2 meters left and 3 ahead.

For convenience, instances of *OrientedPoint* can be used in any context where a heading is expected: so for example if `OP` is an *OrientedPoint*, then `relative heading of OP` is equivalent to `relative heading of OP.heading`. Since *OrientedPoint* is a subclass of *Point*, expressions involving two oriented points like `OP1 relative to OP2` can be ambiguous: the polymorphic operator `relative to` accepts both vectors and headings, and either version could be meant here. Scenic rejects such expressions as being ambiguous: more explicit syntax like `OP1.position relative to OP2` must be used instead.

Orientation

Orientations represent orientations in 3D space. Scenic represents orientations internally using quaternions, though for convenience they can be created using Euler angles. Scenic follows the right hand rule with the Z,X,Y order of rotations. In other words, Euler angles are given as (Yaw, Pitch, Roll), in radians, and applied in that order. To help visualize, one can consider their right hand with fingers extended orthogonally. The index finger points along positive X, the middle finger bends left along positive Y, and the thumb ends up pointing along positive Z. For rotations, align your right thumb with a positive axis and the way your fingers curl is a positive rotation.

New in version 3.0.

Vector Field

Vector fields associate an orientation to each point in space. For example, a vector field could represent the shortest paths to a destination, or the nominal traffic direction on a road (e.g. *`scenic.domains.driving.model.roadDirection`*).

Changed in version 3.0: Vector fields now return an *Orientation* instead of a scalar heading.

Region

Regions represent sets of points in space. Scenic provides a variety of ways to define regions in 2D and 3D space: meshes, rectangles, circular sectors, line segments, polygons, occupancy grids, and explicit lists of points, among others.

Regions can have an associated vector field giving points in the region preferred orientations. For example, a region representing a lane of traffic could have a preferred orientation aligned with the lane, so that we can easily talk about distances along the lane, even if it curves. Another possible use of preferred orientations is to give the surface of an object normal vectors, so that other objects placed on the surface face outward by default.

The main operations available for use with all regions are:

- the *`(vector | Object) in region`* operator to test containment within a region;
- the *`visible region`* operator to get the part of a region which is visible from the *`ego`*;
- the *`in region`* specifier to choose a position uniformly at random inside a region;
- the *`on region`* specifier to choose a position like *`in region`* or to project an existing position onto the region's surface.

If you need to perform more complex operations on regions, or are writing a world model and need to define your own regions, you will have to work with the *`Region`* class (which regions are instances of) and its subclasses for particular

types of regions. These are listed in the *Regions Types* reference. If you are working on Scenic's internals, see the *scenic.core.regions* module for full details.

Shape

Shapes represent the shape of an object, i.e., the volume it occupies modulo translation, rotation, and scaling. Shapes are represented by meshes, automatically converted to unit size and centered; Scenic considers the side of the shape facing the positive Y axis to be its front.

Shapes can be created from an arbitrary mesh or using one of the geometric primitives below. For convenience, a shape created with specified dimensions will set the default dimensions for any *Object* created with that shape. When creating a *MeshShape*, if no dimensions are provided then dimensions will be inferred from the mesh. *MeshShape* also takes an optional `initial_rotation` parameter, which allows directions other than the positive Y axis to be considered the front of the shape.

class MeshShape(*mesh*, *dimensions=None*, *scale=1*, *initial_rotation=None*)

A Shape subclass defined by a `trimesh.base.Trimesh` object.

The mesh passed must be a `trimesh.base.Trimesh` object that represents a well defined volume (i.e. the `is_volume` property must be true), meaning the mesh must be watertight, have consistent winding and have outward facing normals.

Parameters

- **mesh** – A mesh object.
- **dimensions** – The raw (before scaling) dimensions of the shape. If dimensions and scale are both specified the dimensions are first set by dimensions, and then scaled by scale.
- **scale** – Scales all the dimensions of the shape by a multiplicative factor. If dimensions and scale are both specified the dimensions are first set by dimensions, and then scaled by scale.
- **initial_rotation** – A 3-tuple containing the yaw, pitch, and roll respectively to apply when loading the mesh. Note the initial_rotation must be fixed.

classmethod fromFile(*path*, *filetype=None*, *compressed=None*, *binary=False*, ***kwargs*)

Load a mesh shape from a file, attempting to infer filetype and compression.

For example: "foo.obj.bz2" is assumed to be a compressed .obj file. "foo.obj" is assumed to be an uncompressed .obj file. "foo" is an unknown filetype, so unless a filetype is provided an exception will be raised.

Parameters

- **path** (*str*) – Path to the file to import.
- **filetype** (*str*) – Filetype of file to be imported. This will be inferred if not provided. The filetype must be one compatible with `trimesh.load`.
- **compressed** (*bool*) – Whether or not this file is compressed (with bz2). This will be inferred if not provided.
- **binary** (*bool*) – Whether or not to open the file as a binary file.
- **kwargs** – Additional arguments to the MeshShape initializer.

class BoxShape(*dimensions=(1, 1, 1)*, *scale=1*, *initial_rotation=None*)

A box shape with all dimensions 1 by default.

class CylinderShape(*dimensions=(1, 1, 1)*, *scale=1*, *initial_rotation=None*, *sections=24*)

A cylinder shape with all dimensions 1 by default.

class ConeShape(*dimensions=(1, 1, 1)*, *scale=1*, *initial_rotation=None*)

A cone shape with all dimensions 1 by default.

class SpheroidShape(*dimensions=(1, 1, 1)*, *scale=1*, *initial_rotation=None*)

A spheroid shape with all dimensions 1 by default.

1.8.3 Region Types Reference

This page covers the *scenic.core.regions.Region* class and its subclasses; for an introduction to the concept of regions in Scenic and the basic operations available for them, see *Region*.

Abstract Regions

class Region(*name*, **dependencies*, *orientation=None*)

An abstract base class for Scenic Regions

intersects(*other*)

Check if this *Region* intersects another.

Return type
bool

intersect(*other*, *triedReversed=False*)

Get a *Region* representing the intersection of this one with another.

If both regions have a preferred orientation, the one of `self` is inherited by the intersection.

Return type
Region

union(*other*, *triedReversed=False*)

Get a *Region* representing the union of this one with another.

Not supported by all region types.

Return type
Region

Point Sets and Lines

class PointSetRegion(*name*, *points*, *kdTree=None*, *orientation=None*, *tolerance=1e-06*)

Region consisting of a set of discrete points.

No *Object* can be contained in a `PointSetRegion`, since the latter is discrete. (This may not be true for subclasses, e.g. `GridRegion`.)

Parameters

- **name** (*str*) – name for debugging
- **points** (*arraylike*) – set of points comprising the region
- **kdTree** (scipy.spatial.KDTree, optional) – k-D tree for the points (one will be computed if none is provided)
- **orientation** (*Vector Field*; optional) – preferred orientation for the region
- **tolerance** (*float; optional*) – distance tolerance for checking whether a point lies in the region

class PolylineRegion(*points=None*, *polyline=None*, *orientation=True*, *name=None*)

Region given by one or more polylines (chain of line segments).

The region may be specified by giving either a sequence of points or `shapely` polylines (a `LineString` or `MultiLineString`).

Parameters

- **points** – sequence of points making up the polyline (or `None` if using the **polyline** argument instead).
- **polyline** – `shapely` polyline or collection of polylines (or `None` if using the **points** argument instead).
- **orientation** (*optional*) – preferred orientation to use, or `True` to use an orientation aligned with the direction of the polyline (the default).
- **name** (*str; optional*) – name for debugging.

property start

Get an *OrientedPoint* at the start of the polyline.

The OP's orientation will be aligned with the orientation of the region, if there is one (the default orientation pointing along the polyline).

property end

Get an *OrientedPoint* at the end of the polyline.

The OP's orientation will be aligned with the orientation of the region, if there is one (the default orientation pointing along the polyline).

signedDistanceTo(*point*)

Compute the signed distance of the PolylineRegion to a point.

The distance is positive if the point is left of the nearest segment, and negative otherwise.

Return type
float

pointAlongBy(*distance*, *normalized=False*)

Find the point a given distance along the polyline from its start.

If **normalized** is true, then distance should be between 0 and 1, and is interpreted as a fraction of the length of the polyline. So for example `pointAlongBy(0.5, normalized=True)` returns the polyline's midpoint.

Return type
Vector

__getitem__(*i*)

Get the ith point along this polyline.

If the region consists of multiple polylines, this order is linear along each polyline but arbitrary across different polylines.

Return type
Vector

__len__()

Get the number of vertices of the polyline.

Return type
int

class PathRegion(*points=None*, *polylines=None*, *tolerance=1e-08*, *name=None*)

A region composed of multiple polylines in 3D space.

One of points or polylines should be provided.

Parameters

- **points** – A list of points defining a single polyline.
- **polylines** – A list of list of points, defining multiple polylines.
- **tolerance** – Tolerance used internally.

2D Regions

2D regions represent a 2D shape parallel to the XY plane, at a certain elevation in space. All 2D regions inherit from *PolygonalRegion*.

Unlike the more *PolygonalRegion*, the simple geometric shapes are allowed to depend on random values: for example, the visible region of an *Object* is a *SectorRegion* based at the object's position, which might not be fixed.

Since 2D regions cannot contain an *Object* (which must be 3D), they define a footprint for convenience. Footprints are always a *PolygonalFootprintRegion*, which represents a 2D polygon extruded infinitely in the positive and negative vertical direction. When checking containment of an *Object* in a 2D region, Scenic will atuomatically use the footprint.

class PolygonalRegion(*points=None*, *polygon=None*, *z=0*, *orientation=None*, *name=None*, *additionalDeps=[]*)

Region given by one or more polygons (possibly with holes) at a fixed z coordinate.

The region may be specified by giving either a sequence of points defining the boundary of the polygon, or a collection of `shapely` polygons (a `Polygon` or `MultiPolygon`).

Parameters

- **points** – sequence of points making up the boundary of the polygon (or `None` if using the **polygon** argument instead).

- **polygon** – `shapely` polygon or collection of polygons (or `None` if using the **points** argument instead).
- **z** – The z coordinate the polygon is located at.
- **orientation** (*Vector Field*; optional) – preferred orientation to use.
- **name** (*str; optional*) – name for debugging.

property boundary: *PolylineRegion*

Get the boundary of this region as a *PolylineRegion*.

class CircularRegion(*center*, *radius*, *resolution=32*, *name=None*)

A circular region with a possibly-random center and radius.

Parameters

- **center** (*Vector*) – center of the disc.
- **radius** (*float*) – radius of the disc.
- **resolution** (*int; optional*) – number of vertices to use when approximating this region as a polygon.
- **name** (*str; optional*) – name for debugging.

class SectorRegion(*center*, *radius*, *heading*, *angle*, *resolution=32*, *name=None*)

A sector of a *CircularRegion*.

This region consists of a sector of a disc, i.e. the part of a disc subtended by a given arc.

Parameters

- **center** (*Vector*) – center of the corresponding disc.
- **radius** (*float*) – radius of the disc.
- **heading** (*float*) – heading of the centerline of the sector.
- **angle** (*float*) – angle subtended by the sector.
- **resolution** (*int; optional*) – number of vertices to use when approximating this region as a polygon.
- **name** (*str; optional*) – name for debugging.

class RectangularRegion(*position*, *heading*, *width*, *length*, *name=None*)

A rectangular region with a possibly-random position, heading, and size.

Parameters

- **position** (*Vector*) – center of the rectangle.
- **heading** (*float*) – the heading of the `length` axis of the rectangle.
- **width** (*float*) – width of the rectangle.
- **length** (*float*) – length of the rectangle.
- **name** (*str; optional*) – name for debugging.

3D Regions

3D regions represent points in 3D space.

Most 3D regions inherit from either *MeshVolumeRegion* or *MeshSurfaceRegion*, which represent the volume (of a watertight mesh) and the surface of a mesh respectively. Various region classes are also provided to create primitive shapes. *MeshVolumeRegion* can be converted to *MeshSurfaceRegion* (and vice versa) using the the `getSurfaceRegion` and `getVolumeRegion` methods.

Mesh regions can use one of two engines for mesh operations: Blender or OpenSCAD. This can be controlled using the `engine` parameter, passing `"blender"` or `"scad"` respectively. Blender is generally more tolerant but can produce unreliable output, such as meshes that have microscopic holes. OpenSCAD is generally more precise, but may crash on certain inputs that it considers ill-defined. By default, Scenic uses Blender internally.

PolygonalFootprintRegions represent the footprint of a 2D region. See *2D Regions* for more details.

class MeshVolumeRegion(**args*, ***kwargs*)

Bases: *MeshRegion*

A region representing the volume of a mesh.

The mesh passed must be a `trimesh.base.Trimesh` object that represents a well defined volume (i.e. the `is_volume` property must be true), meaning the mesh must be watertight, have consistent winding and have outward facing normals.

The mesh is first placed so the origin is at the center of the bounding box (unless `centerMesh` is `False`). The mesh is scaled to `dimensions`, translated so the center of the bounding box of the mesh is at `positon`, and then rotated to `rotation`.

Meshes are centered by default (since `centerMesh` is true by default). If you disable this operation, do note that scaling and rotation transformations may not behave as expected, since they are performed around the origin.

Parameters

- **mesh** – The base mesh for this region.
- **name** – An optional name to help with debugging.
- **dimensions** – An optional 3-tuple, with the values representing width, length, height respectively. The mesh will be scaled such that the bounding box for the mesh has these dimensions.
- **position** – An optional position, which determines where the center of the region will be.
- **rotation** – An optional Orientation object which determines the rotation of the object in space.
- **orientation** – An optional vector field describing the preferred orientation at every point in the region.
- **tolerance** – Tolerance for internal computations.
- **centerMesh** – Whether or not to center the mesh after copying and before transformations.
- **onDirection** – The direction to use if an object being placed on this region doesn't specify one.
- **engine** – Which engine to use for mesh operations. Either "blender" or "scad".

getSurfaceRegion()

Return a region equivalent to this one, except as a MeshSurfaceRegion

classmethod fromFile(*path*, *filetype=None*, *compressed=None*, *binary=False*, ***kwargs*)

Load a mesh region from a file, attempting to infer filetype and compression.

For example: "foo.obj.bz2" is assumed to be a compressed .obj file. "foo.obj" is assumed to be an uncompressed .obj file. "foo" is an unknown filetype, so unless a filetype is provided an exception will be raised.

Parameters

- **path** (*str*) – Path to the file to import.
- **filetype** (*str*) – Filetype of file to be imported. This will be inferred if not provided. The filetype must be one compatible with `trimesh.load`.
- **compressed** (*bool*) – Whether or not this file is compressed (with bz2). This will be inferred if not provided.
- **binary** (*bool*) – Whether or not to open the file as a binary file.
- **kwargs** – Additional arguments to the MeshRegion initializer.

class MeshSurfaceRegion(**args*, ***kwargs*)

Bases: *MeshRegion*

A region representing the surface of a mesh.

The mesh is first placed so the origin is at the center of the bounding box (unless `centerMesh` is `False`). The mesh is scaled to `dimensions`, translated so the center of the bounding box of the mesh is at `positon`, and then rotated to `rotation`.

Meshes are centered by default (since `centerMesh` is true by default). If you disable this operation, do note that scaling and rotation transformations may not behave as expected, since they are performed around the origin.

If an orientation is not passed to this mesh, a default orientation is provided which provides an orientation that aligns an object's z axis with the normal vector of the face containing that point, and has a yaw aligned with a yaw of 0 in the global coordinate system.

Parameters

- **mesh** – The base mesh for this region.
- **name** – An optional name to help with debugging.
- **dimensions** – An optional 3-tuple, with the values representing width, length, height respectively. The mesh will be scaled such that the bounding box for the mesh has these dimensions.
- **position** – An optional position, which determines where the center of the region will be.
- **rotation** – An optional Orientation object which determines the rotation of the object in space.
- **orientation** – An optional vector field describing the preferred orientation at every point in the region.
- **tolerance** – Tolerance for internal computations.
- **centerMesh** – Whether or not to center the mesh after copying and before transformations.
- **onDirection** – The direction to use if an object being placed on this region doesn't specify one.

getVolumeRegion()

Return a region equivalent to this one, except as a MeshVolumeRegion

classmethod fromFile(*path*, *filetype=None*, *compressed=None*, *binary=False*, ***kwargs*)

Load a mesh region from a file, attempting to infer filetype and compression.

For example: "foo.obj.bz2" is assumed to be a compressed .obj file. "foo.obj" is assumed to be an uncompressed .obj file. "foo" is an unknown filetype, so unless a filetype is provided an exception will be raised.

Parameters

- **path** (*str*) – Path to the file to import.
- **filetype** (*str*) – Filetype of file to be imported. This will be inferred if not provided. The filetype must be one compatible with `trimesh.load`.
- **compressed** (*bool*) – Whether or not this file is compressed (with bz2). This will be inferred if not provided.
- **binary** (*bool*) – Whether or not to open the file as a binary file.
- **kwargs** – Additional arguments to the MeshRegion initializer.

class BoxRegion(**args*, ***kwargs*)

Region in the shape of a rectangular cuboid, i.e. a box.

By default the unit box centered at the origin and aligned with the axes is used.

Parameters are the same as *MeshVolumeRegion*, with the exception of the `mesh` parameter which is excluded.

class SpheroidRegion(**args*, ***kwargs*)

Region in the shape of a spheroid.

By default the unit sphere centered at the origin and aligned with the axes is used.

Parameters are the same as *MeshVolumeRegion*, with the exception of the `mesh` parameter which is excluded.

class PolygonalFootprintRegion(*polygon*, *name=None*)

Region that contains all points in a polygonal footprint, regardless of their z value.

This region cannot be sampled from, as it has infinite height and therefore infinite volume.

Parameters

- **polygon** – A `shapely` `Polygon` or `MultiPolygon`, that defines the footprint of this region.
- **name** – An optional name to help with debugging.

Niche Regions

class GridRegion(*name*, *grid*, *Ax*, *Ay*, *Bx*, *By*, *orientation=None*)

Bases: *PointSetRegion*

A Region given by an obstacle grid.

A point is considered to be in a *GridRegion* if the nearest grid point is not an obstacle.

Parameters

- **name** (*str*) – name for debugging
- **grid** – 2D list, tuple, or NumPy array of 0s and 1s, where 1 indicates an obstacle and 0 indicates free space
- **Ax** (*float*) – spacing between grid points along X axis
- **Ay** (*float*) – spacing between grid points along Y axis

- **Bx** (*float*) – X coordinate of leftmost grid column
- **By** (*float*) – Y coordinate of lowest grid row
- **orientation** (*Vector Field*; optional) – orientation of region

1.8.4 Distributions Reference

Scenic provides functions for sampling from various types of probability distributions, and it is also possible to define custom types of distributions.

If you want to sample multiple times from the same distribution (for example if the distribution is passed as an argument to a helper function), you can use the *resample* function.

Built-in Distributions

Range(*low*, *high*)

Uniformly-distributed real number in the interval.

DiscreteRange(*low*, *high*)

Uniformly-distributed integer in the (fixed) interval.

Normal(*mean*, *stdDev*)

Normal distribution with the given mean and standard deviation.

TruncatedNormal(*mean*, *stdDev*, *low*, *high*)

Normal distribution as above, but truncated to the given window.

Uniform(*value*, ...)

Uniform over a finite set of values. The Uniform distribution can also be used to uniformly select over a list of unknown length. This can be done using the unpacking operator (which supports distributions over lists) as follows: `Uniform(*list)`.

Discrete({*value*: *weight*, ... })

Discrete distribution over a finite set of values, with weights (which need not add up to 1). Each value is sampled with probability proportional to its weight.

Uniform Distribution over a Region

Scenic can also sample points uniformly at random from a *Region*, using the `in region` and `on region` specifiers. Most subclasses of *Region* support random sampling. A few regions, such as the `everywhere` region representing all space, cannot be sampled from since a uniform distribution over them does not exist.

Defining Custom Distributions

If necessary, custom distributions may be implemented by subclassing the `Distribution` class. New subclasses must implement the `sampleGiven` method, which computes a random sample from the distribution given values for its dependencies (if any). See `Range` (the implementation of the uniform distribution over a range of real numbers) for a simple example of how to define a subclass. Additional functionality can be enabled by implementing the optional `clone`, `bucket`, and `supportInterval` methods; see their documentation for details.

1.8.5 Statements Reference

Compound Statements

Class Definition

```
class <name>[(<superclass>)]:
    [<property>: <value>]*
```

Defines a Scenic class. If a superclass is not explicitly specified, *Object* is used (see *Objects and Classes Reference*). The body of the class defines a set of properties its objects have, together with default values for each property. Properties are inherited from superclasses, and their default values may also use the special syntax `self.property` to refer to one of the other properties of the same object, which is then a *dependency* of the default value. The order in which to evaluate properties satisfying all dependencies is computed (and cyclic dependencies detected) during *Specifier Resolution*.

Scenic classes may also define attributes and methods in the same way as Python classes.

Behavior Definition

```
behavior <name>(<arguments>):
    [precondition: <boolean>]*
    [invariant: <boolean>]*
    <statement>+
```

Defines a dynamic behavior, which can be assigned to a Scenic object by setting its `behavior` property using the `with behavior behavior` specifier; this makes the object an agent. See our tutorial on *Dynamic Scenarios* for examples of how to write behaviors.

Behavior definitions have the same form as function definitions, with an argument list and a body consisting of one or more statements; the body may additionally begin with definitions of preconditions and invariants. Preconditions are checked when a behavior is started, and invariants are checked at every time step of the simulation while the behavior is executing (including time step zero, like preconditions, but *not* including time spent inside sub-behaviors: this allows sub-behaviors to break and restore invariants before they return).

The body of a behavior executes in parallel with the simulation: in each time step, it must either `take` specified action(s) or `wait` and perform no actions. After each `take` or `wait` statement, the behavior's execution is suspended,

the simulation advances one step, and the behavior is then resumed. It is thus an error for a behavior to enter an infinite loop which contains no *take* or *wait* statements (or *do* statements invoking a sub-behavior; see below): the behavior will never yield control to the simulator and the simulation will stall.

Behaviors end naturally when their body finishes executing (or if they `return`): if this happens, the agent performing the behavior will take no actions for the rest of the scenario. Behaviors may also *terminate* the current scenario, ending it immediately.

Behaviors may invoke sub-behaviors, optionally for a limited time or until a desired condition is met, using *do* statements. It is also possible to (temporarily) interrupt the execution of a sub-behavior under certain conditions and resume it later, using try-interrupt statements.

Monitor Definition

```
monitor <name>(<arguments>):
    <statement>+
```

Defines a type of monitor, which can be run in parallel with the simulation like a dynamic behavior. Monitors are not associated with an *Object* and cannot take actions, but can *wait* to wait for the next time step (or use *terminate* or *terminate simulation* to end the scenario/simulation). A monitor can be instantiated in a scenario with the *require monitor* statement.

The main purpose of monitors is to evaluate complex temporal properties that are not expressible using the temporal operators available for *require* `LTL formula` statements. They can maintain state and use *require* to enforce requirements depending on that state. For examples of monitors, see our tutorial on *Dynamic Scenarios*.

Changed in version 3.0: Monitors may take arguments, and must be explicitly instantiated using a *require monitor* statement.

Modular Scenario Definition

```
scenario <name>(<arguments>):
    [precondition: <boolean>]*
    [invariant: <boolean>]*
    [setup:
        <statement>+]
    [compose:
        <statement>+]
```

```
scenario <name>(<arguments>):
    <statement>+
```

Defines a Scenic modular scenario. Scenario definitions, like *behavior definitions*, may include preconditions and invariants. The body of a scenario consists of two optional parts: a `setup` block and a `compose` block. The `setup` block contains code that runs once when the scenario begins to execute, and is a list of statements like a top-level Scenic program (so it may create objects, define requirements, etc.). The `compose` block orchestrates the execution of sub-scenarios during a dynamic scenario, and may use *do* and any of the other statements allowed inside behaviors (except *take*, which only makes sense for an individual agent). If a modular scenario does not use preconditions, invariants, or sub-scenarios (i.e., it only needs a `setup` block) it may be written in the second form above, where the entire body of the `scenario` comprises the `setup` block.

See also:

Our tutorial on *Composing Scenarios* gives many examples of how to use modular scenarios.

Try-Interrupt Statement

```
try:
    <statement>+
[interrupt when <boolean>:
    <statement>+]*
[except <exception> [as <name>]:
    <statement>+]*
```

A `try-interrupt` statement can be placed inside a behavior (or *compose* block of a modular scenario) to run a series of statements, including invoking sub-behaviors with *do*, while being able to interrupt at any point if given conditions are met. When a `try-interrupt` statement is encountered, the statements in the `try` block are executed. If at any time step one of the `interrupt` conditions is met, the corresponding `interrupt` block (its *handler*) is entered and run. Once the interrupt handler is complete, control is returned to the statement that was being executed under the `try` block.

If there are multiple `interrupt` clauses, successive clauses take precedence over those which precede them; furthermore, during execution of an interrupt handler, successive `interrupt` clauses continue to be checked and can interrupt the handler. Likewise, if `try-interrupt` statements are nested, the outermost statement takes precedence and can interrupt the inner statement at any time. When one handler interrupts another and then completes, the original handler is resumed (and it may even be interrupted again before control finally returns to the `try` block).

The `try-interrupt` statement may conclude with any number of `except` blocks, which function identically to their Python counterparts.

Simple Statements

The following statements can occur throughout a Scenic program unless otherwise stated.

model *name*

Select a world model to use for this scenario. The statement `model X` is equivalent to `from X import *` except that `X` can be replaced using the `--model` command-line option or the `model` keyword argument to the top-level APIs. When writing simulator-agnostic scenarios, using the `model` statement is preferred to a simple `import` since a more specific world model for a particular simulator can then be selected at compile time.

import *module*

Import a Scenic or Python module. This statement behaves *as in Python*, but when importing a Scenic module it also imports any objects created and requirements imposed in that module. Scenic also supports the form `from module import identifier, ...`, which as in Python imports the module plus one or more identifiers from its namespace.

param *name* = *value*, ...

Defines one or more global parameters of the scenario. These have no semantics in Scenic, simply having their values included as part of the generated *Scene*, but provide a general-purpose way to encode arbitrary global information.

If multiple `param` statements define parameters with the same name, the last statement takes precedence, except that Scenic world models imported using the *model* statement do not override existing values for global parameters. This allows models to define default values for parameters which can be overridden by particular scenarios. Global parameters can also be overridden at the command line using the *--param* option, or from the top-level API using the `params` argument to *scenic.scenarioFromFile*.

To access global parameters within the scenario itself, you can read the corresponding attribute of the *globalParameters* object. For example, if you declare `param weather = 'SUNNY'`, you could then access this parameter later in the program via *globalParameters*.`weather`. If the parameter was not overridden, this would evaluate to `'SUNNY'`; if Scenic was run with the command-line option `--param weather SNOW`, it would evaluate to `'SNOW'` instead.

Some simulators provide global parameters whose names are not valid identifiers in Scenic. To support giving values to such parameters without renaming them, Scenic allows the names of global parameters to be quoted strings, as in this example taken from an *X-Plane* scenario:

```
param simulation_length = 30
param 'sim/weather/cloud_type[0]' = DiscreteRange(0, 5)
param 'sim/weather/rain_percent' = 0
```

require *boolean*

Defines a hard requirement, requiring that the given condition hold in all instantiations of the scenario. This is equivalent to an "observe" statement in other probabilistic programming languages.

require[*number*] *boolean*

Defines a soft requirement; like *require* above but enforced only with the given probability, thereby requiring that the given condition hold with at least that probability (which must be a literal number, not an expression). For example, `require[0.75] ego in parking_lot` would require that the ego be in the parking lot at least 75% percent of the time.

require *LTL formula*

Defines a temporal requirement, requiring that the given Linear Temporal Logic formula hold in a dynamic scenario. See *Temporal Operators* for the list of supported LTL operators.

Note that an expression that does not use any temporal operators is evaluated only in the current time step. So for example:

- `require A and always B` will only require that `A` hold at time step zero, while `B` must hold at every time step (note that this is the same behavior you would get if you wrote `require A` and `require always B` separately);
- `require (always A) implies B` requires that if `A` is true at every time step, then `B` must be true at time step zero;
- `require always A implies B` requires that in *every* time step when `A` is true, `B` must also be true (since `B` is within the scope of the *always* operator).

require monitor *monitor*

Require a condition encoded by a monitor hold during the scenario. See *Monitor Definition* for how to define types of monitors.

It is legal to create multiple instances of a monitor with varying parameters. For example:

```
monitor ReachesBefore(obj1, region, obj2):
    reached = False
    while not reached:
        if obj1 in region:
            reached = True
        else:
            require obj2 not in region
            wait

require monitor ReachesBefore(ego, goal, racecar2)
require monitor ReachesBefore(ego, goal, racecar3)
```

terminate when *boolean*

Terminates the scenario when the provided condition becomes true. If this statement is used in a modular scenario which was invoked from another scenario, only the current scenario will end, not the entire simulation.

terminate simulation when *boolean*

The same as `terminate when`, except terminates the entire simulation even when used inside a sub-scenario (so there is no difference between the two statements when used at the top level).

terminate after *scalar* (seconds | steps)

Like `terminate when` above, but terminates the scenario after the given amount of time. The time limit can be an expression, but must be a non-random value.

mutate *identifier*, ... [by *scalar*]

Enables mutation of the given list of objects (any *Point*, *OrientedPoint*, or *Object*), with an optional scale factor (default 1). If no objects are specified, mutation applies to every *Object* already created.

The default mutation system adds Gaussian noise to the `position` and `heading` properties, with standard deviations equal to the scale factor times the `positionStdDev` and `headingStdDev` properties.

Note: User-defined classes may specify custom mutators to allow mutation to apply to properties other than `position` and `heading`. This is done by providing a value for the `mutator` property, which should be an instance of `Mutator`. Mutators inherited from superclasses (such as the default `position` and `heading` mutators from *Point* and *OrientedPoint*) will still be applied unless the new mutator disables them; see `Mutator` for details.

---

record [initial | final] *value* [as *name*]

Record the value of an expression during each simulation. The value can be recorded at the start of the simulation (`initial`), at the end of the simulation (`final`), or at every time step (if neither `initial` nor `final` is specified). The recorded values are available in the `records` dictionary of *`SimulationResult`*: its keys are the given names of the records (or synthesized names if not provided), and the corresponding values are either the value of the recorded expression or a tuple giving its value at each time step as appropriate. For debugging, the records can also be printed out using the *`--show-records`* command-line option.

Dynamic Statements

The following statements are valid only in dynamic behaviors, monitors, and *`compose`* blocks.

take *action*, ...

Takes the action(s) specified and pass control to the simulator until the next time step. Unlike *`wait`*, this statement may not be used in monitors or modular scenarios, since these do not take actions.

wait

Take no actions this time step.

terminate

Immediately end the scenario. As for *`terminate when`*, if this statement is used in a modular scenario which was invoked from another scenario, only the current scenario will end, not the entire simulation. Inside a behavior being run by an agent, the "current scenario" for this purpose is the scenario which created the agent.

terminate simulation

Immediately end the entire simulation.

do *behavior/scenario*, ...

Run one or more sub-behaviors or sub-scenarios in parallel. This statement does not return until all invoked sub-behaviors/scenarios have completed.

do *behavior/scenario*, ... until *boolean*

As above, except the sub-behaviors/scenarios will terminate when the condition is met.

do *behavior/scenario* for *scalar* (seconds | steps)

Run sub-behaviors/scenarios for a set number of simulation seconds/time steps. This statement can return before that time if all the given sub-behaviors/scenarios complete.

do choose *behavior/scenario*, ...

Randomly pick one of the given behaviors/scenarios whose preconditions are satisfied, and run it. If no choices are available, the simulation is rejected.

This statement also allows the more general form `do choose { behaviorOrScenario: weight, ... }`, giving weights for each choice (which need not add up to 1). Among all choices whose preconditions are satisfied, this picks a choice with probability proportional to its weight.

do shuffle *behavior/scenario*, ...

Like `do choose` above, except that when the chosen sub-behavior/scenario completes, a different one whose preconditions are satisfied is chosen to run next, and this repeats until all the sub-behaviors/scenarios have run once. If at any point there is no available choice to run (i.e. we have a deadlock), the simulation is rejected.

This statement also allows the more general form `do shuffle { behaviorOrScenario: weight, ... }`, giving weights for each choice (which need not add up to 1). Each time a new sub-behavior/scenario needs to be selected, this statement finds all choices whose preconditions are satisfied and picks one with probability proportional to its weight.

abort

Used in an interrupt handler to terminate the current `try-interrupt` statement.

override *object specifier*, ...

Override one or more properties of an object, e.g. its `behavior`, for the duration of the current scenario. The properties will revert to their previous values when the current scenario terminates. It is illegal to override dynamic properties, since they are set by the simulator each time step and cannot be mutated manually.

1.8.6 Objects and Classes Reference

This page describes the classes built into Scenic, representing *points*, *oriented points*, and physical *objects*, and how they are instantiated to create objects.

Note: The documentation given here describes only the public properties and methods provided by the built-in classes. If you are working on Scenic's internals, you can find more complete documentation in the `scenic.core.object_types` module.

Instance Creation

```
new <class> [<specifier> [, <specifier>]*]
```

Instantiates a Scenic object from a Scenic class. The properties of the object are determined by the given set of zero or more specifiers. For details on the available specifiers and how they interact, see the *Specifiers Reference*.

Instantiating an instance of *Object* has a side effect: the object is added to the scenario being defined.

Changed in version 3.0: Instance creation now requires the `new` keyword. As a result, Scenic classes can be referred to without creating an instance.

Built-in Classes

Point

Locations in space. This class provides the fundamental property `position` and several associated properties.

`class Point` <specifiers>

The Scenic base class `Point`.

The default mutator for *Point* adds Gaussian noise to `position` with a standard deviation given by the `positionStdDev` property.

Properties

- **position** (*Vector*; dynamic) – Position of the point. Default value is the origin (0,0,0).
- **width** (*float*) – Default value 0 (only provided for compatibility with operators that expect an *Object*).
- **length** (*float*) – Default value 0.
- **height** (*float*) – Default value 0.
- **baseOffset** (*Vector*) – Only provided for compatibility with the *on region* specifier. Default value is (0,0,0).
- **contactTolerance** (*float*) – Only provided for compatibility with the specifiers that expect an *Object*. Default value 0.
- **onDirection** (*Vector*) – The direction used to determine where to place this *Point* on a region, when using the modifying *on* specifier. See the *on region* page for more details. Default value is None, indicating the direction will be inferred from the region this object is being placed on.
- **visibleDistance** (*float*) – Distance used to determine the visible range of this object. Default value 50.
- **viewRayDensity** (*float*) – By default determines the number of rays used during visibility checks. This value is the density of rays per degree of visible range in one dimension. The total number of rays sent will be this value squared per square degree of this object's view angles. This value determines the default value for `viewRayCount`, so if `viewRayCount` is overwritten this value is ignored. Default value 5.
- **viewRayCount** (*None | tuple[float, float]*) – The total number of horizontal and vertical view angles to be sent, or None if this value should be computed automatically. Default value `None`.

- **viewRayDistanceScaling** (*bool*) – Whether or not the number of rays should scale with the distance to the object. Ignored if `viewRayCount` is passed. Default value `False`.
- **mutationScale** (*float*) – Overall scale of mutations, as set by the `mutate` statement. Default value 0 (mutations disabled).
- **positionStdDev** (*tuple[float, float, float]*) – Standard deviation of Gaussian noise for each dimension (x,y,z) to be added to this object's `position` when mutation is enabled with scale 1. Default value (1,1,0), mutating only the x,y values of the point.

property visibleRegion

The visible region of this object.

The visible region of a *Point* is a sphere centered at its `position` with radius `visibleDistance`.

OrientedPoint

A location along with an orientation, defining a local coordinate system. This class subclasses *Point*, adding the fundamental property `orientation` and several associated properties.

class OrientedPoint <specifiers>

The Scenic class `OrientedPoint`.

The default mutator for *OrientedPoint* adds Gaussian noise to `yaw` while leaving `pitch` and `roll` unchanged, using the three standard deviations (for yaw/pitch/roll respectively) given by the `orientationStdDev` property. It then also applies the mutator for *Point*.

The default mutator for *OrientedPoint* adds Gaussian noise to `yaw`, `pitch` and `roll` according to `orientationStdDev`. By default the standard deviations for `pitch` and `roll` are zero so that, by default, only `yaw` is mutated.

Properties

- **yaw** (*float; dynamic*) – Yaw of the *OrientedPoint* in radians in the local coordinate system provided by `parentOrientation`. Default value 0.
- **pitch** (*float; dynamic*) – Pitch of the *OrientedPoint* in radians in the local coordinate system provided by `parentOrientation`. Default value 0.
- **roll** (*float; dynamic*) – Roll of the *OrientedPoint* in radians in the local coordinate system provided by `parentOrientation`. Default value 0.
- **parentOrientation** (*Orientation*) – The local coordinate system that the *OrientedPoint*'s `yaw`, `pitch`, and `roll` are interpreted in. Default value is the global coordinate system, where an object is flat in the XY plane, facing North.
- **orientation** (*Orientation*; dynamic; final) – The orientation of the *OrientedPoint* relative to the global coordinate system. Derived from the `yaw`, `pitch`, `roll`, and `parentOrientation` of this *OrientedPoint* and non-overridable.
- **heading** (*float; dynamic; final*) – Yaw value of this *OrientedPoint* in the global coordinate system. Derived from `orientation` and non-overridable.
- **viewAngles** (*tuple[float,float]*) – Horizontal and vertical view angles of this *OrientedPoint* in radians. Horizontal view angle can be up to 2 and vertical view angle can be up to . Values greater than these will be truncated. Default value is (2,)
- **orientationStdDev** (*tuple[float,float,float]*) – Standard deviation of Gaussian noise to add to this object's Euler angles (yaw, pitch, roll) when mutation is enabled with scale 1. Default value (5°, 0, 0), mutating only the `yaw` of this *OrientedPoint*.

property visibleRegion

The visible region of this object.

The visible region of an *OrientedPoint* restricts that of *Point* (a sphere with radius `visibleDistance`) based on the value of `viewAngles`. In general, it is a capped rectangular pyramid subtending an angle of `viewAngles[0]` horizontally and `viewAngles[1]` vertically, as long as those angles are less than /2; larger angles yield various kinds of wrap-around regions. See *`ViewRegion`* for details.

Object

A physical object. This class subclasses *OrientedPoint*, adding a variety of properties including:

- `width`, `length`, and `height` to define the dimensions of the object;
- `shape` to define the *Shape* of the object;
- `allowCollisions`, `requireVisible`, and `regionContainedIn` to control the built-in requirements that apply to the object;
- `behavior`, specifying the object's dynamic behavior if any;
- `speed`, `velocity`, and other properties capturing the dynamic state of the object during simulations.

The built-in requirements applying to each object are:

- The object must be completely contained within its container, the region specified as its `regionContainedIn` property (by default the entire workspace).
- The object must be visible from the ego object if the `requireVisible` property is set to `True` (default value `False`).
- The object must not intersect another object (i.e., their bounding boxes must not overlap), unless either of the two objects has their `allowCollisions` property set to `True`.

Changed in version 3.0: `requireVisible` is now `False` by default.

class Object <specifiers>

The Scenic class `Object`.

This is the default base class for Scenic classes.

Properties

- **width** (*float*) – Width of the object, i.e. extent along its X axis. Default value of 1 inherited from the object's `shape`.
- **length** (*float*) – Length of the object, i.e. extent along its Y axis. Default value of 1 inherited from the object's `shape`.
- **height** (*float*) – Height of the object, i.e. extent along its Z axis. Default value of 1 inherited from the object's `shape`.
- **shape** (*Shape*) – The shape of the object, which must be an instance of *Shape*. The default shape is a box, with default unit dimensions.
- **allowCollisions** (*bool*) – Whether the object is allowed to intersect other objects. Default value `False`.
- **regionContainedIn** (*Region* or `None`) – A *Region* the object is required to be contained in. If `None`, the object need only be contained in the scenario's workspace.

- **baseOffset** (*Vector*) – An offset from the `position` of the Object to the base of the object, used by the *on region* specifier. Default value is `(0, 0, -self.height/2)`, placing the base of the Object at the bottom center of the Object's bounding box.
- **contactTolerance** (*float*) – The maximum distance this object can be away from a surface to be considered on the surface. Objects are placed at half this distance away from a point when the *on region* specifier or a directional specifier like *(left | right) of Object [by scalar]* is used. Default value 1e-4.
- **sideComponentThresholds** (*DimensionLimits*) – Used to determine the various sides of an object (when using the default implementation). The three interior 2-tuples represent the maximum and minimum bounds for each dimension's (x,y,z) surface. See *defaultSideSurface* for details. Default value `((-0.5, 0.5), (-0.5, 0.5), (-0.5, 0.5))`.
- **cameraOffset** (*Vector*) – Position of the camera for the *can see* operator, relative to the object's `position`. Default `(0, 0, 0)`.
- **requireVisible** (*bool*) – Whether the object is required to be visible from the `ego` object. Default value `False`.
- **occluding** (*bool*) – Whether or not this object can occlude other objects. Default value `True`.
- **showVisibleRegion** (*bool*) – Whether or not to display the visible region in the Scenic internal visualizer.
- **color** (tuple[float, float, float, float] or tuple[float, float, float] or `None`) – An optional color (with optional alpha) property that is used by the internal visualizer, or possibly simulators. All values should be between 0 and 1. Default value `None`
- **velocity** (*Vector*; *dynamic*) – Velocity in dynamic simulations. Default value is the velocity determined by `speed` and `orientation`.
- **speed** (*float; dynamic*) – Speed in dynamic simulations. Default value 0.
- **angularVelocity** (*Vector*; *dynamic*)
- **angularSpeed** (*float; dynamic*) – Angular speed in dynamic simulations. Default value 0.
- **behavior** – Behavior for dynamic agents, if any (see *Dynamic Scenarios*). Default value `None`.
- **lastActions** – Tuple of actions taken by this agent in the last time step (or `None` if the object is not an agent or this is the first time step).

startDynamicSimulation()

Hook called when the object is created in a dynamic simulation.

Does nothing by default; provided for objects to do simulator-specific initialization as needed.

Changed in version 3.0: This method is called on objects created in the middle of dynamic simulations, not only objects present in the initial scene.

property visibleRegion

The visible region of this object.

The visible region of an *Object* is the same as that of an *OrientedPoint* (see *OrientedPoint.visibleRegion*) except that it is offset by the value of `cameraOffset` (which is the zero vector by default).

1.8.7 Specifiers Reference

Specifiers are used to define the properties of an object when a Scenic class is *instantiated*. This page describes all the specifiers built into Scenic, and the procedure used to *resolve* a set of specifiers into an assignment of values to properties.

Each specifier assigns values to one or more properties of an object, as a function of the arguments of the specifier and possibly other properties of the object assigned by other specifiers. For example, the `left of X by Y` specifier assigns the `position` property of the object being defined so that the object is a distance `Y` to the left of `X`: this requires knowing the `width` of the object first, so we say the `left of` specifier **specifies** the `position` property and **depends** on the `width` property.

In fact, the `left of` specifier also specifies the `parentOrientation` property (to be the `orientation` of `X`), but it does this with a lower **priority**. Multiple specifiers can specify the same property, but only the specifier that specifies the property with the highest priority is used. If a property is specified multiple times with the same priority, an ambiguity error is raised. We represent priorities as integers, with priority 1 being the highest and larger integers having progressively lower priorities (e.g. priority 2 supersedes priority 3). When a specifier specifies a property with a priority lower than 1, we say it **optionally** specifies the property, since it can be overridden (for example using the `with` specifier), whereas a specifier specifying the property with priority 1 cannot be overridden.

Certain specifiers can also *modify* already-specified values. These **modifying specifiers** do not cause an ambiguity error as above if another specifier specifies the same property with the same priority: they take the already-specified value and manipulate it in some way (potentially also specifying other properties as usual). Note that no property can be modified twice. The only modifying specifier currently in Scenic is `on region`, which can be used either as a standard specifier or a modifying specifier (the modifying version projects the already-specified position onto the given region – see below).

The *Specifier Resolution* process works out which specifier determines each property of an object, as well as an appropriate order in which to evaluate the specifiers so that dependencies have already been computed when needed.

General Specifiers

with *property value*

Specifies:

- the given property, with priority 1

Dependencies: None

Assigns the given property to the given value. This is currently the only specifier available for properties other than `position` and `orientation`.

Position Specifiers

at *vector*

Specifies:

- `position` with priority 1

Dependencies: None

Positions the object at the given global coordinates.

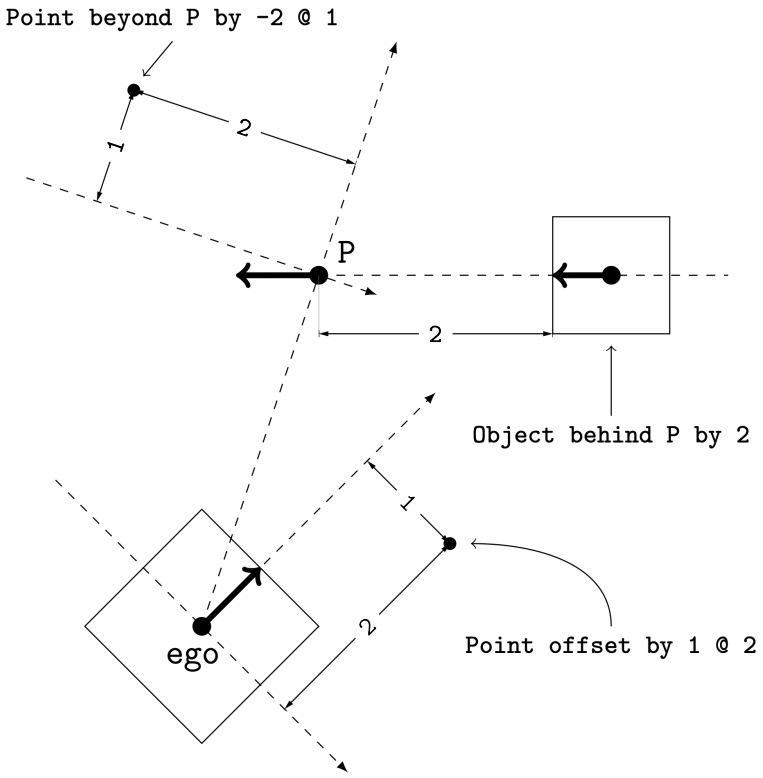


Fig. 6: Illustration of the `beyond`, `behind`, and `offset by` specifiers. Each *`OrientedPoint`* (e.g. `P`) is shown as a bold arrow.

in *region*

Specifies:

- `position` with priority 1
- `parentOrientation` with priority 3 (if the region has a preferred orientation)

Dependencies: None

Positions the object uniformly at random in the given *Region*. If the Region has a preferred orientation (a vector field), also specifies `parentOrientation` to be equal to that orientation at the object's `position`.

contained in *region*

Specifies:

- `position` with priority 1
- `regionContainedIn` with priority 1
- `parentOrientation` with priority 3 (if the region has a preferred orientation)

Dependencies: None

Like *`in region`*, but also enforces that the object be entirely contained in the given *Region*.

on *region*

Specifies:

- `position` with priority 1; **modifies** existing value, if any
- `parentOrientation` with priority 2 (if the region has a preferred orientation)

Dependencies: `baseOffset` • `contactTolerance` • `onDirection`

If `position` is not already specified with priority 1, positions the *base* of the object uniformly at random in the given *Region*, offset by `contactTolerance` (to avoid a collision). The base of the object is determined by adding the object's `baseOffset` to its `position`.

Note that while `on` can be used with *Region*, *Object* and *Vector*, it cannot be used with a distribution containing anything other than *Region*. When used with an object the base of the object being placed is placed on the target object's *`onSurface`* and when used with a vector the base of the object being placed is set to that position.

If instead `position` has already been specified with priority 1, then its value is modified by projecting it onto the given region. More precisely, we find the closest point in the region along `onDirection` (or its negation[1]), and place the base of the object at that point. If `onDirection` is not specified, a default value is inferred from the region. A region can either specify a default value to be used, or for volumes straight up is used and for surfaces the mean of the face normal values is used (weighted by the area of the faces).

If the region has a preferred orientation (a vector field), `parentOrientation` is specified to be equal to that orientation at the object's `position` (whether or not this specifier is being used as a modifying specifier). Note that this is done with higher priority than all other specifiers which optionally specify `parentOrientation`, and in particular the `ahead of` specifier and its variants: therefore the code `new Object ahead of taxi by 100, on road` aligns the new object with the road at the point 100 m ahead of the taxi rather than with the taxi itself (while also using projection to ensure the new object is on the surface of the road rather than under or over it if the road isn't flat).

offset by *vector*

Specifies:

- `position` with priority 1
- `parentOrientation` with priority 3

Dependencies: None

Positions the object at the given coordinates in the local coordinate system of *`ego`* (which must already be defined). Also specifies `parentOrientation` to be equal to the ego's orientation.

New in version 3.0: `offset by` now specifies `parentOrientation`, whereas previously it did *not* optionally specify `heading`.

[1] This allows for natural projection even when an object is below the desired surface, such as placing a car, ahead of another car, on an uphill road.

offset along *direction* by *vector*

Specifies:

- `position` with priority 1
- `parentOrientation` with priority 3

Dependencies: None

Positions the object at the given coordinates in a local coordinate system centered at *ego* and oriented along the given direction (which can be a *Heading*, an *Orientation*, or a *Vector Field*). Also specifies `parentOrientation` to be equal to the ego's orientation.

beyond *vector* by (*vector* | *scalar*) [from (*vector* | *OrientedPoint*)]

Specifies:

- `position` with priority 1
- `parentOrientation` with priority 3

Dependencies: None

Positions the object at coordinates given by the second vector, in a local coordinate system centered at the first vector and oriented along the line of sight from the third vector (i.e. an orientation of `(0,0,0)` in the local coordinate system faces directly away from the third vector). If the second argument is a scalar *D* instead of a vector, it is interpreted as the vector `(0, D, 0)`: thus `beyond X by D from Y` places the new object a distance of *D* behind *X* from the perspective of *Y*. If no third argument is provided, it is assumed to be the *ego*.

The value of `parentOrientation` is specified to be the orientation of the third argument if it is an *OrientedPoint* (including *Object* such as *ego*); otherwise the global coordinate system is used. For example, `beyond taxi by (1, 3, 0)` means 3 meters behind the taxi and one meter to the right as viewed by the *ego*.

visible [from (*Point* | *OrientedPoint*)]

Specifies:

- `position` with priority 3
- also adds a requirement (see below)

Dependencies: None

Requires that this object is visible from the *ego* or the given *Point*/*OrientedPoint*. See the *Visibility System* reference for a discussion of the visibility model.

Also optionally specifies `position` to be a uniformly random point in the visible region of the ego, or of the given Point/OrientedPoint if given. Note that the position set by this specifier is slightly stricter than simply adding a requirement that the ego `can see` the object: the specifier makes the *center* of the object (its `position`) visible, while the `can see` condition will be satisfied even if the center is not visible as long as some other part of the object is visible.

not visible [from (*Point* | *OrientedPoint*)]

Specifies:

- `position` with priority 3
- also adds a requirement (see below)

Dependencies: `regionContainedIn`

Requires that this object is *not* visible from the ego or the given *Point*/*OrientedPoint*.

Similarly to `visible [from (Point | OrientedPoint)]`, this specifier can position the object uniformly at random in the *non-visible* region of the ego. However, it depends on `regionContainedIn`, in order to restrict the non-visible region to the container of the object being created, which is hopefully a bounded region (if the non-visible region is unbounded, it cannot be uniformly sampled from and an error will be raised).

(left | right) of (*vector*) [by *scalar*]

Specifies:

- `position` with priority 1

Dependencies: `width` • `orientation`

Without the optional `by scalar`, positions the object immediately to the left/right of the given position; i.e., so that the midpoint of the right/left side of the object's bounding box is at that position. If `by scalar` is used, the object is placed further to the left/right by the given distance.

(left | right) of *OrientedPoint* [by *scalar*]

Specifies:

- `position` with priority 1
- `parentOrientation` with priority 3

Dependencies: `width`

Positions the object to the left/right of the given *OrientedPoint*. Also inherits `parentOrientation` from the given *OrientedPoint*.

(left | right) of *Object* [by *scalar*]

Specifies:

- `position` with priority 1
- `parentOrientation` with priority 3

Dependencies: `width` • `contactTolerance`

Positions the object to the left/right of the given *Object*. This accounts for both objects' dimensions, placing them so that the distance between their bounding boxes is exactly the desired scalar distance (or `contactTolerance` if `by scalar` is not used). Also inherits `parentOrientation` from the given *OrientedPoint*.

(ahead of | behind) *vector* [by *scalar*]

Specifies:

- `position` with priority 1

Dependencies: `length` • `orientation`

Without the optional `by` *`scalar`*, positions the object immediately ahead of/behind the given position; i.e., so that the midpoint of the front/back side of the object's bounding box is at that position. If `by` *`scalar`* is used, the object is placed further ahead/behind by the given distance.

(ahead of | behind) *OrientedPoint* [by *scalar*]

Specifies:

- `position` with priority 1
- `parentOrientation` with priority 3

Dependencies: `length`

Positions the object ahead of/behind the given *OrientedPoint*. Also inherits `parentOrientation` from the given *OrientedPoint*.

(ahead of | behind) *Object* [by *scalar*]

Specifies:

- `position` with priority 1
- `parentOrientation` with priority 3

Dependencies: `length` • `contactTolerance`

Positions the object ahead of/behind the given *Object*. This accounts for both objects' dimensions, placing them so that the distance between their bounding boxes is exactly the desired scalar distance (or `contactTolerance` if `by` *`scalar`* is not used). Also inherits `parentOrientation` from the given *OrientedPoint*.

(above | below) *vector* [by *scalar*]

Specifies:

- `position` with priority 1

Dependencies: `height` • `orientation`

Without the optional `by` *`scalar`*, positions the object immediately above/below the given position; i.e., so that the midpoint of the top/bottom side of the object's bounding box is at that position. If `by` *`scalar`* is used, the object is placed further above/below by the given distance.

(above | below) *OrientedPoint* [by *scalar*]

Specifies:

- `position` with priority 1
- `parentOrientation` with priority 3

Dependencies: `height`

Positions the object above/below the given *OrientedPoint*. Also inherits `parentOrientation` from the given *OrientedPoint*.

(above | below) *Object* [by *scalar*]

Specifies:

- `position` with priority 1
- `parentOrientation` with priority 3

Dependencies: `height` • `contactTolerance`

Positions the object above/below the given *Object*. This accounts for both objects' dimensions, placing them so that the distance between their bounding boxes is exactly the desired scalar distance (or `contactTolerance` if `by` *`scalar`* is not used). Also inherits `parentOrientation` from the given *OrientedPoint*.

following *vectorField* [from *vector*] for *scalar*

Specifies:

- `position` with priority 1
- `parentOrientation` with priority 3

Dependencies: None

Positions the object at a point obtained by following the given *Vector Field* for the given distance starting from *ego* (or the position optionally provided with `from` *`vector`*). Specifies `parentOrientation` to be the orientation of the vector field at the resulting point.

Note: This specifier uses a forward Euler approximation of the continuous vector field. The choice of step size can be customized for individual fields: see the documentation of *Vector Field*. If necessary, you can also call the underlying method *`VectorField.followFrom`* directly.

Orientation Specifiers

facing *orientation*

Specifies:

- `yaw` with priority 1
- `pitch` with priority 1
- `roll` with priority 1

Dependencies: `parentOrientation`

Sets the object's `yaw`, `pitch`, and `roll` so that its orientation in global coordinates is equal to the given orientation. If a single scalar is given, it is interpreted as a *Heading*: so for example `facing 45 deg` orients the object in the XY plane, facing northwest. If a triple of scalars is given, it is interpreted as a triple of global Euler angles: so for example `facing (45 deg, 90 deg, 0)` would orient the object to face northwest as above but then apply a 90° pitch upwards.

facing *vectorField*

Specifies:

- `yaw` with priority 1
- `pitch` with priority 1
- `roll` with priority 1

Dependencies: `position` • `parentOrientation`

Sets the object's `yaw`, `pitch`, and `roll` so that its orientation in global coordinates is equal to the orientation provided by the given *Vector Field* at the object's `position`.

facing (toward | away from) *vector*

Specifies:

- `yaw` with priority 1

Dependencies: `position` • `parentOrientation`

Sets the object's `yaw` so that it faces toward/away from the given position (thereby depending on the object's `position`).

facing directly (toward | away from) *vector*

Specifies:

- `yaw` with priority 1
- `pitch` with priority 1

Dependencies: `position` • `parentOrientation`

Sets the object's `yaw` *and* `pitch` so that it faces directly toward/away from the given position (thereby depending on the object's `position`).

apparently facing *heading* [from *vector*]

Specifies:

- `yaw` with priority 1

Dependencies: `position` • `parentOrientation`

Sets the `yaw` of the object so that it has the given heading with respect to the line of sight from *ego* (or the `from` vector). For example, if the *ego* is in the XY plane, then `apparently facing 90 deg` orients the new object so that the ego's camera views its left side head-on.

Specifier Resolution

Specifier resolution is the process of determining, given the set of specifiers used to define an object, which properties each specifier should determine and what order to evaluate the specifiers in. As each specifier can specify multiple properties with various priorities, and can depend on the results of other specifiers, this process is somewhat non-trivial. Assuming there are no cyclic dependencies or conflicts, the process will conclude with each property being determined by its unique highest-priority specifier if one exists (possibly modified by a modifying specifier), and otherwise by its default value, with default values from subclasses overriding those in superclasses.

The full procedure, given a set of specifiers S used to define an instance of class C, works as follows:

1. If a property is specified at the same priority level by multiple specifiers in S, an ambiguity error is raised.
2. The set of properties P for the new object is found by combining the properties specified by all members of S with the properties inherited from the class C.
3. Default value specifiers from C (or if not overridden, from its superclasses) are added to S as needed so that each property in P is paired with a unique non-modifying specifier in S specifying it (taking the highest-priority specifier, if there are multiple), plus up to one modifying specifier modifying it.
4. The dependency graph of the specifiers S is constructed (with edges from each specifier to the others which depend on its results). If it is cyclic, an error is raised.
5. The graph is topologically sorted and the specifiers are evaluated in this order to determine the values of all properties P of the new object.

1.8.8 Operators Reference

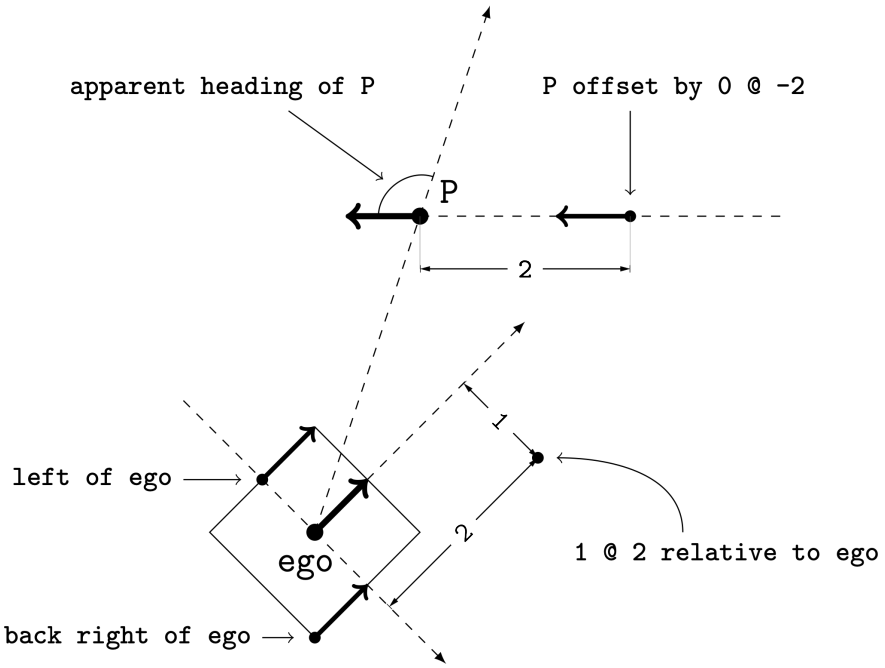


Fig. 7: Illustration of several operators. Each *OrientedPoint* (e.g. P) is shown as a bold arrow.

Scalar Operators

relative heading of *heading* [from *heading*]

The relative heading of the given heading with respect to ego (or the heading provided with the optional from heading)

apparent heading of *OrientedPoint* [from *vector*]

The apparent heading of the OrientedPoint, with respect to the line of sight from ego (or the position provided with the optional from vector)

distance [from *vector*] to *vector*

The distance to the given position from ego (or the position provided with the optional from vector)

angle [from *vector*] to *vector*

The heading (azimuth) to the given position from ego (or the position provided with the optional from vector). For example, if angle to taxi is zero, then taxi is due North of ego

altitude [from *vector*] to *vector*

The altitude to the given position from ego (or the position provided with the optional from vector). For example, if altitude to plane is , then plane is directly above ego.

Boolean Operators

(*Point* | *OrientedPoint*) can see (*vector* | *Object*)

Whether or not a position or *Object* is visible from a *Point* or *OrientedPoint*, accounting for occlusion.

See the *Visibility System* reference for a discussion of the visibility model.

(*vector* | *Object*) in *region*

Whether a position or *Object* lies in the *Region*; for the latter, the object must be completely contained in the region.

Orientation Operators

scalar deg

The given angle, interpreted as being in degrees. For example 90 deg evaluates to /2

vectorField at *vector*

The orientation specified by the vector field at the given position

(*heading* | *vectorField*) relative to (*heading* | *vectorField*)

The first heading/vector field, interpreted as an offset relative to the second heading/vector field. For example, `-5 deg relative to 90 deg` is simply 85 degrees. If either direction is a vector field, then this operator yields an expression depending on the `position` property of the object being specified.

Vector Operators

vector (relative to | offset by) *vector*

The first vector, interpreted as an offset relative to the second vector (or vice versa). For example, `(5, 5, 5) relative to (100, 200, 300)` is `(105, 205, 305)`. Note that this polymorphic operator has a specialized version for instances of *OrientedPoint*, defined *below*: so for example `(-3, 0, 0) relative to taxi` will not use the version of this operator for vectors (even though the *Object* taxi can be coerced to a vector).

vector offset along *direction* by *vector*

The second vector, interpreted in a local coordinate system centered at the first vector and oriented along the given direction (which, if a vector field, is evaluated at the first vector to obtain an orientation)

Region Operators

visible *region*

The part of the given region which is visible from the ego object (i.e. the intersection of the given region with the visible region of the ego).

not visible *region*

The part of the given region which is *not* visible from the ego object (as above, based on the ego's visible region).

region visible from (*Point* | *OrientedPoint*)

The part of the given region visible from the given *Point* or *OrientedPoint* (like `visible region` but from an arbitrary *Point*/*OrientedPoint*).

region not visible from (*Point* | *OrientedPoint*)

The part of the given region not visible from the given *Point* or *OrientedPoint* (like `not visible region` but from an arbitrary *Point*/*OrientedPoint*).

OrientedPoint Operators

vector relative to *OrientedPoint*

The given vector, interpreted in the local coordinate system of the OrientedPoint. So for example `(1, 2, 0) relative to ego` is 1 meter to the right and 2 meters ahead of ego.

OrientedPoint offset by *vector*

Equivalent to `vector relative to OrientedPoint` above

(front | back | left | right | top | bottom) of *Object*

The midpoint of the corresponding side of the bounding box of the *Object*, inheriting the Object's orientation.

(front | back) (left | right) of *Object*

The midpoint of the corresponding edge of the Object's bounding box, inheriting the Object's orientation.

(top | bottom) (front | back) (left | right) of *Object*

The corresponding corner of the Object's bounding box, inheriting the Object's orientation.

Temporal Operators

Temporal operators can be used inside `require` statements to constrain how a dynamic scenario evolves over time. The semantics of these operators are taken from Linear Temporal Logic (specifically, we use RV-LTL [B10] to properly model the finite length of Scenic simulations).

always *condition*

Require the given condition to hold throughout the execution of the dynamic scenario.

eventually *condition*

Require the given condition to hold at some point during the execution of the dynamic scenario.

next *condition*

Require the given condition to hold at the next time step of the dynamic scenario.

For example, while `require X` requires that `X` hold at time step 0 (the start of the simulation), `require next X` requires that `X` hold at time step 1. The requirement `require always (X implies next X)` says that for every time step N, if `X` is true at that time step then it is also true at step $N+1$; equivalently, if `X` ever becomes true, it must remain true for the rest of the simulation.

condition until *condition*

Require the second condition to hold at some point, and the first condition to hold at every time step before then (after which it is unconstrained).

Note that this is the so-called *strong until*, since it requires the second condition to eventually become true. For the *weak until*, which allows the second condition to never hold (in which case the first condition must *always* hold), you can write `require (X until Y) or (always X and not Y)`.

hypothesis implies *conclusion*

Require the conclusion to hold if the hypothesis holds.

This is syntactic sugar for `not hypothesis or conclusion`. It is mainly useful in making requirements that constrain multiple time steps easier to read: for example, `require always X implies Y` requires that at every time step when `X` holds, `Y` must also hold.

References

1.8.9 Built-in Functions Reference

These functions are built into Scenic and may be used without needing to import any modules.

Miscellaneous Python Functions

The following functions work in the same way as their Python counterparts except that they accept random values:

- `sin`, `cos`, `hypot` (from the Python `math` module)
- `len`, `max`, `min`, `round`
- `float`, `int`, `str`

The other Python built-in functions (e.g. `enumerate`, `range`, `open`) are available but do not accept random arguments.

Note: If in the definition of a scene you would like to pass random values into some other function from the Python standard library (or any other Python package), you will need to wrap the function with the *`distributionFunction`* decorator. This is not necessary when calling external functions inside requirements or dynamic behaviors.

---

filter

The `filter` function works as in Python except it is now able to operate over random lists. This feature can be used to work around Scenic's lack of support for randomized control flow in certain cases. In particular, Scenic does not allow iterating over a random list, but it is still possible to select a random element satisfying a desired criterion using `filter`:

```
mylist = Uniform([-1, 1, 2], [-3, 4])     # pick one of these lists 50/50
filtered = filter(lambda e: e > 0, y)     # extract only the positive elements
x = Uniform(*filtered)                    # pick one of them at random
```

In the last line, we use Python's unpacking operator * to use the elements of the chosen list which pass the filter as arguments to *Uniform*; thus `x` is sampled as a uniformly-random choice among such elements.[1]

For an example of this idiom in a realistic scenario, see `examples/driving/OAS_scenarios/oas_scenario_28.scenic`.

resample

The *resample* function takes a distribution and samples a new value from it, conditioned on the values of its parameters, if any. This is useful in cases where you have a complicated distribution that you want multiple samples from.

For example, in the program

```
x = Uniform(0, 5)
y = Range(x, x+1)
z = resample(y)
```

with probability 1/2 both `y` and `z` are independent uniform samples from the interval $(0, 1)$, and with probability 1/2 they are independent uniform samples from $(5, 6)$. It is never the case that $y \in (0, 1)$ and $z \in (5, 6)$ or vice versa, which would require inconsistent assignments to `x`.

Note: This function can only be applied to the basic built-in distributions (see the *Distributions Reference*). Resampling a more complex expression like `x + y` where `x` and `y` are distributions would be ambiguous (what if `x` and `y` are used elsewhere?) and so is not allowed.

---

localPath

The *localPath* function takes a relative path with respect to the directory containing the `.scenic` file where it is used, and converts it to an absolute path. Note that the path is returned as a `pathlib.Path` object.

[1] If there are no such elements, i.e., the filtered list is empty, then Scenic will reject the scenario and try sampling again.

verbosePrint

The *verbosePrint* function operates like print except that it you can specify at what verbosity level (see *--verbosity*) it should actually print. If no level is specified, it prints at all levels except verbosity 0.

Scenic libraries intended for general use should use this function instead of print so that all non-error messages from Scenic can be silenced by setting verbosity 0.

simulation

The *simulation* function, available for use in dynamic behaviors and scenarios, returns the currently-running *Simulation*. This allows access to global information about the simulation, e.g. *simulation*().currentTime to find the current time step; however, it is provided primarily so that scenarios written for a specific simulator may use simulator-specific functionality (by calling custom methods provided by that simulator's subclass of *Simulation*).

1.8.10 Visibility System

The Scenic visibility system is composed of two main parts: *visible regions* and *visibility checks*, which are described in detail below. An object is defined to be visible (modulo occlusion) if it lies within the horizontal and vertical viewAngles of the object and is within it's visibleDistance, i.e. if it lies in the *visible region* of the object. This is not how Scenic actually checks visibility though, instead relying on *visibility checks* which internally use ray tracing and can account for occlusion.

Visible Regions

All Scenic objects define a *visible region*, a *Region* that is "visible" from a given *Object*. This region is defined by two groups of properties: spatial ones like position and orientation, and visibility specific ones:

- viewAngles : The horizontal and vertical angles (in radians) of the object's field of view. The horizontal view angle must be between 0 and 2 and the vertical view angle must be between 0 and .
- visibleDistance: Distance used to determine the visible range of the object.
- cameraOffset: Position of the camera relative to the object's position.

While visible regions do in fact define what an object can see, Scenic does not directly use them to determine if something is visible from an object: instead they serve an accessory role (e.g. making sampling more efficient). The visible region of a *Point* is a sphere, while that of an *OrientedPoint* or *Object* can be a variety of shapes (see *ViewRegion* for details). An object's visible region is used by various specifiers and operators, such as the *visible {region}* operator, the *visible* specifier, etc. Note that an object's visible region is represented by a mesh and so is not exact, and that while Scenic takes occlusion by other objects into account when testing visibility, the visible region itself ignores occlusion.

Visibility Checks

It is often useful to determine whether something is actually visible from another object, i.e. a visibility check. Scenic performs such checks using ray tracing, allowing it to account for other objects occluding visibility. Something is considered visible if any ray (within viewAngles) collides with it (within visibleDistance), without colliding with an occluding object first. Since Scenic sends a finite number of rays, it is possible for false negatives to occur, though this can be tuned using the properties below. Visibility checks are used by various specifiers and operators, such as the *can see* operator, the *visible* specifier, etc.

Various object properties directly affect how Scenic performs visibility checks (including those listed above for visible regions):

- `viewRayDensity`: By default determines the number of rays used during visibility checks. This value is the density of rays per degree of visible range in one dimension. The total number of rays sent will be this value squared per square degree of this object's view angles. This value determines the default value for `viewRayCount`, so if `viewRayCount` is overwritten this value is ignored.
- `viewRayCount`: The total number of horizontal and vertical view angles to be sent, or None if this value should be computed automatically.
- `viewRayDistanceScaling`: Whether or not the number of rays should scale with the distance to the object. Ignored if `viewRayCount` is passed.
- `occluding`: Whether or not this object occludes visibility.

Scenic uses several internal heuristics to speed up visibility checks, such as only sending rays where an object might actually be visible. Even with these heuristics, certain types of checks, such as those where an object is fully occluded but would otherwise be visible, can be very expensive. We recommend tuning `viewRayDensity` if runtimes are problematic, though note this may increase the risk of false negatives. Setting `viewRayDistanceScaling` to `True` can also help, especially in situations where objects can be very far away or very close, but one wishes to avoid setting `viewRayDensity` to a higher value. If one is seeking to emulate a specific camera resolution, one might instead wish to directly set `viewRayCount` (e.g. setting it to (1920, 1080) to emulate a full HD camera).

Semantics and Scenario Generation

The pages above describe the semantics of each of Scenic's constructs individually; the following pages cover the semantics of entire Scenic programs, and how scenes and simulations are generated from them.

1.8.11 Scene Generation

The "output" of a Scenic program has two parts: a *scene* describing a configuration of physical objects, and a *policy* defining how those objects behave over time. The latter is relevant only for running dynamic simulations from a Scenic program, and is discussed in our page on *Execution of Dynamic Scenarios*. In this page, we describe how scenes are generated from a Scenic program.

In Scenic, a scene consists of the following data:

- a set of *objects* present in the scene (one of which *may* be designated the `ego` object);
- concrete values for all of the properties of these objects, such as `position`, `heading`, etc.;
- concrete values for each global parameter.

A Scenic program defines a probability distribution over such scenes in the usual way for imperative probabilistic programming languages with constraints (often called *observations*). Running the program ignoring any `require` statements and making random choices whenever a distribution is evaluated yields a distribution over possible executions of the program and therefore over generated scenes. Then any executions which violate a `require` condition are discarded, normalizing the probabilities of the remaining executions.

The Scenic tool samples from this distribution using rejection sampling: repeatedly sampling scenes until one is found which satisfies the requirements. This approach has the advantage of allowing arbitrarily-complex requirements and sampling from the exact distribution we want. However, if the requirements have a low probability of being satisfied, it may take many iterations to find a valid scene: in the worst case, if the requirements cannot be satisfied, rejection sampling will run forever (although the `Scenario.generate` function imposes a finite limit on the number of iterations by default). To reduce the number of iterations required in some common cases, Scenic applies several "pruning" techniques to exclude parts of the scene space which violate the requirements ahead of time (this is done during compilation; see *our paper* for details). The scene generation procedure then works as follows:

1. Decide which user-defined requirements will be enforced for this sample (*soft requirements* have only some probability of being required).

2. Invoke the external sampler to sample any external parameters.
3. Sample values for all distributions defined in the scene (all expressions which have random values, represented internally as `Distribution` objects).
4. Check if the sampled values satisfy the built-in and user-defined requirements: if not, reject the sample and repeat from step (2).

1.8.12 Execution of Dynamic Scenarios

As described in our tutorial on *Dynamic Scenarios*, Scenic scenarios can specify the behavior of agents over time, defining a *policy* which chooses actions for each agent at each time step. Having sampled an initial scene from a Scenic program (see *Scene Generation*), we can run a dynamic simulation by setting up the scene in a simulator and running the policy in parallel to control the agents. The API for running dynamic simulations is described in *Using Scenic Programmatically* (mainly the `Simulator.simulate` method); this page details how Scenic executes such simulations.

The policy for each agent is given by its dynamic behavior, which is a coroutine that usually executes like an ordinary function, but is suspended when it takes an action (using `take` or `wait`) and resumed after the simulation has advanced by one time step. As a result, behaviors effectively run in parallel with the simulation. Behaviors are also suspended when they invoke a sub-behavior using `do`, and are not resumed until the sub-behavior terminates.

When a behavior is first invoked, its preconditions are checked, and if any are not satisfied, the simulation is rejected, requiring a new simulation to be sampled.[1] The behavior's invariants are handled similarly, except that they are also checked whenever the behavior is resumed (i.e. after taking an action and after a sub-behavior terminates).

Monitors and `compose` blocks of modular scenarios execute in the same way as behaviors, with `compose` blocks also including additional checks to see if any of their `terminate when` conditions have been met or their temporal requirements violated.

In detail, a single time step of a dynamic simulation is executed according to the following procedure:

1. Execute all currently-running modular scenarios for one time step. Specifically, for each such scenario:
    a. Check if any of its temporal requirements have already been violated[2]; if so, reject the simulation.
    b. Check if the scenario's time limit (if `terminate after` has been used) has been reached; if so, go to step (e) below to stop the scenario.
    c. If the scenario is not currently running a sub-scenario (with `do`), check its invariants; if any are violated, reject the simulation.[Page 89, 1]
    d. If the scenario has a `compose` block, run it for one time step (i.e. resume it until it or a subscenario it is currently running using `do` executes `wait`). If the block executes a `require` statement with a false condition, reject the simulation. If it executes `terminate` or `terminate simulation`, or finishes executing, go to step (e) below to stop the scenario.
    e. If the scenario is stopping for one of the reasons above, first recursively stop any sub-scenarios it is running, then revert the effects of any `override` statements it executed. Next, check if any of its temporal requirements were not satisfied: if so, reject the simulation. Otherwise, the scenario returns to its parent scenario if it was invoked using `do`; if it was the top-level scenario, or if it executed `terminate simulation`, we set a flag indicating the top-level scenario has terminated. (We do not terminate immediately since we still need to check monitors in the next step.)

[1] By default, violations of preconditions and invariants cause the simulation to be rejected; however, `Simulator.simulate` has an option to treat them as fatal errors instead.

[2] More precisely, whether it is impossible for the requirement to be satisfied no matter how the simulation continues. For example, given the requirement `require always X`, if `X` is false in the current time step then the whole simulation will certainly violate the requirement and we can reject. On the other hand, given the requirement `require eventually X`, the fact that `X` is currently false does not mean the requirement will necessarily be violated, since `X` could become true later. For such requirements Scenic will not reject until the simulation has completed, at which point we can tell with certainty whether or not the requirement was satisfied.

2. Save the values of all *record* statements, as well as *record initial* statements if it is time step 0.
3. Run each monitor instantiated in the currently-running scenarios for one time step (i.e. resume it until it executes *wait*). If it executes a *require* statement with a false condition, reject the simulation. If it executes *terminate*, stop the scenario which instantiated it as in step (1e) above. If it executes *terminate simulation*, set the termination flag (and continue running any other monitors).
4. If the termination flag is set, any of the *terminate simulation when* conditions are satisfied, or a time limit passed to *Simulator.simulate* has been reached, go to step (10) to terminate the simulation.
5. Execute the dynamic behavior of each agent to select its action(s) for the time step. Specifically, for each agent's behavior:
   a. If the behavior is not currently running a sub-behavior (with *do*), check its invariants; if any are violated, reject the simulation.[1]
   b. Resume the behavior until it (or a subbehavior it is currently running using *do*) executes *take* or *wait*. If the behavior executes a *require* statement with a false condition, reject the simulation. If it executes *terminate*, stop the scenario which defined the agent as in step (1e) above. If it executes *terminate simulation*, go to step (10) to terminate the simulation. Otherwise, save the (possibly empty) set of actions specified for the agent to take.
6. For each agent, execute the actions (if any) its behavior chose in the previous step.
7. Run the simulator for one time step.
8. Increment the simulation clock (the `currentTime` attribute of *Simulation*).
9. Update every dynamic property of every object to its current value in the simulator.
10. If the simulation is stopping for one of the reasons above, first check if any of the temporal requirements of any remaining scenarios were not satisfied: if so, reject the simulation. Otherwise, save the values of any *record final* statements.

1.9 Command-Line Options

The **scenic** command supports a variety of options. Run **scenic -h** for a full list with short descriptions; we elaborate on some of the most important options below.

Options may be given before and after the path to the Scenic file to run, so the syntax of the command is:

```
$ scenic [options] FILE [options]
```

1.9.1 General Scenario Control

-m <model>, **--model** <model>

Specify the world model to use for the scenario, overriding any *model* statement in the scenario. The argument must be the fully qualified name of a Scenic module found on your `PYTHONPATH` (it does not necessarily need to be built into Scenic). This allows scenarios written using a generic model, like that provided by the *Driving Domain*, to be executed in a particular simulator (see the *dynamic scenarios tutorial* for examples).

The equivalent of this option for the Python API is the `model` argument to *scenic.scenarioFromFile*.

-p <param> <value>, **--param** <param> <value>

Specify the value of a global parameter. This assignment overrides any *param* statements in the scenario. If the given value can be interpreted as an `int` or `float`, it is; otherwise it is kept as a string.

The equivalent of this option for the Python API is the `params` argument to *scenic.scenarioFromFile* (which, however, does not attempt to convert strings to numbers).

-s <seed>, **--seed** <seed>

Specify the random seed used by Scenic, to make sampling deterministic.

This option sets the seed for the Python random number generator `random` and the `numpy` random number generator `numpy.random`, so external Python code called from within Scenic can also be made deterministic (although `random` and `numpy.random` should not be used in place of Scenic's own sampling constructs in Scenic code).

--scenario <scenario>

If the given Scenic file defines multiple scenarios, select which one to run. The named modular scenario must not require any arguments.

The equivalent of this option for the Python API is the `scenario` argument to *scenic.scenarioFromFile*.

--2d

Compile the scenario in 2D compatibility mode.

The equivalent of this option for the Python API is the `mode2D` argument to *scenic.scenarioFromFile*.

1.9.2 Dynamic Simulations

-S, --simulate

Run dynamic simulations from scenes instead of plotting scene diagrams. This option will only work for scenarios which specify a simulator, which is done automatically by the world models for the simulator interfaces that support dynamic scenarios, e.g. *scenic.simulators.carla.model* and *scenic.simulators.lgsvl.model*. If your scenario is written for an abstract domain, like *scenic.domains.driving*, you will need to use the *--model* option to specify the specific model for the simulator you want to use.

--time <steps>

Maximum number of time steps to run each simulation (the default is infinity). Simulations may end earlier if termination criteria defined in the scenario are met (see *terminate when* and *terminate*).

--count <number>

Number of successful simulations to run (i.e., not counting rejected simulations). The default is to run forever.

1.9.3 Debugging

--version

Show which version of Scenic is being used.

-v <verbosity>, **--verbosity** <verbosity>

Set the verbosity level, from 0 to 3 (default 1):

0

Nothing is printed except error messages and warnings (to `stderr`). Warnings can be suppressed using the `PYTHONWARNINGS` environment variable.

1

The main steps of compilation and scene generation are indicated, with timing statistics.

2

Additionally, details on which modules are being compiled and the reasons for any scene/simulation rejections are printed.

3
Additionally, the actions taken by each agent at each time step of a dynamic simulation are printed.

This option can be configured from the Python API using *scenic.setDebuggingOptions*.

--show-params
Show values of global parameters for each generated scene.

--show-records
Show recorded values (see *record*) for each dynamic simulation.

-b, --full-backtrace
Include Scenic's internals in backtraces printed for uncaught exceptions. This information will probably only be useful if you are developing Scenic.

This option can be enabled from the Python API using *scenic.setDebuggingOptions*.

--pdb
If an error occurs, enter the Python interactive debugger pdb. Implies the *-b* option.

This option can be enabled from the Python API using *scenic.setDebuggingOptions*.

--pdb-on-reject
If a scene/simulation is rejected (so that another must be sampled), enter pdb. Implies the *-b* option.

This option can be enabled from the Python API using *scenic.setDebuggingOptions*.

1.10 Using Scenic Programmatically

While Scenic is most easily invoked as a command-line tool, it also provides a Python API for compiling Scenic programs, sampling scenes from them, and running dynamic simulations.

1.10.1 Compiling Scenarios and Generating Scenes

The top-level interface to Scenic is provided by two functions in the `scenic` module which compile a Scenic program:

scenarioFromFile(*path*, *params={}*, *model=None*, *scenario=None*, ***, *mode2D=False*, ***kwargs*)
Compile a Scenic file into a *Scenario*.

Parameters

- **path** (*str*) – Path to a Scenic file.
- **params** (*dict*) – Global parameters to override, as a dictionary mapping parameter names to their desired values.
- **model** (*str*) – Scenic module to use as world model.
- **scenario** (*str*) – If there are multiple modular scenarios in the file, which one to compile; if not specified, a scenario called 'Main' is used if it exists.
- **mode2D** (*bool*) – Whether to compile this scenario in `2D compatibility mode`.

Returns
A *Scenario* object representing the Scenic scenario.

Note for Scenic developers: this function accepts additional keyword arguments which are intended for internal use and debugging only. See *_scenarioFromStream* for details.

scenarioFromString(*string*, *params={}*, *model=None*, *scenario=None*, ***, *filename='<string>'*, *mode2D=False*, ***kwargs*)

Compile a string of Scenic code into a *Scenario*.

The optional **filename** is used for error messages. Other arguments are as in *scenarioFromFile*.

The resulting *Scenario* object represents the abstract scenario defined by the Scenic program. To sample concrete scenes from this object, you can call the *Scenario.generate* method, which returns a *Scene*. If you are only using static scenarios, you can extract the sampled values for all the global parameters and objects in the scene from the *Scene* object. For example:

```
import random, scenic
random.seed(12345)
scenario = scenic.scenarioFromString('ego = new Object with foo Range(0, 5)')
scene, numIterations = scenario.generate()
print(f'ego has foo = {scene.egoObject.foo}')
```

```
ego has foo = 2.083099362726706
```

1.10.2 Running Dynamic Simulations

To run dynamic scenarios, you must instantiate an instance of the *Simulator* class for the particular simulator you want to use. Each simulator interface that supports dynamic simulations defines a subclass of *Simulator*; for example, *NewtonianSimulator* for the simple Newtonian simulator built into Scenic. These subclasses provide simulator-specific functionality, and have different requirements for their use: see the specific documentation of each interface under *scenic.simulators* for details.

Once you have an instance of *Simulator*, you can ask it to run a simulation from a *Scene* by calling the *Simulator.simulate* method. If Scenic is able to run a simulation that satisfies all the requirements in the Scenic program (potentially after multiple attempts – Scenic uses rejection sampling), this method will return a *Simulation* object. Results of the simulation can then be obtained by inspecting its `result` attribute, which is an instance of *SimulationResult* (simulator-specific subclasses of *Simulation* may also provide additional information). For example:

```
import scenic
from scenic.simulators.newtonian import NewtonianSimulator
scenario = scenic.scenarioFromFile('examples/driving/badlyParkedCarPullingIn.scenic',
                                   model='scenic.simulators.newtonian.driving_model',
                                   mode2D=True)
scene, _ = scenario.generate()
simulator = NewtonianSimulator()
simulation = simulator.simulate(scene, maxSteps=10)
if simulation:  # `simulate` can return None if simulation fails
        result = simulation.result
        for i, state in enumerate(result.trajectory):
                egoPos, parkedCarPos = state
                print(f'Time step {i}: ego at {egoPos}; parked car at {parkedCarPos}')
```

If you want to monitor data from simulations to see if the system you are testing violates its specfications, you may want to use VerifAI instead of implementing your own code along the lines above. VerifAI supports running tests from Scenic programs, specifying system specifications using temporal logic or arbitrary Python monitor functions, actively searching the space of parameters in a Scenic program to find concrete scenarios where the system violates its specs[1], and more. See the VerifAI documentation for details.

[1] VerifAI's active samplers can be used directly from Scenic when VerifAI is installed. See *scenic.core.external_params*.

1.10.3 Storing Scenes/Simulations for Later Use

Scene and *Simulation* objects are heavyweight and not themselves suitable for bulk storage or transmission over a network[2]. However, Scenic provides serialization routines which can encode such objects into relatively short sequences of bytes. Compact encodings are achieved by storing only the sampled values of the primitive random variables in the scenario: all non-random information is obtained from the original Scenic file.

Having compiled a Scenic scenario into a *Scenario* object, any scenes you generate from the scenario can be encoded as bytes using the *Scenario.sceneToBytes* method. For example, to save a scene to a file one could use code like the following:

```
import scenic, tempfile, pathlib
scenario = scenic.scenarioFromFile('examples/gta/parkedCar.scenic', mode2D=True)
scene, _ = scenario.generate()
data = scenario.sceneToBytes(scene)
with open(pathlib.Path(tempfile.gettempdir()) / 'test.scene', 'wb') as f:
        f.write(data)
print(f'ego car position = {scene.egoObject.position}')
```

Then you could restore the scene in another process, obtaining the same position for the ego car:

```
import scenic, tempfile, pathlib
scenario = scenic.scenarioFromFile('examples/gta/parkedCar.scenic', mode2D=True)
with open(pathlib.Path(tempfile.gettempdir()) / 'test.scene', 'rb') as f:
        data = f.read()
scene = scenario.sceneFromBytes(data)
print(f'ego car position = {scene.egoObject.position}')
```

Notice how we need to compile the scenario a second time in order to decode the scene, if the original *Scenario* object is not available. If you need to send a large number of scenes from one computer to another, for example, it suffices to send the Scenic file for the underlying scenario, plus the encodings of each of the scenes.

You can encode and decode simulations run from a *Scenario* in a similar way, using the *Scenario.simulationToBytes* and *Scenario.simulationFromBytes* methods. One additional concern when replaying a serialized simulation is that if your simulator is not deterministic (or you change the simulator configuration), the original simulation and its replay can diverge, leading to unexpected behavior or exceptions. Scenic can attempt to detect such divergences by saving the exact history of the simulation and comparing it to the replay, but this greatly increases the size of the encoded simulation. See *Simulator.simulate* for the available options.

Note: The serialization format used for scenes and simulations is suitable for long-term storage (for instance if you want to save all the simulations you've run so that you can return to one later for further analysis), but it is not guaranteed to be compatible across major versions of Scenic.

See also:

If you get exceptions or unexpected behavior when using the API, Scenic provides various debugging features: see *Debugging*.

[2] If you really do need to store/transmit such objects, you may be able to do so using dill, a drop-in replacement for Python's standard `pickle` library. Be aware that pickling will produce much larger encodings than Scenic's own APIs, as they need to include all the information present in the original Scenic file and its associated resources (e.g. for driving scenarios, the entire road map). Unpickling malicious files can also trigger arbitrary code execution, while Scenic's deserialization APIs can be used with untrusted data (as long as you trust the Scenic program you're running, of course).

1.11 Developing Scenic

This page covers information useful if you will be developing Scenic, either changing the language itself or adding new built-in libraries or simulator interfaces.

To find documentation (and code) for specific parts of Scenic's implementation, see our page on *Scenic Internals*.

1.11.1 Getting Started

Start by cloning our repository on GitHub and setting up your virtual environment. Then to install Scenic and its development dependencies in your virtual environment run:

```
$ python -m pip install -e ".[dev]"
```

This will perform an "editable" install, so that any changes you make to Scenic's code will take effect immediately when running Scenic in your virtual environment.

Scenic uses the isort and black tools to automatically sort `import` statements and enforce a consistent code style. Run the command **`pre-commit install`** to set up hooks which will run every time you commit and correct any formatting problems (you can then inspect the files and try committing again). You can also manually run the formatters on the files changed since the last commit with **`pre-commit run`**.[1]

1.11.2 Running the Test Suite

Scenic has an extensive test suite exercising most of the features of the language. We use the pytest Python testing tool. To run the entire test suite, run the command **`pytest`** inside the virtual environment from the root directory of the repository.

Some of the tests are quite slow, e.g. those which test the parsing and construction of road networks. We add a `--fast` option to pytest which skips such tests, while still covering all of the core features of the language. So it is convenient to often run **`pytest --fast`** as a quick check, remembering to run the full **`pytest`** before making any final commits. You can also run specific parts of the test suite with a command like **`pytest tests/syntax/test_specifiers.py`**, or use pytest's `-k` option to filter by test name, e.g. **`pytest -k specifiers`**.

Note that many of Scenic's tests are probabilistic, so in order to reproduce a test failure you may need to set the random seed. We use the pytest-randomly plugin to help with this: at the beginning of each run of `pytest`, it prints out a line like:

```
Using --randomly-seed=344295085
```

Adding this as an option, i.e. running **`pytest --randomly-seed=344295085`**, will reproduce the same sequence of tests with the same Python/Scenic random seed. As a shortcut, you can use **`--randomly-seed=last`** to use the seed from the previous testing run.

If you're running the test suite on a headless server or just want to stop windows from popping up during testing, use the **`--no-graphics`** option to skip graphical tests.

[1] To run the formatters on *all* files, changed or otherwise, use **`make format`** in the root directory of the repository. But this should not be necessary if you installed the pre-commit hooks and so all files already committed are clean.

1.11.3 Debugging

You can use Python's built-in debugger pdb to debug the parsing, compilation, sampling, and simulation of Scenic programs. The Scenic command-line option *-b* will cause the backtraces printed from uncaught exceptions to include Scenic's internals; you can also use the *--pdb* option to automatically enter the debugger on such exceptions. If you're trying to figure out why a scenario is taking many iterations of rejection sampling, first use the *--verbosity* option to print out the reason for each rejection. If the problem doesn't become clear, you can use the *--pdb-on-reject* option to automatically enter the debugger when a scene or simulation is rejected.

If you're using the Python API instead of invoking Scenic from the command line, these debugging features can be enabled using the following function from the `scenic` module:

setDebuggingOptions(*, *verbosity=0*, *fullBacktrace=False*, *debugExceptions=False*, *debugRejections=False*)

Configure Scenic's debugging options.

Parameters

- **verbosity** (*int*) – Verbosity level. Zero by default, although the command-line interface uses 1 by default. See the *--verbosity* option for the allowed values.
- **fullBacktrace** (*bool*) – Whether to include Scenic's innards in backtraces (like the *-b* command-line option).
- **debugExceptions** (*bool*) – Whether to use pdb for post-mortem debugging of uncaught exceptions (like the *--pdb* option).
- **debugRejections** (*bool*) – Whether to enter pdb when a scene or simulation is rejected (like the *--pdb-on-reject* option).

It is possible to put breakpoints into a Scenic program using the Python built-in function `breakpoint`. Note however that since code in a Scenic program is not always executed the way you might expect (e.g. top-level code is only run once, whereas code in requirements can run every time we generate a sample: see *How Scenic is Compiled*), some care is needed when interpreting what you see in the debugger. The same consideration applies when adding `print` statements to a Scenic program. For example, a top-level *print(x)* will not print out the actual value of `x` every time a sample is generated: instead, you will get a single print at compile time, showing the *Distribution* object which represents the distribution of `x` (and which is bound to `x` in the Python namespace used internally for the Scenic module).

1.11.4 Building the Documentation

Scenic's documentation is built using Sphinx. The freestanding documentation pages (like this one) are found under the `docs` folder, written in the reStructuredText format. The detailed documentation of Scenic's internal classes, functions, etc. is largely auto-generated from their docstrings, which are written in a variant of Google's style understood by the `Napoleon` Sphinx extension (see the docstring of *Scenario.generate* for a simple example: click the `[source]` link to the right of the function signature to see the code).

If you modify the documentation, you should build a copy of it locally to make sure everything looks good before you push your changes to GitHub (where they will be picked up automatically by ReadTheDocs). To compile the documentation, enter the `docs` folder and run **`make html`**. The output will be placed in the `docs/_build/html` folder, so the root page will be at `docs/_build/html/index.html`. If your changes do not appear, it's possible that Sphinx has not detected them; you can run **`make clean`** to delete all the files from the last compilation and start from a clean slate.

Scenic extends Sphinx in a number of ways to improve the presentation of Scenic code and add various useful features: see `docs/conf.py` for full details. Some of the most commonly-used features are:

- a `scenic` role which extends the standard Sphinx `samp` role with Scenic syntax highlighting;
- a `sampref` role which makes a cross-reference like `keyword` but allows emphasizing variables like `samp`;

- the `term` role for glossary terms is extended so that the cross-reference will work even if the link is plural but the glossary entry is singular or vice versa.

1.12 Scenic Internals

This section of the documentation describes the implementation of Scenic. Much of this information will probably only be useful for people who need to make some change to the language (e.g. adding a new type of distribution). However, the detailed documentation on Scenic's abstract application domains (in *`scenic.domains`*) and simulator interfaces (in *`scenic.simulators`*) may be of interest to people using those features.

1.12.1 How Scenic is Compiled

The process of compiling a Scenic program into a *`Scenario`* object can be split into several phases. Understanding what each phase does is useful if you plan to modify the Scenic language.

For more details on Phases 1 and 2 (parsing Scenic and converting it into Python), see the *Guide to the Scenic Parser & Compiler*.

Phase 1: Scenic Parser

In this phase the program is parsed using the Scenic parser. The parser is generated from a PEG grammar (`scenic.gram`) using the Pegen parser generator. The parser generates an abstract syntax tree (Scenic AST) for the program. Scenic AST is a superset of Python AST defined in `ast.py` and has additional nodes for representing Scenic-specific constructs.

Phase 2: Scenic Compiler

In this phase, the Scenic AST is transformed into a Python AST. The Scenic Compiler walks the Scenic AST and replaces Scenic-specific nodes with corresponding Python AST nodes.

Phase 3: AST Compilation

Compile the Python AST down to a Python `code` object.

Phase 4: Python Execution

In this phase the Python `code` object compiled in Phase 3 is executed. When run, the definitions of objects, global parameters, requirements, behaviors, etc. produce Python data structures used internally by Scenic to keep track of the distributions, functions, coroutines, etc. used in their definitions. For example, a random value will evaluate to a *`Distribution`* object storing information about which distribution it is drawn from; actually sampling from that distribution will not occur until after the compilation process (when calling *`Scenario.generate`*). A *`require`* statement will likewise produce a closure which can be used at sampling time to check whether its condition is satisfied or not.

Note that since this phase only happens once, at compile time and not sampling time, top-level code in a Scenic program[1] is only executed **once** even when sampling many scenes from it. This is done deliberately, in order to generate a static representation of the semantics of the Scenic program which can be used for sampling without needing to re-run the entire program.

[1] As compared to code inside a `require` statement or a dynamic behavior, which will execute every time a scene is sampled or a simulation is run respectively.

Phase 5: Scenario Construction

In this phase the various pieces of the internal representation of the program resulting from Phase 4 are bundled into a *Scenario* object and returned to the user. This phase is also where the program is analyzed and pruning techniques applied to optimize the scenario for later sampling.

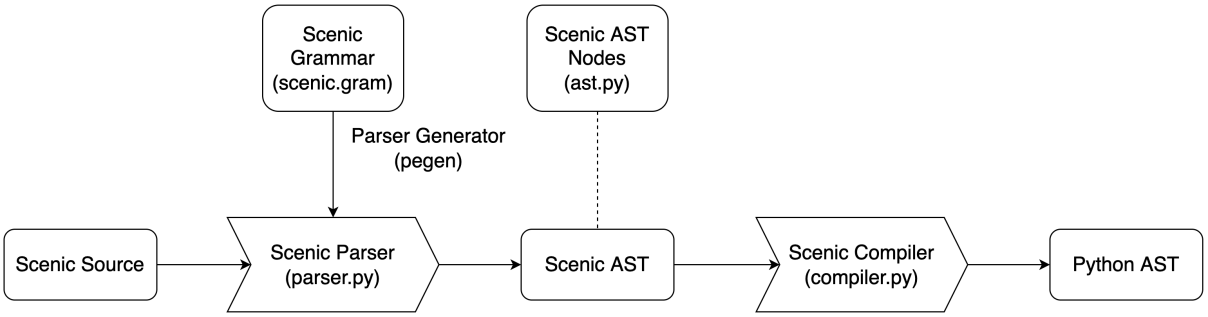
Sampling and Executing Scenarios

Sampling scenes and executing dynamic simulations from them are not part of the compilation process[2]. For documentation on how those are done, see *Scenario.generate* and *scenic.core.simulators* respectively.

1.12.2 Guide to the Scenic Parser & Compiler

This page describes the process of parsing Scenic code and compiling it into equivalent Python. We also include a tutorial illustrating how to add a new syntax construct to Scenic.

Architecture & Terminology



Scenic AST

A Scenic AST is an abstract syntax tree for representing Scenic programs. It is a superset of Python AST and includes nodes for Scenic-specific language constructs.

The *scenic.syntax.ast* module defines all Scenic-specific AST nodes, which are instances of the *AST* class defined in the same file.

AST nodes should include fields to store objects. To add fields, add a parameter to the initializer and define fields by assigning values to `self`.

When adding fields, be sure to update the `_fields` and `__match_args__` fields. `_fields` lists the names of the fields in the AST node and is used by the AST module to traverse the tree, fill in the missing information, etc. `__match_args__` is used by the test suite to assert the structure of the AST node using Python's structural pattern matching.

[2] Although there are some syntax errors which are currently not detected until those stages.

Scenic Grammar

The Scenic Grammar (`syntax/scenic.gram`) is a formal grammar that defines the syntax of the Scenic language. It is written as a Parsing Expression Grammar (PEG) using the Pegen parser generator.

Please refer to Pegen's documentation on how to write a grammar.

Scenic Parser

The Scenic Parser takes Scenic source code and outputs the corresponding abstract syntax tree. It is generated from the grammar file using Pegen.

When you modify `scenic.gram`, you need to regenerate the parser by calling **`make`** or running

```
$ python -m pegen ./src/scenic/syntax/scenic.gram -o ./src/scenic/syntax/parser.py
```

at the project root. When running the test suite with **`pytest`**, the parser is automatically updated before test execution.

`tests/syntax/test_parser.py` includes parser tests and ensures that the parser generates the desired AST.

Scenic Compiler

The Scenic Compiler is a Scenic AST-to-Python AST compiler. The generated Python AST can be passed to the Python interpreter for execution.

Internally, the compiler is a subclass of `ast.NodeTransformer`. It must define visitors for each Scenic AST node which return corresponding Python AST nodes.

Tutorial: Adding New Syntax

In order to add new syntax, you'll want to do the following:

1. add AST nodes to `ast.py`
2. add grammar to `scenic.gram`
3. write parser tests
4. add visitor to `compiler.py`
5. write compiler tests

The rest of this section will demonstrate how we can add the *`implies`* operator using the new parser architecture.

Step 1: Add AST Nodes

First, we define AST nodes that represent the syntax. Since the `implies` operator is a binary operator, the AST node will have two fields for each operand.

```
class ImpliesOp(AST):
    __match_args__ = ("hypothesis", "conclusion")

    def __init__(
        self, hypothesis: ast.AST, conclusion: ast.AST, *args: Any, **kwargs: Any
```

(continues on next page)

(continued from previous page)

```
    ) -> None:
        super().__init__(*args, **kwargs)
        self.hypothesis = hypothesis
        self.conclusion = conclusion
        self._fields = ["hypothesis", "conclusion"]
```

- On line 1, *AST* (*scenic.syntax.ast.AST*, not `ast.AST`) is the base class that all Scenic AST nodes extend.
- On line 2, `__match_args__` is a syntax for using structural pattern matching on argument positions. This is to make it easier to write parser tests.
- On line 5, the initializer takes two required arguments corresponding to the operator's operands (`hypothesis` and `conclusion`). Note that their types are `ast.AST`, which is the base class for *all* AST nodes, including both Scenic AST nodes and Python AST nodes. The additional arguments `*args` and `**kwargs` should be passed to the base class' initializer to store extra information such as line number, offset, etc.
- On line 10, `_fields` is a special field that specifies the child nodes. This is used by the library functions such as `generic_visit` to traverse the syntax tree.

Step 2: Add Grammar

Note: The grammar described here is slightly simplified for the sake of brevity. For the actual grammar used by the parser, see the *Scenic Grammar*.

The next step is to update the `scenic.gram` file with a rule that matches our new construct. We'll add a rule called `scenic_implication`: all Scenic grammar rules should be prefixed with `scenic_` so that we can easily distinguish Scenic-specific rules from those in the original Python grammar.

```
scenic_implication (memo):
    | invalid_scenic_implication  # special rule to explain invalid uses of "implies"
    | a=disjunction "implies" b=disjunction { s.ImpliesOp(a, b, LOCATIONS) }
    | disjunction
```

Our rule has three alternatives, which the parser considers in order. For the moment, let's consider the second alternative, which is the one defining the actual syntax of `implies`: it matches any text matching the `disjunction` rule, followed by the word `implies`, followed by any text matching the `disjunction` rule. In the grammar, precedence and associativity of operators are defined by using separate rules for each precedence level. The `disjunction` rule matches any expression defined using `or` or an operator with higher precedence than `or`. Since implication should bind less tightly than `or`, we use `disjunction` for its operands in our rule. To allow `scenic_implication` to match higher-precedence operators as well as just `implies`, we add the third alternative, which matches any `disjunction`.

Returning to the second alternative, we define its outcome, i.e., the AST node which it generates if it matches, using the ordinary Python code inside the curly brackets. Here `s` refers to the Scenic AST module, so `s.ImpliesOp(a, b, LOCATIONS)` creates an instance of the `ImpliesOp` class we defined above with `a` the `hypothesis` and `b` the `conclusion`. The special term `LOCATIONS` will be replaced with a set of named arguments to express source code locations.

The `implies` operator is unique in that it takes exactly two operands: we disallow `A implies B implies C` as being ambiguous, rather than parsing it as `(A implies B) implies C` (left-associatively) or `A implies (B implies C)` (right-associatively). In order to block the ambiguous case and force the developer to make the meaning clear by wrapping one of the operands in parentheses, our rule says that the right-hand side of the implication must be a `disjunction` rather than an arbitrary expression. This will cause the code `A implies B implies C` to result in a syntax error, because no rules will match.

In order to replace the generic syntax error with a more informative one, we add the `invalid_scenic_implication` rule as the first alternative. Rules with the `invalid_` prefix are special rules for generating custom error messages. Pegen first tries to parse the input *without* using `invalid_` rules. If that fails, it tries parsing again, this time allowing `invalid_` rules: those rules can then generate errors when they match.

```
invalid_scenic_implication[NoReturn]:
    | a=disjunction "implies" disjunction "implies" b=disjunction {
        self.raise_syntax_error_known_range(
            f"`implies` must take exactly two operands", a, b
        )
     }
```

The `invalid_scenic_implication` rule looks for an implication with more than two arguments (e.g. `A implies B implies C`) and raises a syntax error with a detailed error message.

Once we are done with the grammar, run **`make`** to generate the parser from the grammar. If there is no error, the file `src/scenic/syntax/parser.py` will be created.

Step 3: Write Parser Tests

Now that we have the parser, we need to add test cases to check that it works as we expect.

The number of test cases depends on the complexity of the grammar rule. Here, I decided to add the following three cases:

```
class TestOperator: # 1
    def test_implies_basic(self): # 2
        mod = parse_string_helper("x implies y") # 3
        stmt = mod.body[0]
        match stmt:
            case Expr(ImpliesOp(Name("x"), Name("y"))): # 4
                assert True
            case _:
                assert False # 5

    def test_implies_precedence(self):
        mod = parse_string_helper("x implies y or z")
        stmt = mod.body[0]
        match stmt:
            case Expr(ImpliesOp(Name("x"), BoolOp(Or(), [Name("y"), Name("z")]))):
                assert True
            case _:
                assert False

    def test_implies_three_operands(self):
        with pytest.raises(SyntaxError) as e:  # 6
            parse_string_helper("x implies y implies z")
        assert "must take exactly two operands" in e.value.msg
```

1. `TestOperator` is a test class that has all tests related to Scenic operators, so it is natural for us to add test cases here.
2. The test case name should contain the names of the grammar we're testing (`implies` in this case)

3. `parse_string_helper` is a thin wrapper around the parser. The return value would be a module, but we're only concerned about the first statement of the body, so we extract that to the `stmt` variable.
4. We use structural pattern matching to match the result with the expected AST structure. In this case, the statement is expected to be an `Expr` whose value is an `ImpliesOp` that takes `Name`s, `x` and `y`.
5. Be sure to add an otherwise case (with `_`) and assert false. Otherwise, no error will be caught even if the returned node does not match the expected structure.
6. Errors can be tested using `pytest.raises`.

Step 4: Add Visitor to Compiler

The next step is to add a visitor method to the compiler so it knows how to compile the `ImpliesOp` AST node to the corresponding Python AST. In this case, we want to compile `A implies B` to a Python function call `Implies(A, B)`.

The visitor class used in the compiler, `ScenicToPythonTransformer`, is a subclass of `ast.NodeTransformer`, which transforms an AST node of class `C` by calling a method called `visit_C` if one exists, otherwise just recursively transforming its child nodes. So to add the ability to compile `ImpliesOp` nodes, we'll add a method named `visit_ImpliesOp`:

```
class ScenicToPythonTransformer(ast.NodeTransformer):
     def visit_ImpliesOp(self, node: s.ImpliesOp):
        return ast.Call(
            func=ast.Name(id="Implies", ctx=loadCtx),
            args=[self.visit(node.hypothesis), self.visit(node.conclusion)],
            keywords=[],
        )
```

Inside the visitor, we construct a Call to a name `Implies` with `node.hypothesis` and `node.conclusion` as its arguments. Note that the arguments need to be recursively visited using `self.visit`; otherwise Scenic AST nodes inside them won't be compiled.

Step 5: Write Compiler Tests

Similarly to step 3, we add tests for the compiler.

```
def test_implies_op(self):
    node, _ = compileScenicAST(ImpliesOp(Name("x"), Name("y")))
    match node:
        case Call(Name("Implies"), [Name("x"), Name("y")]):
            assert True
        case _:
            assert False
```

`compileScenicAST` is a function that invokes the node transformer. We match the compiled node against the desired structure, which in this case is a call to a function with two arguments.

This completes adding the `implies` operator.

1.12.3 Scenic Grammar

This page gives the formal Parsing Expression Grammar (PEG) used to parse the Scenic language. It is in the format of the Pegen parser generator, and is based on the Python grammar from CPython (see `Grammar/python.gram` in the CPython repository). In the source code, the grammar can be found at `src/scenic/syntax/scenic.gram`.

```
# PEG grammar for Scenic
# Based on the Python grammar at https://github.com/we-like-parsers/pegen/blob/main/data/
↪python.gram

@class ScenicParser

@subheader'''
import enum
import io
import itertools
import os
import sys
import token
from typing import (
    Any, Callable, Iterator, List, Literal, Tuple, TypeVar, Union, NoReturn
)

from pegen.tokenizer import Tokenizer

import scenic.syntax.ast as s
from scenic.core.errors import ScenicParseError

# Singleton ast nodes, created once for efficiency
Load = ast.Load()
Store = ast.Store()
Del = ast.Del()

Node = TypeVar("Node")
FC = TypeVar("FC", ast.FunctionDef, ast.AsyncFunctionDef, ast.ClassDef)

EXPR_NAME_MAPPING = {
    ast.Attribute: "attribute",
    ast.Subscript: "subscript",
    ast.Starred: "starred",
    ast.Name: "name",
    ast.List: "list",
    ast.Tuple: "tuple",
    ast.Lambda: "lambda",
    ast.Call: "function call",
    ast.BoolOp: "expression",
    ast.BinOp: "expression",
    ast.UnaryOp: "expression",
    ast.GeneratorExp: "generator expression",
    ast.Yield: "yield expression",
    ast.YieldFrom: "yield expression",
    ast.Await: "await expression",
    ast.ListComp: "list comprehension",
```

(continues on next page)

(continued from previous page)

```
    ast.SetComp: "set comprehension",
    ast.DictComp: "dict comprehension",
    ast.Dict: "dict literal",
    ast.Set: "set display",
    ast.JoinedStr: "f-string expression",
    ast.FormattedValue: "f-string expression",
    ast.Compare: "comparison",
    ast.IfExp: "conditional expression",
    ast.NamedExpr: "named expression",
}


def parse_file(
    path: str,
    py_version: Optional[tuple]=None,
    token_stream_factory: Optional[
        Callable[[Callable[[], str]], Iterator[tokenize.TokenInfo]]
    ] = None,
    verbose:bool = False,
) -> ast.Module:
    """Parse a file."""
    with open(path) as f:
        tok_stream = (
            token_stream_factory(f.readline)
            if token_stream_factory else
            tokenize.generate_tokens(f.readline)
        )
        tokenizer = Tokenizer(tok_stream, verbose=verbose, path=path)
        parser = ScenicParser(
            tokenizer,
            verbose=verbose,
            filename=os.path.basename(path),
            py_version=py_version
        )
        return parser.parse("file")


def parse_string(
    source: str,
    mode: Union[Literal["eval"], Literal["exec"]],
    py_version: Optional[tuple]=None,
    token_stream_factory: Optional[
        Callable[[Callable[[], str]], Iterator[tokenize.TokenInfo]]
    ] = None,
    verbose: bool = False,
    filename: str = "<unknown>",
) -> Any:
    """Parse a string."""
    tok_stream = (
        token_stream_factory(io.StringIO(source).readline)
        if token_stream_factory else
        tokenize.generate_tokens(io.StringIO(source).readline)
```

(continues on next page)

(continued from previous page)

```
    )
    tokenizer = Tokenizer(tok_stream, verbose=verbose)
    parser = ScenicParser(tokenizer, verbose=verbose, py_version=py_version,␣
↪filename=filename)
    return parser.parse(mode if mode == "eval" else "file")


class Target(enum.Enum):
    FOR_TARGETS = enum.auto()
    STAR_TARGETS = enum.auto()
    DEL_TARGETS = enum.auto()


class Parser(Parser):

    #: Name of the source file, used in error reports
    filename : str

    def __init__(self,
        tokenizer: Tokenizer, *,
        verbose: bool = False,
        filename: str = "<unknown>",
        py_version: Optional[tuple] = None,
    ) -> None:
        super().__init__(tokenizer, verbose=verbose)
        self.filename = filename
        self.py_version = min(py_version, sys.version_info) if py_version else sys.
↪version_info

    def parse(self, rule: str, call_invalid_rules: bool = False) -> Optional[ast.AST]:
        self.call_invalid_rules = call_invalid_rules
        res = getattr(self, rule)()

        if res is None:

            # Grab the last token that was parsed in the first run to avoid
            # polluting a generic error reports with progress made by invalid rules.
            last_token = self._tokenizer.diagnose()

            if not call_invalid_rules:
                self.call_invalid_rules = True

                # Reset the parser cache to be able to restart parsing from the
                # beginning.
                self._reset(0)  # type: ignore
                self._cache.clear()

                res = getattr(self, rule)()

            self.raise_raw_syntax_error("invalid syntax", last_token.start, last_token.
↪end)
```

(continues on next page)

(continued from previous page)

```
        return res

    def check_version(self, min_version: Tuple[int, ...], error_msg: str, node: Node) ->␣
↪Node:
        """Check that the python version is high enough for a rule to apply.

        """
        if self.py_version >= min_version:
            return node
        else:
            raise ScenicParseError(SyntaxError(
                f"{error_msg} is only supported in Python {min_version} and above."
            ))

    def raise_indentation_error(self, msg: str) -> None:
        """Raise an indentation error."""
        last_token = self._tokenizer.diagnose()
        args = (self.filename, last_token.start[0], last_token.start[1] + 1, last_token.
↪line)
        if sys.version_info >= (3, 10):
            args += (last_token.end[0], last_token.end[1] + 1)
        raise ScenicParseError(IndentationError(msg, args))

    def get_expr_name(self, node) -> str:
        """Get a descriptive name for an expression."""
        # See https://github.com/python/cpython/blob/master/Parser/pegen.c#L161
        assert node is not None
        node_t = type(node)
        if node_t is ast.Constant:
            v = node.value
            if v is Ellipsis:
                return "ellipsis"
            elif v is None:
                return str(v)
            # Avoid treating 1 as True through == comparison
            elif v is True:
                return str(v)
            elif v is False:
                return str(v)
            else:
                return "literal"

        try:
            return EXPR_NAME_MAPPING[node_t]
        except KeyError:
            raise ValueError(
                f"unexpected expression in assignment {type(node).__name__} "
                f"(line {node.lineno})."
            )

    def get_invalid_target(self, target: Target, node: Optional[ast.AST]) ->␣
↪Optional[ast.AST]:
```

(continues on next page)

(continued from previous page)

```
        """Get the meaningful invalid target for different assignment type."""
        if node is None:
            return None

        # We only need to visit List and Tuple nodes recursively as those
        # are the only ones that can contain valid names in targets when
        # they are parsed as expressions. Any other kind of expression
        # that is a container (like Sets or Dicts) is directly invalid and
        # we do not need to visit it recursively.
        if isinstance(node, (ast.List, ast.Tuple)):
            for e in node.elts:
                if (inv := self.get_invalid_target(target, e)) is not None:
                    return inv
        elif isinstance(node, ast.Starred):
            if target is Target.DEL_TARGETS:
                return node
            return self.get_invalid_target(target, node.value)
        elif isinstance(node, ast.Compare):
            # This is needed, because the `a in b` in `for a in b` gets parsed
            # as a comparison, and so we need to search the left side of the comparison
            # for invalid targets.
            if target is Target.FOR_TARGETS:
                if isinstance(node.ops[0], ast.In):
                    return self.get_invalid_target(target, node.left)
                return None

            return node
        elif isinstance(node, (ast.Name, ast.Subscript, ast.Attribute)):
            return None
        else:
            return node

    def set_expr_context(self, node, context):
        """Set the context (Load, Store, Del) of an ast node."""
        node.ctx = context
        return node

    def ensure_real(self, number: ast.Constant):
        value = ast.literal_eval(number.string)
        if type(value) is complex:
            self.raise_syntax_error_known_location("real number required in complex␣
↪literal", number)
        return value

    def ensure_imaginary(self, number:  ast.Constant):
        value = ast.literal_eval(number.string)
        if type(value) is not complex:
            self.raise_syntax_error_known_location("imaginary number required in complex␣
↪literal", number)
        return value

    def generate_ast_for_string(self, tokens):
```

(continues on next page)

(continued from previous page)

```
        """Generate AST nodes for strings."""
        err_args = None
        line_offset = tokens[0].start[0]
        line = line_offset
        col_offset = 0
        source = "(\\n"
        for t in tokens:
            n_line = t.start[0] - line
            if n_line:
                col_offset = 0
            source += """\\n""" * n_line + ' ' * (t.start[1] - col_offset) + t.string
            line, col_offset = t.end
        source += "\\n)"
        try:
            m = ast.parse(source)
        except SyntaxError as err:
            args = (err.filename, err.lineno + line_offset - 2, err.offset, err.text)
            if sys.version_info >= (3, 10):
                args += (err.end_lineno + line_offset - 2, err.end_offset)
            err_args = (err.msg, args)
            # Ensure we do not keep the frame alive longer than necessary
            # by explicitely deleting the error once we got what we needed out
            # of it
            del err

        # Avoid getting a triple nesting in the error report that does not
        # bring anything relevant to the traceback.
        if err_args is not None:
            raise ScenicParseError(SyntaxError(*err_args))

        node = m.body[0].value
        # Since we asked Python to parse an alterred source starting at line 2
        # we alter the lineno of the returned AST to recover the right line.
        # If the string start at line 1, tha AST says 2 so we need to decrement by 1
        # hence the -2.
        ast.increment_lineno(node, line_offset - 2)
        return node

    def extract_import_level(self, tokens: List[tokenize.TokenInfo]) -> int:
        """Extract the relative import level from the tokens preceding the module name.

        '.' count for one and '...' for 3.

        """
        level = 0
        for t in tokens:
            if t.string == ".":
                level += 1
            else:
                level += 3
        return level
```

(continues on next page)

(continued from previous page)

```
    def set_decorators(self,
        target: FC,
        decorators: list
    ) -> FC:
        """Set the decorators on a function or class definition."""
        target.decorator_list = decorators
        return target

    def get_comparison_ops(self, pairs):
        return [op for op, _ in pairs]

    def get_comparators(self, pairs):
        return [comp for _, comp in pairs]

    def set_arg_type_comment(self, arg, type_comment):
        if type_comment or sys.version_info < (3, 9):
            arg.type_comment = type_comment
        return arg

    def make_arguments(self,
        pos_only: Optional[List[Tuple[ast.arg, None]]],
        pos_only_with_default: List[Tuple[ast.arg, Any]],
        param_no_default: Optional[List[Tuple[ast.arg, None]]],
        param_default: Optional[List[Tuple[ast.arg, Any]]],
        after_star: Optional[Tuple[Optional[ast.arg], List[Tuple[ast.arg, Any]],␣
↪Optional[ast.arg]]]
    ) -> ast.arguments:
        """Build a function definition arguments."""
        defaults = (
            [d for _, d in pos_only_with_default if d is not None]
            if pos_only_with_default else
            []
        )
        defaults += (
            [d for _, d in param_default if d is not None]
            if param_default else
            []
        )

        pos_only = pos_only or pos_only_with_default

        # Because we need to combine pos only with and without default even
        # the version with no default is a tuple
        pos_only = [p for p, _ in pos_only]
        params = (param_no_default or []) + ([p for p, _ in param_default] if param_
↪default else [])

        # If after_star is None, make a default tuple
        after_star = after_star or (None, [], None)

        return ast.arguments(
            posonlyargs=pos_only,
```

(continues on next page)

(continued from previous page)

```
            args=params,
            defaults=defaults,
            vararg=after_star[0],
            kwonlyargs=[p for p, _ in after_star[1]],
            kw_defaults=[d for _, d in after_star[1]],
            kwarg=after_star[2]
        )

    def _build_syntax_error(
        self,
        message: str,
        start: Optional[Tuple[int, int]] = None,
        end: Optional[Tuple[int, int]] = None
    ) -> None:
        line_from_token = start is None and end is None
        if start is None or end is None:
            tok = self._tokenizer.diagnose()
            start = start or tok.start
            end = end or tok.end

        if line_from_token:
            line = tok.line
        else:
            # End is used only to get the proper text
            line = "\\n".join(
                self._tokenizer.get_lines(list(range(start[0], end[0] + 1)))
            )

        # tokenize.py index column offset from 0 while Cpython index column
        # offset at 1 when reporting SyntaxError, so we need to increment
        # the column offset when reporting the error.
        args = (self.filename, start[0], start[1] + 1, line)
        if sys.version_info >= (3, 10):
            args += (end[0], end[1] + 1)

        return ScenicParseError(SyntaxError(message, args))

    def raise_raw_syntax_error(
        self,
        message: str,
        start: Optional[Tuple[int, int]] = None,
        end: Optional[Tuple[int, int]] = None
    ) -> NoReturn:
        raise self._build_syntax_error(message, start, end)

    def make_syntax_error(self, message: str) -> None:
        return self._build_syntax_error(message)

    def expect_forced(self, res: Any, expectation: str) -> Optional[tokenize.TokenInfo]:
        if res is None:
            last_token = self._tokenizer.diagnose()
            self.raise_raw_syntax_error(
```

(continues on next page)

(continued from previous page)

```
                f"expected {expectation}", last_token.start, last_token.start
            )
        return res

    def raise_syntax_error(self, message: str) -> NoReturn:
        """Raise a syntax error."""
        tok = self._tokenizer.diagnose()
        raise self._build_syntax_error(message, tok.start, tok.end if tok.type != 4 else␣
↪tok.start)

    def raise_syntax_error_known_location(
        self, message: str, node: Union[ast.AST, tokenize.TokenInfo]
    ) -> NoReturn:
        """Raise a syntax error that occured at a given AST node."""
        if isinstance(node, tokenize.TokenInfo):
            start = node.start
            end = node.end
        else:
            start = node.lineno, node.col_offset
            end = node.end_lineno, node.end_col_offset

        raise self._build_syntax_error(message, start, end)

    def raise_syntax_error_known_range(
        self,
        message: str,
        start_node: Union[ast.AST, tokenize.TokenInfo],
        end_node: Union[ast.AST, tokenize.TokenInfo]
    ) -> NoReturn:
        if isinstance(start_node, tokenize.TokenInfo):
            start = start_node.start
        else:
            start = start_node.lineno, start_node.col_offset

        if isinstance(end_node, tokenize.TokenInfo):
            end = end_node.end
        else:
            end = end_node.end_lineno, end_node.end_col_offset

        raise self._build_syntax_error(message, start, end)

    def raise_syntax_error_starting_from(
        self,
        message: str,
        start_node: Union[ast.AST, tokenize.TokenInfo]
    ) -> NoReturn:
        if isinstance(start_node, tokenize.TokenInfo):
            start = start_node.start
        else:
            start = start_node.lineno, start_node.col_offset

        last_token = self._tokenizer.diagnose()
```

(continues on next page)

(continued from previous page)

```
        raise self._build_syntax_error(message, start, last_token.start)

    def raise_syntax_error_invalid_target(
        self, target: Target, node: Optional[ast.AST]
    ) -> NoReturn:
        invalid_target = self.get_invalid_target(target, node)

        if invalid_target is None:
            return None

        if target in (Target.STAR_TARGETS, Target.FOR_TARGETS):
            msg = f"cannot assign to {self.get_expr_name(invalid_target)}"
        else:
            msg = f"cannot delete {self.get_expr_name(invalid_target)}"

        self.raise_syntax_error_known_location(msg, invalid_target)

    # scenic helpers
    def extend_new_specifiers(self, node: s.New, specifiers: List[ast.AST]) -> s.New:
        node.specifiers.extend(specifiers)
        return node
'''


# rule for adding hard keywords
# scenic_hard_keyword:


# STARTING RULES
# ==============

start: file

file[ast.Module]: a=[statements] ENDMARKER { ast.Module(body=a or [], type_ignores=[]) }
interactive[ast.Interactive]: a=statement_newline { ast.Interactive(body=a) }
eval[ast.Expression]: a=expressions NEWLINE* ENDMARKER { ast.Expression(body=a) }
func_type[ast.FunctionType]: '(' a=[type_expressions] ')' '->' b=expression NEWLINE*␣
↪ENDMARKER { ast.FunctionType(argtypes=a, returns=b) }
fstring[ast.Expr]: star_expressions

# GENERAL STATEMENTS
# ==================

statements[list]: a=statement+ { list(itertools.chain.from_iterable(a)) }

statement[list]: a=scenic_compound_stmt { [a] } | a=compound_stmt { [a] } | a=scenic_
↪stmts { a } | a=simple_stmts { a }

statement_newline[list]:
    | a=compound_stmt NEWLINE { [a] }
    | simple_stmts
```

(continues on next page)

(continued from previous page)

```
    | NEWLINE { [ast.Pass(LOCATIONS)] }
    | ENDMARKER { None }

simple_stmts[list]:
    | a=simple_stmt !';' NEWLINE { [a] } # Not needed, there for speedup
    | a=';'.simple_stmt+ [';'] NEWLINE { a }

scenic_stmts[list]:
    | a=scenic_stmt !';' NEWLINE { [a] } # Not needed, there for speedup
    | a=';'.scenic_stmt+ [';'] NEWLINE { a }

# NOTE: assignment MUST precede expression, else parsing a simple assignment
# will throw a SyntaxError.
simple_stmt (memo):
    | assignment
    | e=star_expressions { ast.Expr(value=e, LOCATIONS) }
    | &'return' return_stmt
    | &('import' | 'from') import_stmt
    | &'raise' raise_stmt
    | 'pass' { ast.Pass(LOCATIONS) }
    | &'del' del_stmt
    | &'yield' yield_stmt
    | &'assert' assert_stmt
    | 'break' { ast.Break(LOCATIONS) }
    | 'continue' { ast.Continue(LOCATIONS) }
    | &'global' global_stmt
    | &'nonlocal' nonlocal_stmt

compound_stmt:
    | &('def' | '@' | 'async') function_def
    | &'if' if_stmt
    | &('class' | '@') class_def
    | &('with' | 'async') with_stmt
    | &('for' | 'async') for_stmt
    | &'try' try_stmt
    | &'while' while_stmt
    | match_stmt

scenic_stmt:
    | scenic_model_stmt
    | scenic_tracked_assignment
    | scenic_param_stmt
    | scenic_require_stmt
    | scenic_record_initial_stmt
    | scenic_record_final_stmt
    | scenic_record_stmt
    | scenic_mutate_stmt
    | scenic_terminate_simulation_when_stmt
    | scenic_terminate_when_stmt
    | scenic_terminate_after_stmt
    | scenic_take_stmt
    | scenic_wait_stmt
```

(continues on next page)

(continued from previous page)

```
    | scenic_terminate_simulation_stmt
    | scenic_terminate_stmt
    | scenic_do_choose_stmt
    | scenic_do_shuffle_stmt
    | scenic_do_for_stmt
    | scenic_do_until_stmt
    | scenic_do_stmt
    | scenic_abort_stmt
    | scenic_simulator_stmt

scenic_compound_stmt:
    | scenic_tracked_assign_new_stmt
    | scenic_assign_new_stmt
    | scenic_expr_new_stmt
    | scenic_behavior_def
    | scenic_monitor_def
    | scenic_scenario_def
    | scenic_try_interrupt_stmt
    | scenic_override_stmt

# SIMPLE STATEMENTS
# =================

# NOTE: annotated_rhs may start with 'yield'; yield_expr must start with 'yield'
assignment:
    | a=NAME ':' b=expression c=['=' d=annotated_rhs { d }] {
        self.check_version(
            (3, 6),
            "Variable annotation syntax is",
            ast.AnnAssign(
                target=ast.Name(
                    id=a.string,
                    ctx=Store,
                    lineno=a.start[0],
                    col_offset=a.start[1],
                    end_lineno=a.end[0],
                    end_col_offset=a.end[1],
                ),
                annotation=b,
                value=c,
                simple=1,
                LOCATIONS,
            )
        ) }
    | a=('(' b=single_target ')' { b }
         | single_subscript_attribute_target) ':' b=expression c=['=' d=annotated_rhs {␣
↪d }] {
        self.check_version(
            (3, 6),
            "Variable annotation syntax is",
            ast.AnnAssign(
                target=a,
```

(continues on next page)

(continued from previous page)

```
                annotation=b,
                value=c,
                simple=0,
                LOCATIONS,
            )
        )
    }
    | a=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_
↪COMMENT] {
        ast.Assign(targets=a, value=b, type_comment=tc, LOCATIONS)
    }
    | a=single_target b=augassign ~ c=(yield_expr | star_expressions) {
        ast.AugAssign(target = a, op=b, value=c, LOCATIONS)
    }
    | invalid_assignment

annotated_rhs: yield_expr | star_expressions

augassign:
    | '+=' { ast.Add() }
    | '-=' { ast.Sub() }
    | '*=' { ast.Mult() }
    | '@=' { self.check_version((3, 5), "The '@' operator is", ast.MatMult()) }
    | '/=' { ast.Div() }
    | '%=' { ast.Mod() }
    | '&=' { ast.BitAnd() }
    | '|=' { ast.BitOr() }
    | '^=' { ast.BitXor() }
    | '<<=' { ast.LShift() }
    | '>>=' { ast.RShift() }
    | '**=' { ast.Pow() }
    | '//=' { ast.FloorDiv() }

return_stmt[ast.Return]:
    | 'return' a=[star_expressions] { ast.Return(value=a, LOCATIONS) }

raise_stmt[ast.Raise]:
    | 'raise' a=expression b=['from' z=expression { z }] { ast.Raise(exc=a, cause=b,␣
↪LOCATIONS) }
    | 'raise' { ast.Raise(exc=None, cause=None, LOCATIONS) }

global_stmt[ast.Global]: 'global' a=','.NAME+ {
    ast.Global(names=[n.string for n in a], LOCATIONS)
}

nonlocal_stmt[ast.Nonlocal]: 'nonlocal' a=','.NAME+ {
    ast.Nonlocal(names=[n.string for n in a], LOCATIONS)
}

del_stmt[ast.Delete]:
    | 'del' a=del_targets &(';' | NEWLINE) { ast.Delete(targets=a, LOCATIONS) }
    | invalid_del_stmt
```

(continues on next page)

(continued from previous page)

```
yield_stmt[ast.Expr]: y=yield_expr { ast.Expr(value=y, LOCATIONS) }

assert_stmt[ast.Assert]: 'assert' a=expression b=[',' z=expression { z }] {
    ast.Assert(test=a, msg=b, LOCATIONS)
}

import_stmt[ast.Import]: import_name | import_from

# Import statements
# -----------------

import_name[ast.Import]: 'import' a=dotted_as_names { ast.Import(names=a, LOCATIONS) }

# note below: the ('.' | '...') is necessary because '...' is tokenized as ELLIPSIS
import_from[ast.ImportFrom]:
    | 'from' a=('.' | '...')* b=dotted_name 'import' c=import_from_targets {
        ast.ImportFrom(module=b, names=c, level=self.extract_import_level(a), LOCATIONS)
     }
    | 'from' a=('.' | '...')+ 'import' b=import_from_targets {
        ast.ImportFrom(names=b, level=self.extract_import_level(a), LOCATIONS)
        if sys.version_info >= (3, 9) else
        ast.ImportFrom(module=None, names=b, level=self.extract_import_level(a),␣
↪LOCATIONS)
     }
import_from_targets[List[ast.alias]]:
    | '(' a=import_from_as_names [','] ')' { a }
    | import_from_as_names !','
    | '*' { [ast.alias(name="*", asname=None, LOCATIONS)] }
    | invalid_import_from_targets
import_from_as_names[List[ast.alias]]:
    | a=','.import_from_as_name+ { a }
import_from_as_name[ast.alias]:
    | a=NAME b=['as' z=NAME { z.string }] { ast.alias(name=a.string, asname=b,␣
↪LOCATIONS) }
dotted_as_names[List[ast.alias]]:
    | a=','.dotted_as_name+ { a }
dotted_as_name[ast.alias]:
    | a=dotted_name b=['as' z=NAME { z.string }] { ast.alias(name=a, asname=b,␣
↪LOCATIONS) }
dotted_name[str]:
    | a=dotted_name '.' b=NAME { a + "." + b.string }
    | a=NAME { a.string }

# COMPOUND STATEMENTS
# ===================

# Common elements
# ---------------

block[list] (memo):
    | NEWLINE INDENT a=statements DEDENT { a }
```

(continues on next page)

(continued from previous page)

```
    | simple_stmts
    | invalid_block

decorators: decorator+
decorator:
    | a=('@' f=dec_maybe_call NEWLINE { f }) { a }
    | a=('@' f=named_expression NEWLINE { f }) {
        self.check_version((3, 9), "Generic decorator are",  a)
     }
dec_maybe_call:
    | dn=dec_primary '(' z=[arguments] ')' {
        ast.Call(func=dn, args=z[0] if z else [], keywords=z[1] if z else [], LOCATIONS)
     }
    | dec_primary
dec_primary:
    | a=dec_primary '.' b=NAME { ast.Attribute(value=a, attr=b.string, ctx=Load,␣
↪LOCATIONS) }
    | a=NAME { ast.Name(id=a.string, ctx=Load, LOCATIONS) }

# Class definitions
# -----------------

class_def[ast.ClassDef]:
    | a=decorators b=class_def_raw { self.set_decorators(b, a) }
    | class_def_raw

class_def_raw[ast.ClassDef]:
    | invalid_class_def_raw
    | 'class' a=NAME b=['(' z=[arguments] ')' { z }] &&':' c=scenic_class_def_block {
        ast.ClassDef(
            a.string,
            bases=b[0] if b else [],
            keywords=b[1] if b else [],
            body=c,
            decorator_list=[],
            LOCATIONS,
        )
     }

scenic_class_def_block:
    | NEWLINE INDENT a=scenic_class_statements DEDENT { a }
    | simple_stmts
    | invalid_block

scenic_class_statements[list]: a=scenic_class_statement+ { list(itertools.chain.from_
↪iterable(a)) }

scenic_class_statement[list]:
    | a=scenic_class_property_stmt { [a] }
    | a=compound_stmt { [a] }
    | a=scenic_stmts { a }
    | a=simple_stmts { a }
```

(continues on next page)

(continued from previous page)

```
scenic_class_property_stmt:
    # not a simple statement; reads NEWLINE
    | a=NAME b=['[' attrs=','.scenic_class_property_attribute+ ']' { attrs } ] ':'␣
↪c=expression NEWLINE {
        s.PropertyDef(
            property=a.string,
            attributes=b if b is not None else [],
            value=c,
            LOCATIONS,
        )
     }

# fail if `NAME [ <expr> ]` pattern is found at top level of class definition and
# <expr> is neither `additive` nor `dynamic`
scenic_class_property_attribute: &&(
      "additive" { s.Additive(LOCATIONS) }
    | "dynamic" { s.Dynamic(LOCATIONS) }
    | "final" { s.Final(LOCATIONS) }
)

# Multiline Specifiers
# --------------------
scenic_assign_new_stmt:
    | a=(z=star_targets '=' { z })+ b=(scenic_new_block) !'=' tc=[TYPE_COMMENT] {
        ast.Assign(targets=a, value=b, type_comment=tc, LOCATIONS)
     }

scenic_tracked_assign_new_stmt:
    | a=scenic_tracked_name '=' b=scenic_new_block { s.TrackedAssign(target=a, value=b,␣
↪LOCATIONS) }

scenic_expr_new_stmt: a=scenic_new_block { ast.Expr(value=a, LOCATIONS) }

scenic_new_block:
    | a=scenic_new_expr ',' NEWLINE INDENT b=scenic_new_block_body DEDENT {
       self.extend_new_specifiers(a, b)
     }

scenic_new_block_body:
    # without trailing comma
    | b=(x=scenic_specifiers ',' NEWLINE { x })* c=scenic_specifiers NEWLINE {
        list(itertools.chain.from_iterable(b)) + c
     }
    # with trailing comma
    | b=(x=scenic_specifiers ',' NEWLINE { x })+ {
       list(itertools.chain.from_iterable(b))
     }


# Behavior
# --------
```

(continues on next page)

(continued from previous page)

```
scenic_behavior_def:
    | "behavior" a=NAME '(' b=[params] ')' &&':' c=scenic_behavior_def_block {
        s.BehaviorDef(
            a.string,
            args=b or self.make_arguments(None, [], None, [], None),
            docstring=c[0],
            header=c[1],
            body=c[2],
            LOCATIONS,
        )
     }

scenic_behavior_def_block:
    # behavior definition must have at least one statement that is not a precondition/
↪invariant definition
    | NEWLINE INDENT a=[x=STRING NEWLINE { x.string }] b=[scenic_behavior_header]␣
↪c=scenic_behavior_statements DEDENT { (a, b or [], c) }
    | invalid_block

scenic_behavior_statements[list]: a=scenic_behavior_statement+ { list(itertools.chain.
↪from_iterable(a)) }

# statements available inside behavior (normal statements + dynamic statements -␣
↪precondition/invariant)
scenic_behavior_statement[list]:
    | scenic_invalid_behavior_statement
    | a=statement { a }

scenic_invalid_behavior_statement:
    | a="invariant" ':' a=expression {
        self.raise_syntax_error_known_location("invariant can only be set at the␣
↪beginning of behavior definitions", a)
     }
    | a="precondition" ':' a=expression {
        self.raise_syntax_error_known_location("precondition can only be set at the␣
↪beginning of behavior definitions", a)
     }

scenic_behavior_header: a=(x=(scenic_precondition_stmt | scenic_invariant_stmt) NEWLINE
↪{ x })+ { a }

scenic_precondition_stmt:
    | "precondition" ':' a=expression { s.Precondition(value=a, LOCATIONS) }

scenic_invariant_stmt:
    | "invariant" ':' a=expression { s.Invariant(value=a, LOCATIONS) }


# Monitor
# -------
```

(continues on next page)

(continued from previous page)

```
scenic_monitor_def:
    | invalid_monitor
    | "monitor" a=NAME '(' b=[params] ')' &&':' c=scenic_monitor_def_block {
        s.MonitorDef(
            a.string,
            args=b or self.make_arguments(None, [], None, [], None),
            docstring=c[0],
            body=c[1],
            LOCATIONS
        )
     }

invalid_monitor[NoReturn]:
    | "monitor" NAME a=':' {
            self.raise_syntax_error_known_location("2.0-style monitor must be converted␣
↪to use parentheses and explicit require", a)
         }

scenic_monitor_def_block:
    | NEWLINE INDENT a=[x=STRING NEWLINE { x.string }] b=scenic_monitor_statements␣
↪DEDENT { (a, b) }

scenic_monitor_statements[list]: a=statement+ { list(itertools.chain.from_iterable(a)) }

# Modular Scenario
# ----------------

scenic_scenario_def:
    | "scenario" a=NAME b=['(' z=[params] ')' { z }] &&':' c=scenic_scenario_def_block {
        s.ScenarioDef(
            a.string,
            args=b or self.make_arguments(None, [], None, [], None),
            docstring=c[0],
            header=c[1],
            setup=c[2],
            compose=c[3],
            LOCATIONS,
        )
     }

# returns a four-tuple (docstring, header, setup block, compose block)
scenic_scenario_def_block:
    | NEWLINE INDENT a=[x=STRING NEWLINE { x.string }] b=[scenic_behavior_header]␣
↪c=[scenic_scenario_setup_block] d=[scenic_scenario_compose_block] DEDENT { (a, b or [],
↪ c or [], d or []) }
    | NEWLINE INDENT a=[x=STRING NEWLINE { x.string }] b=statements DEDENT { (a, [], b,␣
↪[]) }

scenic_scenario_setup_block:
    | "setup" &&':' b=block { b }

scenic_scenario_compose_block:
```

(continues on next page)

(continued from previous page)

```
    | "compose" &&':' b=block { b }

# Override
# --------

scenic_override_stmt:
    # restricting `e` to `primary` rather than `expression` to disambiguate keywords that␣
↪are both specifiers and operators (e.g. `at`, `offset by`)
    | "override" e=primary ss=scenic_specifiers NEWLINE { s.Override(target=e,␣
↪specifiers=ss) }
    | "override" e=primary ss=scenic_specifiers ',' NEWLINE INDENT t=scenic_new_block_
↪body DEDENT {
        s.Override(target=e, specifiers=ss + t)
     }

# Function definitions
# --------------------

function_def[Union[ast.FunctionDef, ast.AsyncFunctionDef]]:
    | d=decorators f=function_def_raw { self.set_decorators(f, d) }
    | f=function_def_raw {self.set_decorators(f, [])}

function_def_raw[Union[ast.FunctionDef, ast.AsyncFunctionDef]]:
    | invalid_def_raw
    | 'def' n=NAME &&'(' params=[params] ')' a=['->' z=expression { z }] &&':' tc=[func_
↪type_comment] b=block {
        ast.FunctionDef(
            name=n.string,
            args=params or self.make_arguments(None, [], None, [], None),
            returns=a,
            body=b,
            type_comment=tc,
            LOCATIONS,
        )
     }
    | 'async' 'def' n=NAME &&'(' params=[params] ')' a=['->' z=expression { z }] &&':'␣
↪tc=[func_type_comment] b=block {
       self.check_version(
            (3, 5),
            "Async functions are",
            ast.AsyncFunctionDef(
                name=n.string,
                args=params or self.make_arguments(None, [], None, [], None),
                returns=a,
                body=b,
                type_comment=tc,
                LOCATIONS,
            )
        )
     }

# Function parameters
```

(continues on next page)

(continued from previous page)

```
# ------------------

params:
    | invalid_parameters
    | parameters

parameters[ast.arguments]:
    | a=slash_no_default b=param_no_default* c=param_with_default* d=[star_etc] {
        self.check_version(
            (3, 8), "Positional only arguments are", self.make_arguments(a, [], b, c, d)
        )
     }
    | a=slash_with_default b=param_with_default* c=[star_etc] {
        self.check_version(
            (3, 8),
            "Positional only arguments are",
            self.make_arguments(None, a, None, b, c),
        )
     }
    | a=param_no_default+ b=param_with_default* c=[star_etc] {
        self.make_arguments(None, [], a, b, c)
     }
    | a=param_with_default+ b=[star_etc] {
        self.make_arguments(None, [], None, a, b)
     }
    | a=star_etc { self.make_arguments(None, [], None, None, a) }

# Some duplication here because we can't write (',' | &')'),
# which is because we don't support empty alternatives (yet).
#

slash_no_default[List[Tuple[ast.arg, None]]]:
    | a=param_no_default+ '/' ',' { [(p, None) for p in a] }
    | a=param_no_default+ '/' &')' { [(p, None) for p in a] }
slash_with_default[List[Tuple[ast.arg, Any]]]:
    | a=param_no_default* b=param_with_default+ '/' ',' { ([(p, None) for p in a] if a␣
↪else []) + b }
    | a=param_no_default* b=param_with_default+ '/' &')' { ([(p, None) for p in a] if a␣
↪else []) + b }

star_etc[Tuple[Optional[ast.arg], List[Tuple[ast.arg, Any]], Optional[ast.arg]]]:
    | invalid_star_etc
    | '*' a=param_no_default b=param_maybe_default* c=[kwds] { (a, b, c) }
    | '*' ',' b=param_maybe_default+ c=[kwds] { (None, b, c) }
    | a=kwds { (None, [], a) }

kwds[ast.arg]:
    | invalid_kwds
    | '**' a=param_no_default { a }

# One parameter.  This *includes* a following comma and type comment.
#
```

(continues on next page)

(continued from previous page)

```
# There are three styles:
# - No default
# - With default
# - Maybe with default
#
# There are two alternative forms of each, to deal with type comments:
# - Ends in a comma followed by an optional type comment
# - No comma, optional type comment, must be followed by close paren
# The latter form is for a final parameter without trailing comma.
#

param_no_default[ast.arg]:
    | a=param ',' tc=TYPE_COMMENT? { self.set_arg_type_comment(a, tc) }
    | a=param tc=TYPE_COMMENT? &')' { self.set_arg_type_comment(a, tc) }
param_with_default[Tuple[ast.arg, Any]]:
    | a=param c=default ',' tc=TYPE_COMMENT? { (self.set_arg_type_comment(a, tc), c) }
    | a=param c=default tc=TYPE_COMMENT? &')' { (self.set_arg_type_comment(a, tc), c) }
param_maybe_default[Tuple[ast.arg, Any]]:
    | a=param c=default? ',' tc=TYPE_COMMENT? { (self.set_arg_type_comment(a, tc), c) }
    | a=param c=default? tc=TYPE_COMMENT? &')' { (self.set_arg_type_comment(a, tc), c) }
param: a=NAME b=annotation? { ast.arg(arg=a.string, annotation=b, LOCATIONS) }
annotation: ':' a=expression { a }
default: '=' a=expression { a } | invalid_default

# If statement
# ------------

if_stmt[ast.If]:
    | invalid_if_stmt
    | 'if' a=named_expression ':' b=block c=elif_stmt { ast.If(test=a, body=b, orelse=c␣
↪or [], LOCATIONS) }
    | 'if' a=named_expression ':' b=block c=[else_block] { ast.If(test=a, body=b,␣
↪orelse=c or [], LOCATIONS) }
elif_stmt[List[ast.If]]:
    | invalid_elif_stmt
    | 'elif' a=named_expression ':' b=block c=elif_stmt { [ast.If(test=a, body=b,␣
↪orelse=c, LOCATIONS)] }
    | 'elif' a=named_expression ':' b=block c=[else_block] { [ast.If(test=a, body=b,␣
↪orelse=c or [], LOCATIONS)] }
else_block[list]:
    | invalid_else_stmt
    | 'else' &&':' b=block { b }

# While statement
# ---------------

while_stmt[ast.While]:
    | invalid_while_stmt
    | 'while' a=named_expression ':' b=block c=[else_block] {
        ast.While(test=a, body=b, orelse=c or [], LOCATIONS)
     }
```

(continues on next page)

(continued from previous page)

```
# For statement
# -------------

for_stmt[Union[ast.For, ast.AsyncFor]]:
    | invalid_for_stmt
    | 'for' t=star_targets 'in' ~ ex=star_expressions &&':' tc=[TYPE_COMMENT] b=block␣
↪el=[else_block] {
        ast.For(target=t, iter=ex, body=b, orelse=el or [], type_comment=tc, LOCATIONS) }
    | 'async' 'for' t=star_targets 'in' ~ ex=star_expressions ':' tc=[TYPE_COMMENT]␣
↪b=block el=[else_block] {
        self.check_version(
            (3, 5),
            "Async for loops are",
            ast.AsyncFor(target=t, iter=ex, body=b, orelse=el or [], type_comment=tc,␣
↪LOCATIONS)) }
    | invalid_for_target

# With statement
# --------------

with_stmt[Union[ast.With, ast.AsyncWith]]:
    | invalid_with_stmt_indent
    | 'with' '(' a=','.with_item+ ','? ')' ':' b=block {
        self.check_version(
            (3, 9),
            "Parenthesized with items",
             ast.With(items=a, body=b, LOCATIONS)
        )
     }
    | 'with' a=','.with_item+ ':' tc=[TYPE_COMMENT] b=block {
        ast.With(items=a, body=b, type_comment=tc, LOCATIONS)
     }
    | 'async' 'with' '(' a=','.with_item+ ','? ')' ':' b=block {
       self.check_version(
            (3, 9),
            "Parenthesized with items",
            ast.AsyncWith(items=a, body=b, LOCATIONS)
        )
     }
    | 'async' 'with' a=','.with_item+ ':' tc=[TYPE_COMMENT] b=block {
       self.check_version(
            (3, 5),
            "Async with statements are",
            ast.AsyncWith(items=a, body=b, type_comment=tc, LOCATIONS)
        )
     }
    | invalid_with_stmt

with_item[ast.withitem]:
    | e=expression 'as' t=star_target &(',' | ')' | ':') {
        ast.withitem(context_expr=e, optional_vars=t)
     }
```

(continues on next page)

(continued from previous page)

```
    | invalid_with_item
    | e=expression { ast.withitem(context_expr=e, optional_vars=None) }

# Try statement
# -------------

try_stmt[ast.Try]:
    | invalid_try_stmt
    | 'try' &&':' b=block f=finally_block {
        ast.Try(body=b, handlers=[], orelse=[], finalbody=f, LOCATIONS)
     }
    | 'try' &&':' b=block ex=except_block+ el=[else_block] f=[finally_block] {
        ast.Try(body=b, handlers=ex, orelse=el or [], finalbody=f or [], LOCATIONS)
     }

scenic_try_interrupt_stmt[s.TryInterrupt]:
    | 'try' &&':' b=block iw=interrupt_when_block+ ex=except_block* el=[else_block]␣
↪f=[finally_block] {
        s.TryInterrupt(
            body=b,
            interrupt_when_handlers=iw,
            except_handlers=ex,
            orelse=el or [],
            finalbody=f or [],
            LOCATIONS,
        )
     }

# Interrupt statement
# -------------------

interrupt_when_block:
    | "interrupt" "when" e=expression &&':' b=block { s.InterruptWhenHandler(cond=e,␣
↪body=b, LOCATIONS) }

# Except statement
# ----------------

except_block[ast.ExceptHandler]:
    | invalid_except_stmt_indent
    | 'except' e=expression t=['as' z=NAME { z.string }] ':' b=block {
        ast.ExceptHandler(type=e, name=t, body=b, LOCATIONS) }
    | 'except' ':' b=block { ast.ExceptHandler(type=None, name=None, body=b, LOCATIONS) }
    | invalid_except_stmt
finally_block[list]:
    | invalid_finally_stmt
    | 'finally' &&':' a=block { a }

# Match statement
# ---------------

# We cannot do version checks here since the production will occur after any other
```

(continues on next page)

(continued from previous page)

```
# production which will have failed since the ast module does not have the right nodes.
match_stmt["ast.Match"]:
    | "match" subject=subject_expr ':' NEWLINE INDENT cases=case_block+ DEDENT {
        ast.Match(subject=subject, cases=cases, LOCATIONS)
     }
    | invalid_match_stmt

# Version checking here allows to avoid tracking down every single possible production
subject_expr:
    | value=star_named_expression ',' values=star_named_expressions? {
        self.check_version(
            (3, 10),
            "Pattern matching is",
            ast.Tuple(elts=[value] + (values or []), ctx=Load, LOCATIONS)
        )
     }
    | e=named_expression { self.check_version((3, 10), "Pattern matching is", e)}

case_block["ast.match_case"]:
    | invalid_case_block
    | "case" pattern=patterns guard=guard? ':' body=block {
        ast.match_case(pattern=pattern, guard=guard, body=body)
     }

guard: 'if' guard=named_expression { guard }

patterns:
    | patterns=open_sequence_pattern {
        ast.MatchSequence(patterns=patterns, LOCATIONS)
     }
    | pattern

pattern:
    | as_pattern
    | or_pattern

as_pattern["ast.MatchAs"]:
    | pattern=or_pattern 'as' target=pattern_capture_target {
        ast.MatchAs(pattern=pattern, name=target, LOCATIONS)
     }
    | invalid_as_pattern

or_pattern["ast.MatchOr"]:
    | patterns='|'.closed_pattern+ {
        ast.MatchOr(patterns=patterns, LOCATIONS) if len(patterns) > 1 else patterns[0]
     }

closed_pattern:
    | literal_pattern
    | capture_pattern
    | wildcard_pattern
    | value_pattern
```

(continues on next page)

(continued from previous page)

```
    | group_pattern
    | sequence_pattern
    | mapping_pattern
    | class_pattern

# Literal patterns are used for equality and identity constraints
literal_pattern:
    | value=signed_number !('+' | '-') { ast.MatchValue(value=value, LOCATIONS) }
    | value=complex_number { ast.MatchValue(value=value, LOCATIONS) }
    | value=strings { ast.MatchValue(value=value, LOCATIONS) }
    | 'None' { ast.MatchSingleton(value=None, LOCATIONS) }
    | 'True' { ast.MatchSingleton(value=True, LOCATIONS) }
    | 'False' { ast.MatchSingleton(value=False, LOCATIONS) }

# Literal expressions are used to restrict permitted mapping pattern keys
literal_expr:
    | signed_number !('+' | '-')
    | complex_number
    | strings
    | 'None' { ast.Constant(value=None, LOCATIONS) }
    | 'True' { ast.Constant(value=True, LOCATIONS) }
    | 'False' { ast.Constant(value=False, LOCATIONS) }

complex_number:
    | real=signed_real_number '+' imag=imaginary_number {
        ast.BinOp(left=real, op=ast.Add(), right=imag, LOCATIONS)
     }
    | real=signed_real_number '-' imag=imaginary_number  {
        ast.BinOp(left=real, op=ast.Sub(), right=imag, LOCATIONS)
     }

signed_number:
    | a=NUMBER { ast.Constant(value=ast.literal_eval(a.string), LOCATIONS) }
    | '-' a=NUMBER {
        ast.UnaryOp(
            op=ast.USub(),
            operand=ast.Constant(
                value=ast.literal_eval(a.string),
                lineno=a.start[0],
                col_offset=a.start[1],
                end_lineno=a.end[0],
                end_col_offset=a.end[1]
            ),
            LOCATIONS,
        )
     }

signed_real_number:
    | real_number
    | '-' real=real_number { ast.UnaryOp(op=ast.USub(), operand=real, LOCATIONS) }

real_number[ast.Constant]:
```

(continues on next page)

(continued from previous page)

```
    | real=NUMBER { ast.Constant(value=self.ensure_real(real), LOCATIONS) }

imaginary_number[ast.Constant]:
    | imag=NUMBER { ast.Constant(value=self.ensure_imaginary(imag), LOCATIONS) }

capture_pattern:
    | target=pattern_capture_target {
        ast.MatchAs(pattern=None, name=target, LOCATIONS)
     }

pattern_capture_target[str]:
    | !"_" name=NAME !('.' | '(' | '=') { name.string }

wildcard_pattern["ast.MatchAs"]:
    | "_" { ast.MatchAs(pattern=None, target=None, LOCATIONS) }

value_pattern["ast.MatchValue"]:
    | attr=attr !('.' | '(' | '=') { ast.MatchValue(value=attr, LOCATIONS) }

attr[ast.Attribute]:
    | value=name_or_attr '.' attr=NAME {
        ast.Attribute(value=value, attr=attr.string, ctx=Load, LOCATIONS)
     }

name_or_attr:
    | attr
    | name=NAME { ast.Name(id=name.string, ctx=Load, LOCATIONS) }

group_pattern:
    | '(' pattern=pattern ')' { pattern }

sequence_pattern["ast.MatchSequence"]:
    | '[' patterns=maybe_sequence_pattern? ']' { ast.MatchSequence(patterns=patterns or␣
↪[], LOCATIONS) }
    | '(' patterns=open_sequence_pattern? ')' { ast.MatchSequence(patterns=patterns or␣
↪[], LOCATIONS) }

open_sequence_pattern:
    | pattern=maybe_star_pattern ',' patterns=maybe_sequence_pattern? {
        [pattern] + (patterns or [])
     }

maybe_sequence_pattern:
    | patterns=','.maybe_star_pattern+ ','? { patterns }

maybe_star_pattern:
    | star_pattern
    | pattern

star_pattern:
    | '*' target=pattern_capture_target { ast.MatchStar(name=target, LOCATIONS) }
    | '*' wildcard_pattern { ast.MatchStar(target=None, LOCATIONS) }
```

(continues on next page)

(continued from previous page)

```
mapping_pattern:
    | '{' '}' { ast.MatchMapping(keys=[], patterns=[], rest=None, LOCATIONS) }
    | '{' rest=double_star_pattern ','? '}' {
        ast.MatchMapping(keys=[], patterns=[], rest=rest, LOCATIONS) }
    | '{' items=items_pattern ',' rest=double_star_pattern ','? '}' {
        ast.MatchMapping(
            keys=[k for k,_ in items],
            patterns=[p for _, p in items],
            rest=rest,
            LOCATIONS,
        )
     }
    | '{' items=items_pattern ','? '}' {
        ast.MatchMapping(
            keys=[k for k,_ in items],
            patterns=[p for _, p in items],
            rest=None,
            LOCATIONS,
        )
     }

items_pattern:
    | ','.key_value_pattern+

key_value_pattern:
    | key=(literal_expr | attr) ':' pattern=pattern { (key, pattern) }

double_star_pattern:
    | '**' target=pattern_capture_target { target }

class_pattern["ast.MatchClass"]:
    | cls=name_or_attr '(' ')' {
        ast.MatchClass(cls=cls, patterns=[], kwd_attrs=[], kwd_patterns=[], LOCATIONS)
     }
    | cls=name_or_attr '(' patterns=positional_patterns ','? ')' {
        ast.MatchClass(cls=cls, patterns=patterns, kwd_attrs=[], kwd_patterns=[],␣
↪LOCATIONS)
     }
    | cls=name_or_attr '(' keywords=keyword_patterns ','? ')' {
        ast.MatchClass(
            cls=cls,
            patterns=[],
            kwd_attrs=[k for k, _ in keywords],
            kwd_patterns=[p for _, p in keywords],
            LOCATIONS,
        )
     }
    | cls=name_or_attr '(' patterns=positional_patterns ',' keywords=keyword_patterns ',
↪'? ')' {
        ast.MatchClass(
            cls=cls,
```

(continues on next page)

(continued from previous page)

```
            patterns=patterns,
            kwd_attrs=[k for k, _ in keywords],
            kwd_patterns=[p for _, p in keywords],
            LOCATIONS,
        )
    }
    | invalid_class_pattern

positional_patterns:
    | args=','.pattern+ { args }

keyword_patterns:
    | ','.keyword_pattern+

keyword_pattern:
    | arg=NAME '=' value=pattern { (arg.string, value) }

# EXPRESSIONS
# -----------

expressions:
    | a=expression b=(',' c=expression { c })+ [','] {
        ast.Tuple(elts=[a] + b, ctx=Load, LOCATIONS) }
    | a=expression ',' { ast.Tuple(elts=[a], ctx=Load, LOCATIONS) }
    | expression

expression (memo):
    | invalid_scenic_instance_creation
    | invalid_expression
    | invalid_legacy_expression
    | a=disjunction 'if' b=disjunction 'else' c=disjunction {
        ast.IfExp(body=a, test=b, orelse=c, LOCATIONS)
    }
    | disjunction
    | lambdef

scenic_temporal_expression (memo):
    | invalid_expression
    | invalid_legacy_expression
    | a=scenic_until 'if' b=scenic_until 'else' c=scenic_until {
        ast.IfExp(body=a, test=b, orelse=c, LOCATIONS)
    }
    | scenic_until
    | lambdef

yield_expr:
    | 'yield' 'from' a=expression { ast.YieldFrom(value=a, LOCATIONS) }
    | 'yield' a=[star_expressions] { ast.Yield(value=a, LOCATIONS) }

star_expressions:
    | a=star_expression b=(',' c=star_expression { c })+ [','] {
        ast.Tuple(elts=[a] + b, ctx=Load, LOCATIONS) }
```

(continues on next page)

(continued from previous page)

```
    | a=star_expression ',' { ast.Tuple(elts=[a], ctx=Load, LOCATIONS) }
    | star_expression

star_expression (memo):
    | '*' a=bitwise_or { ast.Starred(value=a, ctx=Load, LOCATIONS) }
    | expression

star_named_expressions: a=','.star_named_expression+ [','] { a }

star_named_expression:
    | '*' a=bitwise_or { ast.Starred(value=a, ctx=Load, LOCATIONS) }
    | named_expression

assignment_expression:
    | a=NAME ':=' ~ b=expression {
        self.check_version(
            (3, 8),
            "The ':=' operator is",
            ast.NamedExpr(
                target=ast.Name(
                    id=a.string,
                    ctx=Store,
                    lineno=a.start[0],
                    col_offset=a.start[1],
                    end_lineno=a.end[0],
                    end_col_offset=a.end[1]
                ),
                value=b,
                LOCATIONS,
            )
        )
     }

named_expression:
    | assignment_expression
    | invalid_named_expression
    | a=expression !':=' { a }

scenic_until (memo):
    | invalid_scenic_until
    | a=scenic_above_until 'until' b=scenic_above_until { s.UntilOp(a, b, LOCATIONS) }
    | scenic_above_until

scenic_above_until (memo):  # anything with precedence above "until"
    | scenic_temporal_prefix
    | scenic_implication

scenic_temporal_prefix (memo):
    | "next" e=scenic_above_until { s.Next(e, LOCATIONS) }
    | "eventually" e=scenic_above_until { s.Eventually(e, LOCATIONS) }
    | "always" e=scenic_above_until { s.Always(e, LOCATIONS) }
```

(continues on next page)

(continued from previous page)

```
scenic_implication (memo):
    | invalid_scenic_implication
    # exclude implication on RHS to disallow "A implies B implies C"
    | a=scenic_temporal_disjunction "implies" b=(scenic_temporal_prefix | scenic_
↪temporal_disjunction) { s.ImpliesOp(a, b, LOCATIONS) }
    | scenic_temporal_disjunction

disjunction (memo):
    | a=conjunction b=('or' c=conjunction { c })+ { ast.BoolOp(op=ast.Or(), values=[a] +␣
↪b, LOCATIONS) }
    | conjunction

scenic_temporal_disjunction (memo):
    | a=scenic_temporal_conjunction b=('or' c=(scenic_temporal_prefix | scenic_temporal_
↪conjunction) { c })+ { ast.BoolOp(op=ast.Or(), values=[a] + b, LOCATIONS) }
    | scenic_temporal_conjunction

conjunction (memo):
    | a=inversion b=('and' c=inversion { c })+ { ast.BoolOp(op=ast.And(), values=[a] + b,
↪ LOCATIONS) }
    | inversion

scenic_temporal_conjunction (memo):
    | a=scenic_temporal_inversion b=('and' c=(scenic_temporal_prefix | scenic_temporal_
↪inversion) { c })+ { ast.BoolOp(op=ast.And(), values=[a] + b, LOCATIONS) }
    | scenic_temporal_inversion

inversion (memo):
    # [SCENIC NOTE]: Fail `not visible <inversion>` to be handled later
    | 'not' !("visible" inversion) a=inversion { ast.UnaryOp(op=ast.Not(), operand=a,␣
↪LOCATIONS) }
    | comparison

scenic_temporal_inversion (memo):
    # Fail `not visible <inversion>` to be handled later
    | 'not' !("visible" scenic_temporal_inversion) a=(scenic_temporal_prefix | scenic_
↪temporal_inversion) { ast.UnaryOp(op=ast.Not(), operand=a, LOCATIONS) }
    | scenic_temporal_group
    | comparison

# Parsing temporal operators only inside "require" would require duplicating
# the entire rule hierarchy for expressions, since for example "always(X)" is a
# valid function call in ordinary Python but should be a temporal operator
# inside require. Instead, we only duplicate the boolean operators (above) and
# add the following rule which allows the introduction of parentheses without
# traversing all the way down to `atom`; the rule looks ahead for a binary
# temporal operator or the end of the parent expression in order to prevent
# matching expressions like "(X) > 5", which should be parsed by `comparison`
# instead. Invalid code like "(always(X)) > 5" is parsed as an ordinary
# expression (with a call to the "always" function) and caught in the compiler.
scenic_temporal_group: '(' a=scenic_temporal_expression ')' &('until' | 'or' | 'and' | ')
↪' | ';' | NEWLINE) { a }
```

(continues on next page)

(continued from previous page)

```
# Scenic instance creation
# ------------------------
scenic_new_expr: 'new' n=NAME ss=[scenic_specifiers] { s.New(className=n.string,␣
↪specifiers=ss, LOCATIONS) }
scenic_specifiers: ss=','.scenic_specifier+ { ss }
scenic_specifier:
    | scenic_valid_specifier
    | invalid_scenic_specifier
scenic_valid_specifier:
    | 'with' p=NAME v=expression { s.WithSpecifier(prop=p.string, value=v, LOCATIONS) }
    | 'at' position=expression { s.AtSpecifier(position=position, LOCATIONS) }
    | "offset" 'by' o=expression { s.OffsetBySpecifier(offset=o, LOCATIONS) }
    | "offset" "along" d=expression 'by' o=expression { s.
↪OffsetAlongSpecifier(direction=d, offset=o, LOCATIONS) }
    | direction=scenic_specifier_position_direction position=expression distance=['by'␣
↪e=expression { e }] {
        s.DirectionOfSpecifier(direction=direction, position=position, distance=distance,
↪ LOCATIONS)
     }
    | "beyond" v=expression 'by' o=expression b=['from' a=expression {a}] { s.
↪BeyondSpecifier(position=v, offset=o, base=b) }
    | "visible" b=['from' r=expression { r }] { s.VisibleSpecifier(base=b, LOCATIONS) }
    | 'not' "visible" b=['from' r=expression { r }] { s.NotVisibleSpecifier(base=b,␣
↪LOCATIONS) }
    | 'in' r=expression { s.InSpecifier(region=r, LOCATIONS) }
    | 'on' r=expression { s.OnSpecifier(region=r, LOCATIONS) }
    | "contained" 'in' r=expression { s.ContainedInSpecifier(region=r, LOCATIONS) }
    | "following" f=expression b=['from' e=expression {e}] 'for' d=expression {
        s.FollowingSpecifier(field=f, distance=d, base=b, LOCATIONS)
     }
    | "facing" "toward" p=expression { s.FacingTowardSpecifier(position=p, LOCATIONS) }
    | "facing" "away" "from" p=expression { s.FacingAwayFromSpecifier(position=p,␣
↪LOCATIONS) }
    | "facing" "directly" "toward" p=expression { s.
↪FacingDirectlyTowardSpecifier(position=p, LOCATIONS) }
    | "facing" "directly" "away" "from" p=expression { s.
↪FacingDirectlyAwayFromSpecifier(position=p, LOCATIONS) }
    | "facing" h=expression { s.FacingSpecifier(heading=h, LOCATIONS) }
    | "apparently" "facing" h=expression v=['from' a=expression { a }] {
        s.ApparentlyFacingSpecifier(heading=h, base=v, LOCATIONS)
     }

scenic_specifier_position_direction:
    | "left" "of" { s.LeftOf(LOCATIONS) }
    | "right" "of" { s.RightOf(LOCATIONS) }
    | "ahead" "of" { s.AheadOf(LOCATIONS) }
    | "behind" { s.Behind(LOCATIONS) }
    | "above" {s.Above(LOCATIONS)}
    | "below" {s.Below(LOCATIONS)}

# Comparisons operators
```

(continues on next page)

(continued from previous page)

```
# --------------------

comparison:
    | a=bitwise_or b=compare_op_bitwise_or_pair+ {
        ast.Compare(left=a, ops=self.get_comparison_ops(b), comparators=self.get_
↪comparators(b), LOCATIONS)
     }
    | bitwise_or

# Make a tuple of operator and comparator
compare_op_bitwise_or_pair:
    | eq_bitwise_or
    | noteq_bitwise_or
    | lte_bitwise_or
    | lt_bitwise_or
    | gte_bitwise_or
    | gt_bitwise_or
    | notin_bitwise_or
    | in_bitwise_or
    | isnot_bitwise_or
    | is_bitwise_or

eq_bitwise_or: '==' a=bitwise_or { (ast.Eq(), a) }
# Do not support the Barry as BDFL <> for not eq
noteq_bitwise_or[tuple]:
    | '!=' a=bitwise_or { (ast.NotEq(), a) }
lte_bitwise_or: '<=' a=bitwise_or { (ast.LtE(), a) }
lt_bitwise_or: '<' a=bitwise_or { (ast.Lt(), a) }
gte_bitwise_or: '>=' a=bitwise_or { (ast.GtE(), a) }
gt_bitwise_or: '>' a=bitwise_or { (ast.Gt(), a) }
notin_bitwise_or: 'not' 'in' a=bitwise_or { (ast.NotIn(), a) }
in_bitwise_or: 'in' a=bitwise_or { (ast.In(), a) }
isnot_bitwise_or: 'is' 'not' a=bitwise_or { (ast.IsNot(), a) }
is_bitwise_or: 'is' a=bitwise_or { (ast.Is(), a) }

# Logical operators
# -----------------

bitwise_or:
    | scenic_visible_from
    | scenic_not_visible_from
    | scenic_can_see
    | a=bitwise_or '|' b=bitwise_xor { ast.BinOp(left=a, op=ast.BitOr(), right=b,␣
↪LOCATIONS) }
    | bitwise_xor

scenic_visible_from: a=bitwise_or "visible" 'from' b=bitwise_xor { s.
↪VisibleFromOp(region=a, base=b, LOCATIONS) }

scenic_not_visible_from: a=bitwise_or "not" "visible" 'from' b=bitwise_xor { s.
↪NotVisibleFromOp(region=a, base=b, LOCATIONS) }
```

(continues on next page)

(continued from previous page)

```
scenic_can_see: a=bitwise_or "can" "see" b=bitwise_xor { s.CanSeeOp(left=a, right=b,␣
↪LOCATIONS) }

bitwise_xor:
    | scenic_offset_along
    | a=bitwise_xor '^' b=bitwise_and { ast.BinOp(left=a, op=ast.BitXor(), right=b,␣
↪LOCATIONS) }
    | bitwise_and

scenic_offset_along: a=bitwise_xor "offset" "along" b=bitwise_xor 'by' c=bitwise_and { s.
↪OffsetAlongOp(base=a, direction=b, offset=c, LOCATIONS) }

bitwise_and:
    | scenic_relative_to
    | a=bitwise_and '&' b=shift_expr { ast.BinOp(left=a, op=ast.BitAnd(), right=b,␣
↪LOCATIONS) }
    | shift_expr

scenic_relative_to: a=bitwise_and ("relative" 'to' | "offset" 'by') b=shift_expr { s.
↪RelativeToOp(left=a, right=b, LOCATIONS) }

shift_expr:
    | scenic_at
    | a=shift_expr '<<' b=sum { ast.BinOp(left=a, op=ast.LShift(), right=b, LOCATIONS) }
    | a=shift_expr '>>' b=sum { ast.BinOp(left=a, op=ast.RShift(), right=b, LOCATIONS) }
    | scenic_prefix_operators

scenic_at: a=shift_expr 'at' b=sum { s.FieldAtOp(left=a, right=b, LOCATIONS) }

# Scenic prefix operators
# -----------------------
scenic_prefix_operators:
    # relative position of
    | "relative" "position" "of" e1=expression 'from' e2=scenic_prefix_operators { s.
↪RelativePositionOp(target=e1, base=e2, LOCATIONS) }
    | "relative" "position" "of" e1=scenic_prefix_operators { s.
↪RelativePositionOp(target=e1, LOCATIONS) }
    # relative heading of
    | "relative" "heading" "of" e1=expression 'from' e2=scenic_prefix_operators { s.
↪RelativeHeadingOp(target=e1, base=e2, LOCATIONS) }
    | "relative" "heading" "of" e1=scenic_prefix_operators { s.
↪RelativeHeadingOp(target=e1, LOCATIONS) }
    # apparent heading of
    | "apparent" "heading" "of" e1=expression 'from' e2=scenic_prefix_operators { s.
↪ApparentHeadingOp(target=e1, base=e2, LOCATIONS) }
    | "apparent" "heading" "of" e1=scenic_prefix_operators { s.
↪ApparentHeadingOp(target=e1, LOCATIONS) }
    # distance from/to
    | &"distance" scenic_distance_from_op
    # distance past
    | "distance" "past" e1=expression 'of' e2=scenic_prefix_operators { s.
↪DistancePastOp(target=e1, base=e2, LOCATIONS) }
```

(continues on next page)

(continued from previous page)

```
    | "distance" "past" e1=scenic_prefix_operators { s.DistancePastOp(target=e1,␣
↪LOCATIONS) }
    # angle from/to
    | &"angle" scenic_angle_from_op
    # altitude from/to
    | &"altitude" scenic_altitude_from_op
    | "follow" e1=expression 'from' e2=expression 'for' e3=scenic_prefix_operators { s.
↪FollowOp(target=e1, base=e2, distance=e3, LOCATIONS) }
    | "visible" e=scenic_prefix_operators { s.VisibleOp(region=e, LOCATIONS) }
    | 'not' "visible" e=scenic_prefix_operators { s.NotVisibleOp(region=e, LOCATIONS) }
    | p=scenic_position_of_op_position 'of' e=scenic_prefix_operators { s.
↪PositionOfOp(position=p, target=e, LOCATIONS) }
    | sum

scenic_distance_from_op:
    | "distance" 'from' e1=expression 'to' e2=scenic_prefix_operators { s.
↪DistanceFromOp(target=e1, base=e2, LOCATIONS) }
    | "distance" 'to' e1=expression 'from' e2=scenic_prefix_operators { s.
↪DistanceFromOp(target=e1, base=e2, LOCATIONS) }
    | "distance" ('to'|'from') e1=scenic_prefix_operators { s.DistanceFromOp(target=e1,␣
↪LOCATIONS) }

scenic_angle_from_op:
    | "angle" 'from' e1=expression 'to' e2=scenic_prefix_operators { s.
↪AngleFromOp(base=e1, target=e2, LOCATIONS) }
    | "angle" 'to' e1=expression 'from' e2=scenic_prefix_operators { s.
↪AngleFromOp(target=e1, base=e2, LOCATIONS) }
    | "angle" 'to' e1=scenic_prefix_operators { s.AngleFromOp(target=e1, LOCATIONS) }
    | "angle" 'from' e1=scenic_prefix_operators { s.AngleFromOp(base=e1, LOCATIONS) }

scenic_altitude_from_op:
    | "altitude" 'from' e1=expression 'to' e2=scenic_prefix_operators { s.
↪AltitudeFromOp(base=e1, target=e2, LOCATIONS) }
    | "altitude" 'to' e1=expression 'from' e2=scenic_prefix_operators { s.
↪AltitudeFromOp(target=e1, base=e2, LOCATIONS) }
    | "altitude" 'to' e1=scenic_prefix_operators { s.AltitudeFromOp(target=e1,␣
↪LOCATIONS) }
    | "altitude" 'from' e1=scenic_prefix_operators { s.AltitudeFromOp(base=e1,␣
↪LOCATIONS) }

scenic_position_of_op_position:
    | "top" "front" "left" { s.TopFrontLeft(LOCATIONS) }
    | "top" "front" "right" { s.TopFrontRight(LOCATIONS) }
    | "top" "back" "left" { s.TopBackLeft(LOCATIONS) }
    | "top" "back" "right" { s.TopBackRight(LOCATIONS) }
    | "bottom" "front" "left" { s.BottomFrontLeft(LOCATIONS) }
    | "bottom" "front" "right" { s.BottomFrontRight(LOCATIONS) }
    | "bottom" "back" "left" { s.BottomBackLeft(LOCATIONS) }
    | "bottom" "back" "right" { s.BottomBackRight(LOCATIONS) }
    | "front" "left" { s.FrontLeft(LOCATIONS) }
    | "front" "right" { s.FrontRight(LOCATIONS) }
    | "back" "left" { s.BackLeft(LOCATIONS) }
```

(continues on next page)

(continued from previous page)

```
    | "back" "right" { s.BackRight(LOCATIONS) }
    | "front" { s.Front(LOCATIONS) }
    | "back" { s.Back(LOCATIONS) }
    | "left" { s.Left(LOCATIONS) }
    | "right" { s.Right(LOCATIONS) }
    | "top" { s.Top(LOCATIONS) }
    | "bottom" { s.Bottom(LOCATIONS) }

# Arithmetic operators
# --------------------

sum:
    | a=sum '+' b=term { ast.BinOp(left=a, op=ast.Add(), right=b, LOCATIONS) }
    | a=sum '-' b=term { ast.BinOp(left=a, op=ast.Sub(), right=b, LOCATIONS) }
    | term

term:
    | scenic_vector
    | scenic_deg
    | a=term '*' b=factor { ast.BinOp(left=a, op=ast.Mult(), right=b, LOCATIONS) }
    | a=term '/' b=factor { ast.BinOp(left=a, op=ast.Div(), right=b, LOCATIONS) }
    | a=term '//' b=factor { ast.BinOp(left=a, op=ast.FloorDiv(), right=b, LOCATIONS) }
    | a=term '%' b=factor { ast.BinOp(left=a, op=ast.Mod(), right=b, LOCATIONS) }
    | a=term '@' b=factor {
        self.check_version((3, 5), "The '@' operator is", ast.BinOp(left=a, op=ast.
↪MatMult(), right=b, LOCATIONS))
     }
    | factor

scenic_vector: a=term '@' b=factor { s.VectorOp(left=a, right=b, LOCATIONS) }
scenic_deg: a=term "deg" { s.DegOp(operand=a, LOCATIONS) }

factor (memo):
    | '+' a=factor { ast.UnaryOp(op=ast.UAdd(), operand=a, LOCATIONS) }
    | '-' a=factor { ast.UnaryOp(op=ast.USub(), operand=a, LOCATIONS) }
    | '~' a=factor { ast.UnaryOp(op=ast.Invert(), operand=a, LOCATIONS) }
    | power

power:
    | a=await_primary '**' b=factor { ast.BinOp(left=a, op=ast.Pow(), right=b,␣
↪LOCATIONS) }
    | scenic_new

scenic_new:
    | scenic_new_expr
    | await_primary

# Primary elements
# ----------------

# Primary elements are things like "obj.something.something", "obj[something]",
↪"obj(something)", "obj" ...
```

(continues on next page)

(continued from previous page)

```
await_primary (memo):
    | 'await' a=primary { self.check_version((3, 5), "Await expressions are", ast.
↪Await(a, LOCATIONS)) }
    | primary

primary:
    | a=primary '.' b=NAME { ast.Attribute(value=a, attr=b.string, ctx=Load, LOCATIONS) }
    | a=primary b=genexp { ast.Call(func=a, args=[b], keywords=[], LOCATIONS) }
    | a=primary '(' b=[arguments] ')' {
        ast.Call(
            func=a,
            args=b[0] if b else [],
            keywords=b[1] if b else [],
            LOCATIONS,
        )
     }
    | a=primary '[' b=slices ']' { ast.Subscript(value=a, slice=b, ctx=Load, LOCATIONS) }
    | atom

slices:
    | a=slice !',' { a }
    | a=','.slice+ [','] {
        ast.Tuple(elts=a, ctx=Load, LOCATIONS)
        if sys.version_info >= (3, 9) else
        (
            ast.ExtSlice(dims=a, LOCATIONS)
            if any(isinstance(e, ast.Slice) for e in a) else
            ast.Index(value=ast.Tuple(elts=[e.value for e in a], ctx=Load, LOCATIONS),␣
↪LOCATIONS)
        )
     }

slice:
    | a=[expression] ':' b=[expression] c=[':' d=[expression] { d }] {
        ast.Slice(lower=a, upper=b, step=c, LOCATIONS)
     }
    | a=named_expression {
        a
        if sys.version_info >= (3, 9) or isinstance(a, ast.Slice) else
        ast.Index(
            value=a,
            lineno=a.lineno,
            col_offset=a.col_offset,
            end_lineno=a.end_lineno,
            end_col_offset=a.end_col_offset
        )
     }

atom:
    | "initial" "scenario" { s.InitialScenario(LOCATIONS) }
    | a=NAME { ast.Name(id=a.string, ctx=Load, LOCATIONS) }
```

(continues on next page)

(continued from previous page)

```
    | 'True' {
        ast.Constant(value=True, LOCATIONS)
        if sys.version_info >= (3, 9) else
        ast.Constant(value=True, kind=None, LOCATIONS)
     }
    | 'False' {
        ast.Constant(value=False, LOCATIONS)
        if sys.version_info >= (3, 9) else
        ast.Constant(value=False, kind=None, LOCATIONS)
     }
    | 'None' {
        ast.Constant(value=None, LOCATIONS)
        if sys.version_info >= (3, 9) else
        ast.Constant(value=None, kind=None, LOCATIONS)
     }
    | &STRING strings
    | a=NUMBER {
        ast.Constant(value=ast.literal_eval(a.string), LOCATIONS)
        if sys.version_info >= (3, 9) else
        ast.Constant(value=ast.literal_eval(a.string), kind=None, LOCATIONS)
     }
    | &'(' (tuple | group | genexp)
    | &'[' (list | listcomp)
    | &'{' (dict | set | dictcomp | setcomp)
    | '...' {
        ast.Constant(value=Ellipsis, LOCATIONS)
        if sys.version_info >= (3, 9) else
        ast.Constant(value=Ellipsis, kind=None, LOCATIONS)
     }

group:
    | '(' a=(yield_expr | named_expression) ')' { a }
    | invalid_group


# Lambda functions
# ----------------

lambdef:
    | 'lambda' a=[lambda_params] ':' b=expression {
        ast.Lambda(args=a or self.make_arguments(None, [], None, [], (None, [], None)),␣
↪body=b, LOCATIONS)
     }

lambda_params:
    | invalid_lambda_parameters
    | lambda_parameters

# lambda_parameters etc. duplicates parameters but without annotations
# or type comments, and if there's no comma after a parameter, we expect
# a colon, not a close parenthesis.  (For more, see parameters above.)
#
```

(continues on next page)

(continued from previous page)

```
lambda_parameters[ast.arguments]:
    | a=lambda_slash_no_default b=lambda_param_no_default* c=lambda_param_with_default*␣
↪d=[lambda_star_etc] {
        self.make_arguments(a, [], b, c, d)
    }
    | a=lambda_slash_with_default b=lambda_param_with_default* c=[lambda_star_etc] {
        self.make_arguments(None, a, None, b, c)
    }
    | a=lambda_param_no_default+ b=lambda_param_with_default* c=[lambda_star_etc] {
        self.make_arguments(None, [], a, b, c)
    }
    | a=lambda_param_with_default+ b=[lambda_star_etc] {
        self.make_arguments(None, [], None, a, b)
    }
    | a=lambda_star_etc { self.make_arguments(None, [], None, [], a) }

lambda_slash_no_default[List[Tuple[ast.arg, None]]]:
    | a=lambda_param_no_default+ '/' ',' { [(p, None) for p in a] }
    | a=lambda_param_no_default+ '/' &':' { [(p, None) for p in a] }

lambda_slash_with_default[List[Tuple[ast.arg, Any]]]:
    | a=lambda_param_no_default* b=lambda_param_with_default+ '/' ',' { ([(p, None) for␣
↪p in a] if a else []) + b }
    | a=lambda_param_no_default* b=lambda_param_with_default+ '/' &':' { ([(p, None) for␣
↪p in a] if a else []) + b }

lambda_star_etc[Tuple[Optional[ast.arg], List[Tuple[ast.arg, Any]], Optional[ast.arg]]]:
    | invalid_lambda_star_etc
    | '*' a=lambda_param_no_default b=lambda_param_maybe_default* c=[lambda_kwds] {
      (a, b, c) }
    | '*' ',' b=lambda_param_maybe_default+ c=[lambda_kwds] {
      (None, b, c) }
    | a=lambda_kwds { (None, [], a) }

lambda_kwds[ast.arg]:
    | invalid_lambda_kwds
    | '**' a=lambda_param_no_default { a }

lambda_param_no_default[ast.arg]:
    | a=lambda_param ',' { a }
    | a=lambda_param &':' { a }

lambda_param_with_default[Tuple[ast.arg, Any]]:
    | a=lambda_param c=default ',' { (a, c) }
    | a=lambda_param c=default &':' { (a, c) }
lambda_param_maybe_default[Tuple[ast.arg, Any]]:
    | a=lambda_param c=default? ',' { (a, c) }
    | a=lambda_param c=default? &':' { (a, c) }
lambda_param[ast.arg]: a=NAME {
    ast.arg(arg=a.string, annotation=None, LOCATIONS)
    if sys.version_info >= (3, 9) else
    ast.arg(arg=a.string, annotation=None, type_comment=None, LOCATIONS)
```

(continues on next page)

(continued from previous page)

```
}

# SCENIC STATEMENTS
# =================

scenic_model_stmt:
    | "model" a=dotted_name { s.Model(name=a, LOCATIONS) }

scenic_tracked_assignment:
    | a=scenic_tracked_name '=' b=expression { s.TrackedAssign(target=a, value=b,␣
↪LOCATIONS) }
scenic_tracked_name:
    | "ego" { s.Ego(LOCATIONS) }
    | "workspace" { s.Workspace(LOCATIONS) }

scenic_param_stmt:
    | "param" elts=','.scenic_param_stmt_param+) { s.Param(elts=elts, LOCATIONS) }
scenic_param_stmt_param: name=scenic_param_stmt_id '=' e=expression { s.parameter(name,␣
↪e, LOCATIONS) }
scenic_param_stmt_id:
    | a=NAME { a.string }
    | a=STRING { a.string[1:-1] } # strip quotes

scenic_require_stmt:
    | 'require' "monitor" e=expression n=['as' scenic_require_stmt_name] {
        s.RequireMonitor(monitor=e, name=n, LOCATIONS)
     }
    | invalid_scenic_require_prob
    | 'require' p=['[' a=NUMBER ']' { float(a.string) }] e=scenic_temporal_expression n=[
↪'as' a=scenic_require_stmt_name { a }] {
        s.Require(cond=e, prob=p, name=n, LOCATIONS)
     }
scenic_require_stmt_name:
    | a=(NAME | NUMBER) { a.string }
    | a=STRING { a.string[1:-1] }

scenic_record_stmt:
    | "record" e=expression n=['as' a=scenic_require_stmt_name { a }] {
        s.Record(value=e, name=n, LOCATIONS)
     }

scenic_record_initial_stmt:
    | "record" "initial" e=expression n=['as' a=scenic_require_stmt_name { a }] {
        s.RecordInitial(value=e, name=n, LOCATIONS)
     }

scenic_record_final_stmt:
    | "record" "final" e=expression n=['as' a=scenic_require_stmt_name { a }] {
        s.RecordFinal(value=e, name=n, LOCATIONS)
     }

scenic_mutate_stmt:
```

(continues on next page)

(continued from previous page)

```
    | "mutate" elts=[(','.scenic_mutate_stmt_id+)] scale=['by' x=expression {x}] {
        s.Mutate(elts=elts if elts is not None else [], scale=scale, LOCATIONS)
     }
scenic_mutate_stmt_id: a=NAME { ast.Name(id=a.string, ctx=Load, LOCATIONS) }

scenic_abort_stmt: "abort" { s.Abort(LOCATIONS) }

scenic_take_stmt: "take" elts=(','.expression+) { s.Take(elts=elts, LOCATIONS) }

scenic_wait_stmt: "wait" { s.Wait(LOCATIONS) }

scenic_terminate_simulation_when_stmt: "terminate" "simulation" "when" v=expression n=[
↪'as' a=scenic_require_stmt_name { a }] { s.TerminateSimulationWhen(v, name=n,␣
↪LOCATIONS) }

scenic_terminate_when_stmt: "terminate" "when" v=expression n=['as' a=scenic_require_
↪stmt_name { a }] { s.TerminateWhen(v, name=n, LOCATIONS) }

scenic_terminate_after_stmt: "terminate" "after" v=scenic_dynamic_duration { s.
↪TerminateAfter(v, LOCATIONS) }

scenic_terminate_simulation_stmt: "terminate" "simulation" { s.
↪TerminateSimulation(LOCATIONS) }

scenic_terminate_stmt: "terminate" { s.Terminate(LOCATIONS) }

scenic_do_choose_stmt: 'do' "choose" e=(','.expression+) { s.DoChoose(e, LOCATIONS) }

scenic_do_shuffle_stmt: 'do' "shuffle" e=(','.expression+) { s.DoShuffle(e, LOCATIONS) }

scenic_do_for_stmt: 'do' e=(','.expression+) 'for' u=scenic_dynamic_duration { s.
↪DoFor(elts=e, duration=u, LOCATIONS) }
scenic_dynamic_duration:
    | v=expression "seconds" { s.Seconds(v, LOCATIONS) }
    | v=expression "steps" { s.Steps(v, LOCATIONS) }
    | invalid_scenic_dynamic_duration

# FIXME: Is this the right way to resolve ambiguity in `do A until B until X`?
scenic_do_until_stmt: 'do' e=(','.disjunction+) 'until' cond=expression { s.
↪DoUntil(elts=e, cond=cond, LOCATIONS) }

scenic_do_stmt: 'do' e=(','.expression+) { s.Do(elts=e, LOCATIONS) }

scenic_simulator_stmt: "simulator" e=expression { s.Simulator(value=e, LOCATIONS) }

# LITERALS
# ========

strings[ast.Str] (memo): a=STRING+ { self.generate_ast_for_string(a) }

list[ast.List]:
    | '[' a=[star_named_expressions] ']' { ast.List(elts=a or [], ctx=Load, LOCATIONS) }
```

(continues on next page)

(continued from previous page)

```
tuple[ast.Tuple]:
    | '(' a=[y=star_named_expression ',' z=[star_named_expressions] { [y] + (z or []) }␣
↪] ')' {
        ast.Tuple(elts=a or [], ctx=Load, LOCATIONS)
    }

set[ast.Set]: '{' a=star_named_expressions '}' { ast.Set(elts=a, LOCATIONS) }

# Dicts
# -----

dict[ast.Dict]:
    | '{' a=[double_starred_kvpairs] '}' {
        ast.Dict(keys=[kv[0] for kv in (a or [])], values=[kv[1] for kv in (a or [])],␣
↪LOCATIONS)
    }
    | '{' invalid_double_starred_kvpairs '}'

double_starred_kvpairs[list]: a=','.double_starred_kvpair+ [','] { a }

double_starred_kvpair:
    | '**' a=bitwise_or { (None, a) }
    | kvpair

kvpair[tuple]: a=expression ':' b=expression { (a, b) }

# Comprehensions & Generators
# ---------------------------

for_if_clauses[List[ast.comprehension]]:
    | a=for_if_clause+ { a }

for_if_clause[ast.comprehension]:
    | 'async' 'for' a=star_targets 'in' ~ b=disjunction c=('if' z=disjunction { z })* {
       self.check_version(
           (3, 6),
           "Async comprehensions are",
           ast.comprehension(target=a, iter=b, ifs=c, is_async=1)
       )
    }
    | 'for' a=star_targets 'in' ~ b=disjunction c=('if' z=disjunction { z })* {
      ast.comprehension(target=a, iter=b, ifs=c, is_async=0) }
    | invalid_for_target

listcomp[ast.ListComp]:
    | '[' a=named_expression b=for_if_clauses ']' { ast.ListComp(elt=a, generators=b,␣
↪LOCATIONS) }
    | invalid_comprehension

setcomp[ast.SetComp]:
    | '{' a=named_expression b=for_if_clauses '}' { ast.SetComp(elt=a, generators=b,␣
```

(continues on next page)

(continued from previous page)

```
↪LOCATIONS) }
    | invalid_comprehension

genexp[ast.GeneratorExp]:
    | '(' a=( assignment_expression | expression !':=') b=for_if_clauses ')' {
       ast.GeneratorExp(elt=a, generators=b, LOCATIONS)
     }
    | invalid_comprehension

dictcomp[ast.DictComp]:
    | '{' a=kvpair b=for_if_clauses '}' { ast.DictComp(key=a[0], value=a[1],␣
↪generators=b, LOCATIONS) }
    | invalid_dict_comprehension

# FUNCTION CALL ARGUMENTS
# =======================

arguments[Tuple[list, list]] (memo):
    | a=args [','] &')' { a }
    | invalid_arguments

args[Tuple[list, list]]:
    | a=','.(starred_expression | ( assignment_expression | expression !':=') !'=')+ b=[
↪',' k=kwargs {k}] {
        (a + ([e for e in b if isinstance(e, ast.Starred)] if b else []),
         ([e for e in b if not isinstance(e, ast.Starred)] if b else [])
        )
     }
    | a=kwargs {
       ([e for e in a if isinstance(e, ast.Starred)],
        [e for e in a if not isinstance(e, ast.Starred)])
    }

kwargs[list]:
    | a=','.kwarg_or_starred+ ',' b=','.kwarg_or_double_starred+ { a + b }
    | ','.kwarg_or_starred+
    | ','.kwarg_or_double_starred+

starred_expression:
    | '*' a=expression { ast.Starred(value=a, ctx=Load, LOCATIONS) }

kwarg_or_starred:
    | invalid_kwarg
    | a=NAME '=' b=expression { ast.keyword(arg=a.string, value=b, LOCATIONS) }
    | a=starred_expression { a }

kwarg_or_double_starred:
    | invalid_kwarg
    | a=NAME '=' b=expression { ast.keyword(arg=a.string, value=b, LOCATIONS) }   # XXX␣
↪Unreachable
    | '**' a=expression { ast.keyword(arg=None, value=a, LOCATIONS) }
```

(continues on next page)

(continued from previous page)

```
# ASSIGNMENT TARGETS
# ==================

# Generic targets
# ---------------

# NOTE: star_targets may contain *bitwise_or, targets may not.
star_targets:
    | a=star_target !',' { a }
    | a=star_target b=(',' c=star_target { c })* [','] {
        ast.Tuple(elts=[a] + b, ctx=Store, LOCATIONS)
     }

star_targets_list_seq[list]: a=','.star_target+ [','] { a }

star_targets_tuple_seq[list]:
    | a=star_target b=(',' c=star_target { c })+ [','] { [a] + b }
    | a=star_target ',' { [a] }

star_target (memo):
    | '*' a=(!'*' star_target) {
        ast.Starred(value=self.set_expr_context(a, Store), ctx=Store, LOCATIONS)
     }
    | target_with_star_atom

target_with_star_atom (memo):
    | a=t_primary '.' b=NAME !t_lookahead { ast.Attribute(value=a, attr=b.string,␣
↪ctx=Store, LOCATIONS) }
    | a=t_primary '[' b=slices ']' !t_lookahead { ast.Subscript(value=a, slice=b,␣
↪ctx=Store, LOCATIONS) }
    | star_atom

star_atom:
    | a=NAME { ast.Name(id=a.string, ctx=Store, LOCATIONS) }
    | '(' a=target_with_star_atom ')' { self.set_expr_context(a, Store) }
    | '(' a=[star_targets_tuple_seq] ')' { ast.Tuple(elts=a, ctx=Store, LOCATIONS) }
    | '[' a=[star_targets_list_seq] ']' {  ast.List(elts=a, ctx=Store, LOCATIONS) }

single_target:
    | single_subscript_attribute_target
    | a=NAME { ast.Name(id=a.string, ctx=Store, LOCATIONS) }
    | '(' a=single_target ')' { a }

single_subscript_attribute_target:
    | a=t_primary '.' b=NAME !t_lookahead { ast.Attribute(value=a, attr=b.string,␣
↪ctx=Store, LOCATIONS) }
    | a=t_primary '[' b=slices ']' !t_lookahead { ast.Subscript(value=a, slice=b,␣
↪ctx=Store, LOCATIONS) }


t_primary:
    | a=t_primary '.' b=NAME &t_lookahead { ast.Attribute(value=a, attr=b.string,␣
```

(continues on next page)

(continued from previous page)

```
↪ctx=Load, LOCATIONS) }
    | a=t_primary '[' b=slices ']' &t_lookahead { ast.Subscript(value=a, slice=b,␣
↪ctx=Load, LOCATIONS) }
    | a=t_primary b=genexp &t_lookahead { ast.Call(func=a, args=[b], keywords=[],␣
↪LOCATIONS) }
    | a=t_primary '(' b=[arguments] ')' &t_lookahead {
       ast.Call(
           func=a,
           args=b[0] if b else [],
           keywords=b[1] if b else [],
           LOCATIONS,
       )
     }
    | a=atom &t_lookahead { a }

t_lookahead: '(' | '[' | '.'

# Targets for del statements
# --------------------------

del_targets: a=','.del_target+ [','] { a }

del_target (memo):
    | a=t_primary '.' b=NAME !t_lookahead { ast.Attribute(value=a, attr=b.string,␣
↪ctx=Del, LOCATIONS) }
    | a=t_primary '[' b=slices ']' !t_lookahead { ast.Subscript(value=a, slice=b,␣
↪ctx=Del, LOCATIONS) }
    | del_t_atom

del_t_atom:
    | a=NAME { ast.Name(id=a.string, ctx=Del, LOCATIONS) }
    | '(' a=del_target ')' { self.set_expr_context(a, Del) }
    | '(' a=[del_targets] ')' { ast.Tuple(elts=a, ctx=Del, LOCATIONS) }
    | '[' a=[del_targets] ']' { ast.List(elts=a, ctx=Del, LOCATIONS) }


# TYPING ELEMENTS
# ---------------

# type_expressions allow */** but ignore them
type_expressions[list]:
    | a=','.expression+ ',' '*' b=expression ',' '**' c=expression { a + [b, c] }
    | a=','.expression+ ',' '*' b=expression { a + [b] }
    | a=','.expression+ ',' '**' b=expression { a + [b] }
    | '*' a=expression ',' '**' b=expression { [a, b] }
    | '*' a=expression { [a] }
    | '**' a=expression { [a] }
    | a=','.expression+ {a}

func_type_comment:
    | NEWLINE t=TYPE_COMMENT &(NEWLINE INDENT) { t.string }  # Must be followed by␣
↪indented block
```

(continues on next page)

(continued from previous page)

```
    | invalid_double_type_comments
    | TYPE_COMMENT

# ========================= END OF THE GRAMMAR ===========================



# ========================= START OF INVALID RULES =======================

# From here on, there are rules for invalid syntax with specialised error messages
invalid_arguments[NoReturn]:
    | a=args ',' '*' {
        self.raise_syntax_error_known_location(
            "iterable argument unpacking follows keyword argument unpacking",
            a[1][-1] if a[1] else a[0][-1],
        )
     }
    | a=expression b=for_if_clauses ',' [args | expression for_if_clauses] {
        self.raise_syntax_error_known_range(
            "Generator expression must be parenthesized",
            a,
            (b[-1].ifs[-1] if b[-1].ifs else b[-1].iter)
        )
     }
    | a=NAME b='=' expression for_if_clauses {
        self.raise_syntax_error_known_range(
            "invalid syntax. Maybe you meant '==' or ':=' instead of '='?", a, b
        )
     }
    | a=args b=for_if_clauses {
        self.raise_syntax_error_known_range(
            "Generator expression must be parenthesized",
            a[0][-1],
            (b[-1].ifs[-1] if b[-1].ifs else b[-1].iter),
        ) if len(a[0]) > 1 else None
     }
    | args ',' a=expression b=for_if_clauses {
        self.raise_syntax_error_known_range(
            "Generator expression must be parenthesized",
            a,
            (b[-1].ifs[-1] if b[-1].ifs else b[-1].iter),
        )
     }
    | a=args ',' args {
        self.raise_syntax_error(
            "positional argument follows keyword argument unpacking"
            if a[1][-1].arg is None else
            "positional argument follows keyword argument",
        )
     }
invalid_kwarg[NoReturn]:
    | a=('True'|'False'|'None') b='=' {
```

(continues on next page)

(continued from previous page)

```
        self.raise_syntax_error_known_range(f"cannot assign to {a.string}", a, b)
    }
    | a=NAME b='=' expression for_if_clauses {
        self.raise_syntax_error_known_range(
            "invalid syntax. Maybe you meant '==' or ':=' instead of '='?", a, b
        )
    }
    | !(NAME '=') a=expression b='=' {
        self.raise_syntax_error_known_range(
            "expression cannot contain assignment, perhaps you meant \"==\"?", a, b,
        )
    }

invalid_scenic_instance_creation[NoReturn]:
    | n=NAME s=scenic_valid_specifier {
        self.raise_syntax_error_known_range("invalid syntax. Perhaps you forgot 'new'?",␣
↪n, s)
    }
invalid_scenic_specifier[NoReturn]:
    | n=NAME {
        self.raise_syntax_error_known_location("invalid specifier.", n)
    }

expression_without_invalid[ast.AST]:
    | a=disjunction 'if' b=disjunction 'else' c=expression { ast.IfExp(body=b, test=a,␣
↪orelse=c, LOCATIONS) }
    | disjunction
    | lambdef
invalid_legacy_expression:
    | a=NAME !'(' b=expression_without_invalid {
        self.raise_syntax_error_known_range(
            f"Missing parentheses in call to '{a.string}' . Did you mean {a.string}(...)?
↪", a, b,
        ) if a.string in ("exec", "print") else
        None
    }
invalid_expression[NoReturn]:
    # !(NAME STRING) is not matched so we don't show this error with some invalid string␣
↪prefixes like: kf"dsfsdf"
    # Soft keywords need to also be ignored because they can be parsed as NAME NAME
    # Soft keywords can follow a disjunction to support expressions like `3 steps`
    | !(NAME STRING | SOFT_KEYWORD) a=disjunction !SOFT_KEYWORD b=expression_without_
↪invalid {
        (
            self.raise_syntax_error_known_range("invalid syntax. Perhaps you forgot a␣
↪comma?", a, b)
            if not isinstance(a, ast.Name) or a.id not in ("print", "exec")
            else None
        )
    }
    | a=disjunction 'if' b=disjunction !('else'|':') {
        self.raise_syntax_error_known_range("expected 'else' after 'if' expression", a,␣
```

(continues on next page)

(continued from previous page)

```
↪b)
    }
invalid_named_expression[NoReturn]:
    | a=expression ':=' expression {
        self.raise_syntax_error_known_location(
            f"cannot use assignment expressions with {self.get_expr_name(a)}", a
        )
    }
    # Use in_raw_rule
    | a=NAME '=' b=bitwise_or !('='|':=') {
        (
            None
            if self.in_recursive_rule else
            self.raise_syntax_error_known_range(
                "invalid syntax. Maybe you meant '==' or ':=' instead of '='?", a, b
            )
        )
    }
    | !(list|tuple|genexp|'True'|'None'|'False') a=bitwise_or b='=' bitwise_or !('='|':=
↪') {
        (
            None
            if self.in_recursive_rule else
            self.raise_syntax_error_known_location(
                f"cannot assign to {self.get_expr_name(a)} here. Maybe you meant '=='␣
↪instead of '='?", a
            )
        )
    }

invalid_scenic_until[NoReturn]:
    | a=scenic_temporal_disjunction 'until' scenic_implication {
        self.raise_syntax_error_known_location(
            f"`until` must take exactly two operands", a
        )
    }

invalid_scenic_implication[NoReturn]:
    | a=scenic_until "implies" scenic_implication {
        self.raise_syntax_error_known_location(
            f"`implies` must take exactly two operands", a
        )
    }

invalid_scenic_require_prob[NoReturn]:
    | 'require' '[' !(NUMBER ']') p=expression ']' scenic_temporal_expression ['as'␣
↪scenic_require_stmt_name] {
        self.raise_syntax_error_known_location(
            f"'require' probability must be a constant", p
        )
    }
```

(continues on next page)

(continued from previous page)

```
invalid_scenic_dynamic_duration[NoReturn]: e=expression {
    self.raise_syntax_error_known_location(
        "duration must specify a unit (seconds or steps)", e
    )
}

invalid_assignment[NoReturn]:
    | a=invalid_ann_assign_target ':' expression {
        self.raise_syntax_error_known_location(
            f"only single target (not {self.get_expr_name(a)}) can be annotated", a
        )
     }
    | a=star_named_expression ',' star_named_expressions* ':' expression {
        self.raise_syntax_error_known_location("only single target (not tuple) can be␣
↪annotated", a) }
    | a=expression ':' expression {
        self.raise_syntax_error_known_location("illegal target for annotation", a) }
    | (star_targets '=')* a=star_expressions '=' {
        self.raise_syntax_error_invalid_target(Target.STAR_TARGETS, a)
     }
    | (star_targets '=')* a=yield_expr '=' {
        self.raise_syntax_error_known_location("assignment to yield expression not␣
↪possible", a)
     }
    | a=star_expressions augassign (yield_expr | star_expressions) {
        self.raise_syntax_error_known_location(
            f"'{self.get_expr_name(a)}' is an illegal expression for augmented assignment
↪", a
        )
     }
invalid_ann_assign_target[ast.AST]:
    | a=list { a }
    | a=tuple { a }
    | '(' a=invalid_ann_assign_target ')' { a }
invalid_del_stmt[NoReturn]:
    | 'del' a=star_expressions {
        self.raise_syntax_error_invalid_target(Target.DEL_TARGETS, a)
     }
invalid_block[NoReturn]:
    | NEWLINE !INDENT { self.raise_indentation_error("expected an indented block") }
invalid_comprehension[NoReturn]:
    | ('[' | '(' | '{') a=starred_expression for_if_clauses {
        self.raise_syntax_error_known_location("iterable unpacking cannot be used in␣
↪comprehension", a)
     }
    | ('[' | '{') a=star_named_expression ',' b=star_named_expressions for_if_clauses {
        self.raise_syntax_error_known_range(
            "did you forget parentheses around the comprehension target?", a, b[-1]
        )
     }
    | ('[' | '{') a=star_named_expression b=',' for_if_clauses {
        self.raise_syntax_error_known_range(
```

(continues on next page)

(continued from previous page)

```
            "did you forget parentheses around the comprehension target?", a, b
        )
    }
invalid_dict_comprehension[NoReturn]:
    | '{' a='**' bitwise_or for_if_clauses '}' {
        self.raise_syntax_error_known_location("dict unpacking cannot be used in dict␣
↪comprehension", a)
    }
invalid_parameters[NoReturn]:
    | param_no_default* invalid_parameters_helper a=param_no_default {
        self.raise_syntax_error_known_location("non-default argument follows default␣
↪argument", a)
    }
    | param_no_default* a='(' param_no_default+ ','? b=')' {
        self.raise_syntax_error_known_range("Function parameters cannot be parenthesized
↪", a, b)
    }
    | a="/" ',' {
        self.raise_syntax_error_known_location("at least one argument must precede /", a)
    }
    | (slash_no_default | slash_with_default) param_maybe_default* a='/' {
        self.raise_syntax_error_known_location("/ may appear only once", a)
    }
    | (slash_no_default | slash_with_default)? param_maybe_default* '*' (',' | param_no_
↪default) param_maybe_default* a='/' {
        self.raise_syntax_error_known_location("/ must be ahead of *", a)
    }
    | param_maybe_default+ '/' a='*' {
        self.raise_syntax_error_known_location("expected comma between / and *", a)
    }
invalid_default:
    | a='=' &(')'|',') {
        self.raise_syntax_error_known_location("expected default value expression", a)
    }
invalid_star_etc:
    | a='*' (')' | ',' (')' | '**')) {
        self.raise_syntax_error_known_location("named arguments must follow bare *", a)
    }
    | '*' ',' TYPE_COMMENT { self.raise_syntax_error("bare * has associated type comment
↪") }
    | '*' param a='=' {
        self.raise_syntax_error_known_location("var-positional argument cannot have␣
↪default value", a)
    }
    | '*' (param_no_default | ',') param_maybe_default* a='*' (param_no_default | ',') {
        self.raise_syntax_error_known_location("* argument may appear only once", a)
    }
invalid_kwds:
    | '**' param a='=' {
        self.raise_syntax_error_known_location("var-keyword argument cannot have default␣
↪value", a)
    }
```

(continues on next page)

(continued from previous page)

```
    | '**' param ',' a=param {
        self.raise_syntax_error_known_location("arguments cannot follow var-keyword␣
↪argument", a)
      }
    | '**' param ',' a=('*'|'**'|'/') {
        self.raise_syntax_error_known_location("arguments cannot follow var-keyword␣
↪argument", a)
      }
invalid_parameters_helper: # This is only there to avoid type errors
    | a=slash_with_default { [a] }
    | a=param_with_default+
invalid_lambda_parameters[NoReturn]:
    | lambda_param_no_default* invalid_lambda_parameters_helper a=lambda_param_no_
↪default {
        self.raise_syntax_error_known_location("non-default argument follows default␣
↪argument", a)
      }
    | lambda_param_no_default* a='(' ','.lambda_param+ ','? b=')' {
        self.raise_syntax_error_known_range("Lambda expression parameters cannot be␣
↪parenthesized", a, b)
      }
    | a="/" ',' {
        self.raise_syntax_error_known_location("at least one argument must precede /", a)
      }
    | (lambda_slash_no_default | lambda_slash_with_default) lambda_param_maybe_default*␣
↪a='/' {
        self.raise_syntax_error_known_location("/ may appear only once", a)
      }
    | (lambda_slash_no_default | lambda_slash_with_default)? lambda_param_maybe_default*
↪'*' (',' | lambda_param_no_default) lambda_param_maybe_default* a='/' {
        self.raise_syntax_error_known_location("/ must be ahead of *", a)
      }
    | lambda_param_maybe_default+ '/' a='*' {
        self.raise_syntax_error_known_location("expected comma between / and *", a)
      }
invalid_lambda_parameters_helper[NoReturn]:
    | a=lambda_slash_with_default { [a] }
    | a=lambda_param_with_default+
invalid_lambda_star_etc[NoReturn]:
    | '*' (':' | ',' (':' | '**')) {
        self.raise_syntax_error("named arguments must follow bare *")
      }
    | '*' lambda_param a='=' {
        self.raise_syntax_error_known_location("var-positional argument cannot have␣
↪default value", a)
      }
    | '*' (lambda_param_no_default | ',') lambda_param_maybe_default* a='*' (lambda_
↪param_no_default | ',') {
        self.raise_syntax_error_known_location("* argument may appear only once", a)
      }
invalid_lambda_kwds:
    | '**' lambda_param a='=' {
```

(continues on next page)

(continued from previous page)

```
        self.raise_syntax_error_known_location("var-keyword argument cannot have default␣
↪value", a)
    }
    | '**' lambda_param ',' a=lambda_param {
        self.raise_syntax_error_known_location("arguments cannot follow var-keyword␣
↪argument", a)
    }
    | '**' lambda_param ',' a=('*'|'**'|'/') {
        self.raise_syntax_error_known_location("arguments cannot follow var-keyword␣
↪argument", a)
    }
invalid_double_type_comments[NoReturn]:
    | TYPE_COMMENT NEWLINE TYPE_COMMENT NEWLINE INDENT {
        self.raise_syntax_error("Cannot have two type comments on def")
    }
invalid_with_item[NoReturn]:
    | expression 'as' a=expression &(',' | ')' | ':') {
        self.raise_syntax_error_invalid_target(Target.STAR_TARGETS, a)
    }

invalid_for_target[NoReturn]:
    | 'async'? 'for' a=star_expressions {
        self.raise_syntax_error_invalid_target(Target.FOR_TARGETS, a)
    }

invalid_group[NoReturn]:
    | '(' a=starred_expression ')' {
        self.raise_syntax_error_known_location("cannot use starred expression here", a)
    }
    | '(' a='**' expression ')' {
        self.raise_syntax_error_known_location("cannot use double starred expression here
↪", a)
    }
invalid_import_from_targets[NoReturn]:
    | import_from_as_names ',' NEWLINE {
        self.raise_syntax_error("trailing comma not allowed without surrounding␣
↪parentheses")
    }

invalid_with_stmt[None]:
    | ['async'] 'with' ','.(expression ['as' star_target])+ &&':' { UNREACHABLE }
    | ['async'] 'with' '(' ','.(expressions ['as' star_target])+ ','? ')' &&':' {␣
↪UNREACHABLE }
invalid_with_stmt_indent[NoReturn]:
    | ['async'] a='with' ','.(expression ['as' star_target])+ ':' NEWLINE !INDENT {
        self.raise_indentation_error(
            f"expected an indented block after 'with' statement on line {a.start[0]}"
        )
    }
    | ['async'] a='with' '(' ','.(expressions ['as' star_target])+ ','? ')' ':' NEWLINE !
↪INDENT {
        self.raise_indentation_error(
```

(continues on next page)

(continued from previous page)

```
            f"expected an indented block after 'with' statement on line {a.start[0]}"
        )
    }

invalid_try_stmt[NoReturn]:
    | a='try' ':' NEWLINE !INDENT {
        self.raise_indentation_error(
            f"expected an indented block after 'try' statement on line {a.start[0]}",
        )
    }
    | 'try' ':' block !('except' | 'finally') {
        self.raise_syntax_error("expected 'except' or 'finally' block")
    }
invalid_except_stmt[None]:
    | 'except' a=expression ',' expressions ['as' NAME ] ':' {
        self.raise_syntax_error_starting_from("multiple exception types must be␣
↪parenthesized", a)
    }
    | a='except' expression ['as' NAME ] NEWLINE { self.raise_syntax_error("expected ':'
↪") }
    | a='except' NEWLINE { self.raise_syntax_error("expected ':'") }
invalid_finally_stmt[NoReturn]:
    | a='finally' ':' NEWLINE !INDENT {
        self.raise_indentation_error(
            f"expected an indented block after 'finally' statement on line {a.start[0]}"
        )
    }
invalid_except_stmt_indent[NoReturn]:
    | a='except' expression ['as' NAME ] ':' NEWLINE !INDENT {
        self.raise_indentation_error(
            f"expected an indented block after 'except' statement on line {a.start[0]}"
        )
    }
    | a='except' ':' NEWLINE !INDENT {
        self.raise_indentation_error(
            f"expected an indented block after 'except' statement on line {a.start[0]}"
        )
    }
invalid_match_stmt[NoReturn]:
    | "match" subject_expr !':' {
        self.check_version(
            (3, 10),
            "Pattern matching is",
            self.raise_syntax_error("expected ':'")
        )
    }
    | a="match" subject=subject_expr ':' NEWLINE !INDENT {
        self.check_version(
            (3, 10),
            "Pattern matching is",
            self.raise_indentation_error(
                f"expected an indented block after 'match' statement on line {a.start[0]}
```

(continues on next page)

(continued from previous page)

```
↪"
                )
            )
        }
invalid_case_block[NoReturn]:
    | "case" patterns guard? !':' { self.raise_syntax_error("expected ':'") }
    | a="case" patterns guard? ':' NEWLINE !INDENT {
        self.raise_indentation_error(
            f"expected an indented block after 'case' statement on line {a.start[0]}"
        )
    }
invalid_as_pattern[NoReturn]:
    | or_pattern 'as' a="_" {
        self.raise_syntax_error_known_location("cannot use '_' as a target", a)
    }
    | or_pattern 'as' !NAME a=expression {
        self.raise_syntax_error_known_location("invalid pattern target", a)
    }
invalid_class_pattern[NoReturn]:
    | name_or_attr '(' a=invalid_class_argument_pattern  {
        self.raise_syntax_error_known_range(
            "positional patterns follow keyword patterns", a[0], a[-1]
        )
    }
invalid_class_argument_pattern[list]:
    | [positional_patterns ','] keyword_patterns ',' a=positional_patterns { a }
invalid_if_stmt[NoReturn]:
    | 'if' named_expression NEWLINE { self.raise_syntax_error("expected ':'") }
    | a='if' a=named_expression ':' NEWLINE !INDENT {
        self.raise_indentation_error(
            f"expected an indented block after 'if' statement on line {a.start[0]}"
        )
    }
invalid_elif_stmt[NoReturn]:
    | 'elif' named_expression NEWLINE { self.raise_syntax_error("expected ':'") }
    | a='elif' named_expression ':' NEWLINE !INDENT {
        self.raise_indentation_error(
            f"expected an indented block after 'elif' statement on line {a.start[0]}"
        )
    }
invalid_else_stmt[NoReturn]:
    | a='else' ':' NEWLINE !INDENT {
        self.raise_indentation_error(
            f"expected an indented block after 'else' statement on line {a.start[0]}"
        )
    }
invalid_while_stmt[NoReturn]:
    | 'while' named_expression NEWLINE { self.raise_syntax_error("expected ':'") }
    | a='while' named_expression ':' NEWLINE !INDENT {
        self.raise_indentation_error(
            f"expected an indented block after 'while' statement on line {a.start[0]}"
        )
```

(continues on next page)

(continued from previous page)

```
    }
invalid_for_stmt[NoReturn]:
    | ['async'] a='for' star_targets 'in' star_expressions ':' NEWLINE !INDENT {
        self.raise_indentation_error(
            f"expected an indented block after 'for' statement on line {a.start[0]}"
        )
    }
invalid_def_raw[NoReturn]:
    | ['async'] a='def' NAME '(' [params] ')' ['->' expression] ':' NEWLINE !INDENT {
        self.raise_indentation_error(
            f"expected an indented block after function definition on line {a.start[0]}"
        )
    }
invalid_class_def_raw[NoReturn]:
    | a='class' NAME ['(' [arguments] ')'] ':' NEWLINE !INDENT {
        self.raise_indentation_error(
            f"expected an indented block after class definition on line {a.start[0]}"
        )
    }

invalid_double_starred_kvpairs[None]:
    | ','.double_starred_kvpair+ ',' invalid_kvpair
    | expression ':' a='*' bitwise_or {
        self.raise_syntax_error_starting_from("cannot use a starred expression in a␣
↪dictionary value", a)
    }
    | expression a=':' &('}'|',') {
        self.raise_syntax_error_known_location("expression expected after dictionary key␣
↪and ':'", a)
    }
invalid_kvpair[None]:
    | a=expression !(':') {
        self.raise_raw_syntax_error(
            "':' expected after dictionary key",
            (a.lineno, a.col_offset),
            (a.end_lineno, a.end_col_offset)
        )
    }
    | expression ':' a='*' bitwise_or {
        self.raise_syntax_error_starting_from("cannot use a starred expression in a␣
↪dictionary value", a)
    }
    | expression a=':' {
        self.raise_syntax_error_known_location("expression expected after dictionary key␣
↪and ':'", a)
    }
```

1.12.4 Scenic Modules

Detailed documentation on Scenic's components is organized by the submodules of the main `scenic` module:

scenic.core	Scenic's core types and associated support code.
scenic.domains	General scenario domains used across simulators.
scenic.formats	Support for file formats not specific to particular simulators.
scenic.simulators	World models and interfaces for particular simulators.
scenic.syntax	The Scenic compiler and associated support code.

scenic.core

Scenic's core types and associated support code.

distributions	Objects representing distributions that can be sampled from.
dynamics	Support for dynamic behaviors and modular scenarios.
errors	Common exceptions and error handling.
external_params	Support for values which are sampled outside of Scenic.
geometry	Utility functions for geometric computation.
lazy_eval	Support for lazy evaluation of expressions and specifiers.
object_types	Implementations of the built-in Scenic classes.
propositions	Objects representing propositions that can be used to specify conditions
pruning	Pruning parts of the sample space which violate requirements.
regions	Objects representing regions in space.
requirements	Support for hard and soft requirements.
sample_checking	The SampleChecker class and it's implementations.
scenarios	Scenario and scene objects.
serialization	Utilities to help serialize Scenic objects.
shapes	Module containing the Shape class and its subclasses, which represent shapes of Objects
simulators	Interface between Scenic and simulators.
specifiers	Specifiers and associated objects.
type_support	Support for checking Scenic types.
utils	Assorted utility functions.
vectors	Scenic vectors and vector fields.
visibility	Implementations of Scenic's visibility functions.
workspaces	Workspaces.

scenic.core.distributions

Objects representing distributions that can be sampled from.

Summary of Module Members

Functions

Uniform	Uniform distribution over a finite list of options.
addSupports	
canUnpackDistributions	Whether the function supports iterable unpacking of distributions.
distributionFunction	Decorator for wrapping a function so that it can take distributions as arguments.
distributionMethod	Decorator for wrapping a method so that it can take distributions as arguments.
makeOperatorHandler	
monotonicDistributionFunction	Like distributionFunction, but additionally specifies that the function is monotonic.
supmax	
supmin	
supportInterval	Lower and upper bounds on this value, if known.
toDistribution	Wrap Python data types with Distributions, if necessary.
underlyingFunction	Original function underlying a distribution wrapper.
unionOfSupports	
unpacksDistributions	Decorator indicating the function supports iterable unpacking of distributions.

Classes

AttributeDistribution	Distribution resulting from accessing an attribute of a distribution
ConstantSamplable	A samplable which always evaluates to a constant value.
DiscreteRange	Distribution over a range of integers.
Distribution	Abstract class for distributions.
FunctionDistribution	Distribution resulting from passing distributions to a function
MethodDistribution	Distribution resulting from passing distributions to a method of a fixed object
MultiplexerDistribution	Distribution selecting among values based on another distribution.
Normal	Normal distribution
OperatorDistribution	Distribution resulting from applying an operator to one or more distributions
Options	Distribution over a finite list of options.
Range	Uniform distribution over a range
Samplable	Abstract class for values which can be sampled, possibly depending on other values.
SliceDistribution	Distributions over `slice` objects.
StarredDistribution	A placeholder for the iterable unpacking operator * applied to a distribution.
TruncatedNormal	Truncated normal distribution.
TupleDistribution	Distributions over tuples (or namedtuples, or lists).
UniformDistribution	Uniform distribution over a variable number of options.

Exceptions

RandomControlFlowError	Exception indicating illegal conditional control flow depending on a random value.
RejectionException	Exception used to signal that the sample currently being generated must be rejected.

Member Details

supportInterval(*thing*)

Lower and upper bounds on this value, if known.

underlyingFunction(*thing*)

Original function underlying a distribution wrapper.

canUnpackDistributions(*func*)

Whether the function supports iterable unpacking of distributions.

unpacksDistributions(*func*)

Decorator indicating the function supports iterable unpacking of distributions.

exception RejectionException

Bases: `Exception`

Exception used to signal that the sample currently being generated must be rejected.

exception RandomControlFlowError

Bases: *ScenicError*

Exception indicating illegal conditional control flow depending on a random value.

This includes trying to iterate over a random value, making a range of random length, etc.

class Samplable(*dependencies*)

Bases: *LazilyEvaluable*

Abstract class for values which can be sampled, possibly depending on other values.

Samplables may specify a proxy object which must have the same distribution as the original after conditioning on the scenario's requirements. This allows transparent conditioning without modifying Samplable fields of immutable objects.

Parameters

dependencies – sequence of values that this value may depend on (formally, objects for which sampled values must be provided to *sampleGiven*). It is legal to include values which are not instances of *Samplable*, e.g. integers.

Attributes

- **_conditioned** – proxy object as described above; set using *conditionTo*.
- **_dependencies** – tuple of other samplables which must be sampled before this one; set by the initializer and subsequently immutable.

static sampleAll(*quantities*)

Sample all the given Samplables, which may have dependencies in common.

Reproducibility note: the order in which the quantities are given can affect the order in which calls to random are made, affecting the final result.

sample(*subsamples=None*)

Sample this value, optionally given some values already sampled.

sampleGiven(*value*)

Sample this value, given values for all its dependencies.

Implemented by subclasses.

Parameters

value (DefaultIdentityDict) – dictionary mapping objects to their sampled values. Guaranteed to provide values for all objects given in the set of dependencies when this *Samplable* was created.

conditionTo(*value*)

Condition this value to another value with the same conditional distribution.

evaluateIn(*context*)

See *LazilyEvaluable.evaluateIn*.

class ConstantSamplable(*value*)

Bases: *Samplable*

A samplable which always evaluates to a constant value.

Only for internal use.

class Distribution(*dependencies, valueType=None)

Bases: Samplable

Abstract class for distributions.

Note: When called during dynamic simulations (vs. scenario compilation), constructors for distributions return *actual sampled values*, not Distribution objects.

Parameters

- **dependencies** – values which this distribution may depend on (see Samplable).
- **valueType** – **_valueType** to use (see below), or None for the default.

Attributes

_valueType – type of the values sampled from this distribution, or *Object* if the type is not known.

_defaultValueType

Default valueType for distributions of this class, when not otherwise specified.

alias of object

_deterministic = False

Whether this type of distribution is a deterministic function of its dependencies.

For example, Options is implemented as deterministic by using an internal DiscreteRange to select which of its finitely-many options to choose from: the value of the Options is then completely determined by the value of the range and the values of each of the options. This simplifies serialization because these dependencies likely have simpler valueTypes than the Options itself (e.g. if we had a random choice between a list and a string, encoding the actual sampled value would require saving type information).

clone()

Construct an independent copy of this Distribution.

Optionally implemented by subclasses.

property isPrimitive

Whether this is a primitive Distribution.

serializeValue(*values*, *serializer*)

Serialize the sampled value of this distribution.

This method is used internally by Scenario.sceneToBytes and related APIs. If you define a new subclass of Distribution, you probably don't need to override this method. If your distribution has an unusual **valueType** (i.e. not float, int, or Vector), see the documentation for Serializer for instructions on how to support serialization.

bucket(*buckets=None*)

Construct a bucketed approximation of this Distribution.

Optionally implemented by subclasses.

This function factors a given Distribution into a discrete distribution over buckets together with a distribution for each bucket. The argument *buckets* controls how many buckets the domain of the original Distribution is split into. Since the result is an independent distribution, the original must support clone.

supportInterval()

Compute lower and upper bounds on the value of this Distribution.

By default returns `(None, None)` indicating that no lower or upper bounds are known. Subclasses may override this method to provide more accurate results.

class TupleDistribution(**coordinates*, *builder=<class 'tuple'>*)

Bases: *Distribution*, Sequence

Distributions over tuples (or namedtuples, or lists).

class SliceDistribution(*start*, *stop*, *step*)

Bases: *Distribution*

Distributions over `slice` objects.

toDistribution(*val*)

Wrap Python data types with Distributions, if necessary.

For example, tuples containing Samplables need to be converted into TupleDistributions in order to keep track of dependencies properly.

class FunctionDistribution(*func*, *args*, *kwargs*, *support=None*, *valueType=None*)

Bases: *Distribution*

Distribution resulting from passing distributions to a function

distributionFunction(*wrapped=None*, ***, *support=None*, *valueType=None*)

Decorator for wrapping a function so that it can take distributions as arguments.

This decorator is mainly for internal use, and is not necessary when defining a function in a Scenic file. It is, however, needed when calling external functions which contain control flow or other operations that Scenic distribution objects (representing random values) do not support.

monotonicDistributionFunction(*method*, *valueType=None*)

Like distributionFunction, but additionally specifies that the function is monotonic.

class StarredDistribution(*value*, *lineno*)

Bases: *Distribution*

A placeholder for the iterable unpacking operator * applied to a distribution.

class MethodDistribution(*method*, *obj*, *args*, *kwargs*, *valueType=None*)

Bases: *Distribution*

Distribution resulting from passing distributions to a method of a fixed object

distributionMethod(*method=None*, ***, *identity=None*)

Decorator for wrapping a method so that it can take distributions as arguments.

class AttributeDistribution(*attribute*, *obj*, *valueType=None*)

Bases: *Distribution*

Distribution resulting from accessing an attribute of a distribution

static inferType(*ty*, *attribute*)

Attempt to infer the type of the given attribute.

class OperatorDistribution(*operator*, *obj*, *operands*, *valueType=None*)

Bases: *Distribution*

Distribution resulting from applying an operator to one or more distributions

static inferType(*ty*, *operator*, *operands*)

Attempt to infer the result type of the given operator application.

class MultiplexerDistribution(*index*, *options*)

Bases: *Distribution*

Distribution selecting among values based on another distribution.

class Range(*low*, *high*)

Bases: *Distribution*

Uniform distribution over a range

class Normal(*mean*, *stddev*)

Bases: *Distribution*

Normal distribution

class TruncatedNormal(*mean*, *stddev*, *low*, *high*)

Bases: *Normal*

Truncated normal distribution.

class DiscreteRange(*low*, *high*, *weights=None*, *emptyMessage='empty DiscreteRange'*)

Bases: *Distribution*

Distribution over a range of integers.

class Options(*opts*)

Bases: *MultiplexerDistribution*

Distribution over a finite list of options.

Specified by a dict giving probabilities; otherwise uniform over a given iterable.

Uniform(**opts*)

Uniform distribution over a finite list of options.

Implemented as an instance of *Options* when the set of options is known statically, and an instance of *UniformDistribution* otherwise.

class UniformDistribution(*opts*)

Bases: *Distribution*

Uniform distribution over a variable number of options.

See *Options* for the more common uniform distribution over a fixed number of options. This class is for the special case where iterable unpacking is applied to a distribution, so that the number of options is unknown at compile time.

scenic.core.dynamics

Support for dynamic behaviors and modular scenarios.

Summary of Module Members

Module Attributes

stuckBehaviorWarningTimeout	Timeout in seconds after which a *StuckBehaviorWarning* will be raised.

Functions

runTryInterrupt

Classes

Behavior	Dynamic behaviors of agents.
BlockConclusion	An enumeration.
DynamicScenario	Internal class for scenarios which can execute during dynamic simulations.
InterruptBlock	
Invocable	Abstract class with common code for behaviors and modular scenarios.
Monitor	Monitors for dynamic simulations.

Exceptions

GuardViolation	Abstract exception raised when a guard of a behavior is violated.
InvariantViolation	Exception raised when an invariant is violated
PreconditionViolation	Exception raised when a precondition is violated
StuckBehaviorWarning	Warning issued when a behavior/scenario may have gotten stuck.

Member Details

exception StuckBehaviorWarning

Bases: UserWarning

Warning issued when a behavior/scenario may have gotten stuck.

When a behavior or compose block of a modular scenario executes for a long time without yielding control, there is no way to tell whether it has entered an infinite loop with no take/wait statements, or is actually doing some long computation. But since forgetting a wait statement in a wait loop is an easy mistake, we raise this warning after a behavior/scenario has run for *stuckBehaviorWarningTimeout* seconds without yielding.

stuckBehaviorWarningTimeout = 10

Timeout in seconds after which a *StuckBehaviorWarning* will be raised.

class Invocable(*args, **kwargs)

Abstract class with common code for behaviors and modular scenarios.

Both of these types of objects can be called like functions, can have guards, and can suspend their own execution to invoke sub-behaviors/scenarios.

_invokeInner(*agent*, *subs*)

Run the given sub-behavior/scenario(s) in parallel.

Implemented by subclasses.

class DynamicScenario(*args, **kwargs)

Bases: *Invocable*

Internal class for scenarios which can execute during dynamic simulations.

Provides additional information complementing *Scenario*, which originally only supported static scenarios. The two classes should probably eventually be merged.

classmethod _requiresArguments()

Whether this scenario cannot be instantiated without arguments.

_bindTo(*scene*)

Bind this scenario to a sampled scene when starting a new simulation.

_prepare(*delayPreconditionCheck=False*)

Prepare the scenario for execution, executing its setup block.

_start()

Start the scenario, starting its compose block, behaviors, and monitors.

_step()

Execute the (already-started) scenario for one time step.

Returns

None if the scenario will continue executing; otherwise a string describing why it has terminated.

_stop(*reason*, *quiet=False*)

Stop the scenario's execution, for the given reason.

_addRequirement(*ty*, *reqID*, *req*, *line*, *name*, *prob*)

Save a requirement defined at compile-time for later processing.

_addDynamicRequirement(*ty*, *req*, *line*, *name*)

Add a requirement defined during a dynamic simulation.

_addMonitor(*monitor*)

Add a monitor during a dynamic simulation.

class Behavior(*args, **kwargs)

Bases: *Invocable*, *Samplable*

Dynamic behaviors of agents.

Behavior statements are translated into definitions of subclasses of this class.

class Monitor(**args*, ***kwargs*)

Bases: *Behavior*

Monitors for dynamic simulations.

Monitor statements are translated into definitions of subclasses of this class.

exception GuardViolation(*behavior*, *lineno*)

Bases: Exception

Abstract exception raised when a guard of a behavior is violated.

This will never be raised directly; either of the subclasses *PreconditionViolation* or *InvariantViolation* will be used, as appropriate.

exception PreconditionViolation(*behavior*, *lineno*)

Bases: *GuardViolation*

Exception raised when a precondition is violated

Raised when a precondition is violated when invoking a behavior or when a precondition encounters a *RejectionException*, so that rejections count as precondition violations.

exception InvariantViolation(*behavior*, *lineno*)

Bases: *GuardViolation*

Exception raised when an invariant is violated

Raised when an invariant is violated when invoking/resuming a behavior or when an invariant encounters a *RejectionException*, so that rejections count as invariant violations.

class BlockConclusion(*value*)

Bases: Enum

An enumeration.

scenic.core.errors

Common exceptions and error handling.

Summary of Module Members

Module Attributes

verbosityLevel	Verbosity level.
showInternalBacktrace	Whether or not to include Scenic's innards in backtraces.
postMortemDebugging	Whether or not to do post-mortem debugging of uncaught exceptions.
postMortemRejections	Whether or not to do "post-mortem" debugging of rejected scenes/simulations.
hiddenFolders	Folders elided from backtraces when *showInternalBacktrace* is false.

Functions

callBeginningScenicTrace	Call the given function, starting the Scenic backtrace at that point.
displayScenicException	Print a Scenic exception, cleaning up the traceback if desired.
excepthook	
getText	Attempt to recover the text of an error from the original Scenic file.
includeFrame	
optionallyDebugRejection	
saveErrorLocation	
setDebuggingOptions	Configure Scenic's debugging options.

Exceptions

ASTParseError	Parse error occuring during modification of the Python AST.
InconsistentScenarioError	Error for scenarios with inconsistent requirements.
InvalidScenarioError	Error raised for syntactically-valid but otherwise problematic Scenic programs.
ParseCompileError	Error occurring during Scenic/Python parsing or compilation.
PythonCompileError	Error occuring during Python compilation of translated Scenic code.
ScenicError	An error produced during Scenic compilation, scene generation, or simulation.
ScenicParseError	Error occuring during Scenic parsing or compilation.
ScenicSyntaxError	An error produced by attempting to parse an invalid Scenic program.
SpecifierError	Error for illegal uses of specifiers.

Member Details

setDebuggingOptions(**, verbosity=0, fullBacktrace=False, debugExceptions=False, debugRejections=False*)

Configure Scenic's debugging options.

Parameters

- **verbosity** (*int*) – Verbosity level. Zero by default, although the command-line interface uses 1 by default. See the *--verbosity* option for the allowed values.
- **fullBacktrace** (*bool*) – Whether to include Scenic's innards in backtraces (like the *-b* command-line option).
- **debugExceptions** (*bool*) – Whether to use pdb for post-mortem debugging of uncaught exceptions (like the *--pdb* option).

- **debugRejections** (*bool*) – Whether to enter pdb when a scene or simulation is rejected (like the *--pdb-on-reject* option).

verbosityLevel = 0

Verbosity level. See *--verbosity* for the allowed values.

showInternalBacktrace = False

Whether or not to include Scenic's innards in backtraces.

Set to True by default so that any errors during import of the scenic module will get full backtraces; the `scenic` module's *__init__.py* sets it to False.

postMortemDebugging = False

Whether or not to do post-mortem debugging of uncaught exceptions.

postMortemRejections = False

Whether or not to do "post-mortem" debugging of rejected scenes/simulations.

hiddenFolders

Folders elided from backtraces when *showInternalBacktrace* is false.

exception ScenicError

Bases: Exception

An error produced during Scenic compilation, scene generation, or simulation.

exception ScenicSyntaxError

Bases: *ScenicError*

An error produced by attempting to parse an invalid Scenic program.

This is intentionally not a subclass of SyntaxError so that pdb can be used for post-mortem debugging of the parser. Our custom excepthook below will arrange to still have it formatted as a SyntaxError, though.

exception ParseCompileError(*exc*)

Bases: *ScenicSyntaxError*

Error occurring during Scenic/Python parsing or compilation.

exception ScenicParseError(*exc*)

Bases: *ParseCompileError*

Error occuring during Scenic parsing or compilation.

exception PythonCompileError(*exc*)

Bases: *ParseCompileError*

Error occuring during Python compilation of translated Scenic code.

exception ASTParseError(*node*, *message*, *filename*)

Bases: *ScenicSyntaxError*

Parse error occuring during modification of the Python AST.

exception InvalidScenarioError

Bases: *ScenicError*

Error raised for syntactically-valid but otherwise problematic Scenic programs.

exception InconsistentScenarioError(*line*, *message*)

Bases: *InvalidScenarioError*

Error for scenarios with inconsistent requirements.

exception SpecifierError

Bases: *ScenicError*

Error for illegal uses of specifiers.

displayScenicException(*exc*, *seen=None*)

Print a Scenic exception, cleaning up the traceback if desired.

If `showInternalBacktrace` is False, this hides frames inside Scenic itself.

callBeginningScenicTrace(*func*)

Call the given function, starting the Scenic backtrace at that point.

This function is just a convenience to make Scenic backtraces cleaner when running Scenic programs from the command line.

getText(*filename*, *lineno*, *line=''*, *offset=0*, *end_offset=None*)

Attempt to recover the text of an error from the original Scenic file.

scenic.core.external_params

Support for values which are sampled outside of Scenic.

External Samplers in General

External samplers provide a mechanism to use different types of sampling techniques, like optimization or quasi-random sampling, from within a Scenic program. Ordinary random values in Scenic are instances of *Distribution*; this module defines a special subclass, *ExternalParameter*, representing a value which is sampled externally. Scenic programs with external parameters are handled as follows:

1. During compilation, all instances of *ExternalParameter* are gathered together and given to the *ExternalSampler.forParameters* function; this function creates an appropriate *ExternalSampler*, whose configuration can be controlled using global parameters (see the function documentation for details).
2. When sampling a scene, before sampling any other distributions the *sample* method of the *ExternalSampler* is called to sample all the external parameters. For active samplers, this method passes along the `feedback` value given to *Scenario.generate*, if any.
3. Once the external parameters have values, the program is equivalent to one without external parameters, and sampling proceeds as usual. As for every instance of *Distribution*, the external parameters will have their `sampleGiven` method called once all their dependencies have been sampled; by default this method just returns the value sampled for this parameter in step (2).

Note: Note that while external parameters, like all instances of *Distribution*, are allowed to have dependencies, they are an exception to the usual rule that dependencies are always sampled before dependents, because the *ExternalSampler.sample* method is called before any other sampling. However, as explained above, the `sampleGiven` method is called in the proper order and external samplers which need to do sampling based on the values of other distributions can be invoked from it. The two-step mechanism with *ExternalSampler.sample* is provided for samplers which sample the whole space of external parameters at once (e.g. the VerifAI samplers).

Samplers from VerifAI

The external sampling mechanism is designed to be extensible. The only built-in *ExternalSampler* is the *VerifaiSampler*, which provides access to the samplers in the VerifAI toolkit (which in turn can use Scenic as a modeling language).

The *VerifaiSampler* supports several types of external parameters corresponding to the primitive distributions: *VerifaiRange* and *VerifaiDiscreteRange* for continuous and discrete intervals, and *VerifaiOptions* for discrete sets. For example, suppose we write:

```
ego = new Object at (VerifaiRange(5, 15), 0)
```

This is equivalent to the ordinary Scenic line `ego = new Object at (Range(5, 15), 0)`, except that the X coordinate of the ego is sampled by VerifAI within the range (5, 15) instead of being uniformly distributed over it. By default the *VerifaiSampler* uses VerifAI's Halton sampler, so the range will still be covered uniformly but more systematically. If we want to use a different sampler, we can set the `verifaiSamplerType` global parameter:

```
param verifaiSamplerType = 'ce'
ego = new Object at (VerifaiRange(5, 15), 0)
```

Now the X coordinate will be sampled using VerifAI's cross-entropy sampler. If we pass a feedback value to *Scenario.generate* which scores the previous scene, then the coordinate will not be sampled uniformly but rather converge to a distribution concentrated on values minimizing the score. Active samplers like cross-entropy can be used for falsification in this way, driving a system toward parts of the parameter space where a specification is violated.

The cross-entropy sampler in VerifAI can be started from a non-uniform prior. Scenic provides a convenient way to define this prior using the ordinary syntax for distributions:

```
param verifaiSamplerType = 'ce'
ego = new Object at (VerifaiParameter.withPrior(Normal(10, 3)), 0)
```

Now cross-entropy sampling will start from a normal distribution with mean 10 and standard deviation 3. Priors are restricted to primitive distributions and in general may be approximated so that VerifAI can handle them – see *VerifaiParameter.withPrior* for details.

For more information on how to customize the sampler, see *VerifaiSampler*.

Summary of Module Members

Classes

ExternalParameter	A value determined by external code rather than Scenic's internal sampler.
ExternalSampler	Abstract class for objects called to sample values for each external parameter.
VerifaiDiscreteRange	A *DiscreteRange* (integer interval) sampled by VerifAI.
VerifaiOptions	An *Options* (discrete set) sampled by VerifAI.
VerifaiParameter	An external parameter sampled using one of VerifAI's samplers.
VerifaiRange	A *Range* (real interval) sampled by VerifAI.
VerifaiSampler	An external sampler exposing the samplers in the VerifAI toolkit.

Member Details

class ExternalSampler(*params*, *globalParams*)

Abstract class for objects called to sample values for each external parameter.

The initializer for this class takes the same arguments as the factory function *forParameters* below.

Attributes

rejectionFeedback – Value passed to the *sample* method when the last sample was rejected. This value can be chosen by a Scenic scenario using the global parameter `externalSamplerRejectionFeedback`.

static forParameters(*params*, *globalParams*)

Create an *ExternalSampler* given the sets of external and global parameters.

The scenario may explicitly select an external sampler by assigning the global parameter `externalSampler` to a subclass of *ExternalSampler*. Otherwise, a *VerifaiSampler* is used by default.

Parameters

- **params** (*tuple*) – Tuple listing each *ExternalParameter*.
- **globalParams** (*dict*) – Dictionary of global parameters for the *Scenario*, made available here to support sampler customization through setting parameters. Note that the values of these parameters may be instances of *Distribution*!

Returns

An *ExternalSampler* configured for the given parameters.

sample(*feedback*)

Sample values for all the external parameters.

Parameters

feedback – Feedback from the last sample (for active samplers).

nextSample(*feedback*)

Actually do the sampling. Implemented by subclasses.

valueFor(*param*)

Return the sampled value for a parameter. Implemented by subclasses.

class VerifaiSampler(*params*, *globalParams*)

Bases: *ExternalSampler*

An external sampler exposing the samplers in the VerifAI toolkit.

The sampler can be configured using the following Scenic global parameters:

- `verifaiSamplerType` – sampler type (see the `verifai.server.choose_sampler` function); the default is `'halton'`
- `verifaiSamplerParams` – `DotMap` of options passed to the sampler

The *VerifaiSampler* supports external parameters which are instances of *VerifaiParameter*.

class ExternalParameter

Bases: *Distribution*

A value determined by external code rather than Scenic's internal sampler.

sampleGiven(*value*)

Specialization of *Samplable.sampleGiven* for external parameters.

By default, this method simply looks up the value previously sampled by *ExternalSampler.sample*.

class VerifaiParameter(*domain*)

Bases: *ExternalParameter*

An external parameter sampled using one of VerifAI's samplers.

static withPrior(*dist*, *buckets=None*)

Creates a *VerifaiParameter* using the given distribution as a prior.

Since the VerifAI cross-entropy sampler currently only supports piecewise-constant distributions, if the prior is not of that form it may be approximated. For most built-in distributions, the approximation is exact: for a particular distribution, check its *bucket* method.

class VerifaiRange(*low*, *high*, *buckets=None*, *weights=None*)

Bases: *VerifaiParameter*

A *Range* (real interval) sampled by VerifAI.

_defaultValueType

alias of `float`

class VerifaiDiscreteRange(*low*, *high*, *weights=None*)

Bases: *VerifaiParameter*

A *DiscreteRange* (integer interval) sampled by VerifAI.

_defaultValueType

alias of `float`

class VerifaiOptions(*opts*)

Bases: *Options*

An *Options* (discrete set) sampled by VerifAI.

scenic.core.geometry

Utility functions for geometric computation.

Summary of Module Members

Functions

allChains	
apparentHeadingAtPoint	
averageVectors	
cleanChain	
cleanPolygon	
cos	
distanceToLine	
findMinMax	
headingOfSegment	
hypot	
max	
min	
normalizeAngle	
plotPolygon	
pointIsInCone	
polygonUnion	
removeHoles	
rotateVector	
sin	
splitSelfIntersections	
triangulatePolygon	Triangulate the given Shapely polygon.
triangulatePolygon_mapbox	
viewAngleToPoint	

Exceptions

TriangulationError	Signals that the installed triangulation libraries are insufficient.

Member Details

exception TriangulationError

Bases: RuntimeError

Signals that the installed triangulation libraries are insufficient.

triangulatePolygon(*polygon*)

Triangulate the given Shapely polygon.

Note that we can't use `shapely.ops.triangulate` since it triangulates point sets, not polygons (i.e., it doesn't respect edges). We need an algorithm for triangulation of polygons with holes (it doesn't need to be a Delaunay triangulation).

Parameters
polygon (*shapely.geometry.Polygon*) – Polygon to triangulate.

Returns
A list of disjoint (except for edges) triangles whose union is the original polygon.

scenic.core.lazy_eval

Support for lazy evaluation of expressions and specifiers.

Lazy evaluation is necessary for expressions like `30 deg relative to roadDirection` where `roadDirection` is a vector field and so defines a different heading at different positions. Scenic defers evaluation of such expressions until they are used in the definition of an object, when the required context (here, a position) is available. This is implemented by representing lazy values as special objects which capture all operations applied to them (in a similar way to *Distribution* objects). The main class of such objects is *DelayedArgument*: in the above example, the `relative to` operator returns such an object. However, since lazy values can appear as arguments to distributions, *Distribution* objects can also require lazy evaluation (prior to sampling); therefore both of these classes derive from a common abstract class *LazilyEvaluable*.

Summary of Module Members

Functions

dependencies	Dependencies which must be sampled before this value.
isLazy	Whether this value requires either sampling or lazy evaluation.
makeDelayedFunctionCall	Utility function for creating a lazily-evaluated function call.
makeDelayedOperatorHandler	
needsLazyEvaluation	Whether the given value requires lazy evaluation.
needsSampling	Whether this value requires sampling.
requiredProperties	Set of properties needed to evaluate the given value, if any.
toLazyValue	Wrap a Python object in a *DelayedArgument* if it needs lazy evaluation.
valueInContext	Evaluate something in the context of an object being constructed.

Classes

DelayedArgument	Specifier arguments requiring other properties to be evaluated first.
LazilyEvaluable	Values which may require evaluation in the context of an object being constructed.

Member Details

class LazilyEvaluable(*requiredProps*, *dependencies=()*)

Values which may require evaluation in the context of an object being constructed.

If a LazilyEvaluable specifies any properties it depends on, then it cannot be evaluated to a normal value except during the construction of an object which already has values for those properties.

Parameters

- **requiredProps** – sequence of strings naming all properties which this value can depend on (formally, which must exist in the object passed as the context to *evaluateIn*).
- **dependencies** – for internal use only (see *Samplable*).

Attributes

_requiredProperties – set of strings as above.

evaluateIn(*context*)

Evaluate this value in the context of an object being constructed.

The object must define all of the properties on which this value depends.

evaluateInner(*context*)

Actually evaluate in the given context, which provides all required properties.

Overridden by subclasses.

static makeContext(***props*)

Make a context with the given properties.

class DelayedArgument(*requiredProps*, *value*, *_internal=False*)

Bases: *LazilyEvaluable*

Specifier arguments requiring other properties to be evaluated first.

The value of a DelayedArgument is given by a function mapping the context (object under construction) to a value.

Note: When called from a dynamic behavior, constructors for delayed arguments return *actual evaluations*, not *DelayedArgument* objects. The agent running the behavior is used as the evaluation context.

Parameters

- **requiredProps** – see *LazilyEvaluable*.
- **value** – function taking a single argument (the context) and returning the corresponding evaluation of this object.
- **_internal** (*bool*) – set to True for internal uses that need to suppress the exceptional handling of calls from dynamic behaviors above.

makeDelayedFunctionCall(*func*, *args*, *kwargs={}*)

Utility function for creating a lazily-evaluated function call.

valueInContext(*value*, *context*)

Evaluate something in the context of an object being constructed.

toLazyValue(*thing*)

Wrap a Python object in a *DelayedArgument* if it needs lazy evaluation.

requiredProperties(*thing*)

Set of properties needed to evaluate the given value, if any.

needsLazyEvaluation(*thing*)

Whether the given value requires lazy evaluation.

dependencies(*thing*)

Dependencies which must be sampled before this value.

needsSampling(*thing*)

Whether this value requires sampling.

isLazy(*thing*)

Whether this value requires either sampling or lazy evaluation.

scenic.core.object_types

Implementations of the built-in Scenic classes.

Defines the 3 Scenic classes *Point*, *OrientedPoint*, and *Object*, and associated helper code (notably their base class *Constructible*, which implements the handling of property definitions and *Specifier Resolution*).

Warning: In 2D compatibility mode, these classes are overwritten with 2D analogs. While we make an effort to map imports to the correct class, this only works if imports use the form `import scenic.core.object_types as object_types` followed by accessing `object_types.Object`. If you instead use `from scenic.core.object_types import Object`, you may get the wrong class.

Summary of Module Members

Module Attributes

Interval	Type alias for an interval (a pair of floats).
DimensionLimits	Type alias for limits on dimensions (a triple of intervals).

Functions

defaultSideSurface	Extracts a side surface from the occupiedSpace of an object.
disableDynamicProxyFor	
enableDynamicProxyFor	
setDynamicProxyFor	

Classes

Constructible	Abstract base class for Scenic objects.
Mutator	An object controlling how the *mutate* statement affects an *Object*.
Object	The Scenic class `Object`.
Object2D	A 2D version of *Object*, used for backwards compatibility with Scenic 2.0
OrientationMutator	Mutator adding Gaussian noise to `yaw`, `pitch`, and `roll`.
OrientedPoint	The Scenic class `OrientedPoint`.
OrientedPoint2D	A 2D version of *OrientedPoint*, used for backwards compatibility with Scenic 2.0
Point	The Scenic base class `Point`.
Point2D	A 2D version of *Point*, used for backwards compatibility with Scenic 2.0
PositionMutator	Mutator adding Gaussian noise to `position`.

Member Details

Interval

Type alias for an interval (a pair of floats).

alias of Tuple[float, float]

DimensionLimits

Type alias for limits on dimensions (a triple of intervals).

alias of Tuple[Tuple[float, float], Tuple[float, float], Tuple[float, float]]

class Constructible(*properties*, *constProps=frozenset({})*, *_internal=False*)

Bases: *Samplable*

Abstract base class for Scenic objects.

Scenic objects, which are constructed using specifiers, are implemented internally as instances of ordinary Python classes. This abstract class implements the procedure to resolve specifiers and determine values for the properties of an object, as well as several common methods supported by objects.

> **Warning:** This class is an implementation detail, and none of its methods should be called directly from a Scenic program.

classmethod _withProperties(*properties*, *constProps=None*)

Create an instance with the given property values.

Values of unspecified properties are determined by specifier resolution as usual.

classmethod _withSpecifiers(*specifiers*, *constProps=None*, *register=True*)

Create an instance from the given specifiers.

_copyWith(***overrides*)

Copy this object, possibly overriding some of its properties.

class Mutator

An object controlling how the *mutate* statement affects an *Object*.

A *Mutator* can be assigned to the mutator property of an *Object* to control the effect of the *mutate* statement. When mutation is enabled for such an object using that statement, the mutator's *appliedTo* method is called to compute a mutated version. The *appliedTo* method can also decide whether to apply mutators inherited from superclasses.

appliedTo(*obj*)

Return a mutated copy of the given object. Implemented by subclasses.

The mutator may inspect the mutationScale attribute of the given object to scale its effect according to the scale given in mutate O by S.

Returns

A pair consisting of the mutated copy of the object (which is most easily created using *_copyWith*) together with a Boolean indicating whether the mutator inherited from the superclass (if any) should also be applied.

class PositionMutator(*stddevs*)

Bases: *Mutator*

Mutator adding Gaussian noise to position. Used by *Point*.

Attributes
stddevs (*tuple[float,float,float]*) – standard deviation of noise for each dimension (x,y,z).

`class OrientationMutator`(*stddevs*)

Bases: *Mutator*

Mutator adding Gaussian noise to `yaw`, `pitch`, and `roll`. Used by *OrientedPoint*.

Attributes
stddevs (*tuple[float,float,float]*) – standard deviation of noise for each angle (yaw, pitch, roll).

`class Point` <specifiers>

Bases: *Constructible*

The Scenic base class `Point`.

The default mutator for *Point* adds Gaussian noise to `position` with a standard deviation given by the `positionStdDev` property.

Properties

- **position** (*Vector*; dynamic) – Position of the point. Default value is the origin (0,0,0).
- **width** (*float*) – Default value 0 (only provided for compatibility with operators that expect an *Object*).
- **length** (*float*) – Default value 0.
- **height** (*float*) – Default value 0.
- **baseOffset** (*Vector*) – Only provided for compatibility with the *on region* specifier. Default value is (0,0,0).
- **contactTolerance** (*float*) – Only provided for compatibility with the specifiers that expect an *Object*. Default value 0.
- **onDirection** (*Vector*) – The direction used to determine where to place this *Point* on a region, when using the modifying *on* specifier. See the *on region* page for more details. Default value is None, indicating the direction will be inferred from the region this object is being placed on.
- **visibleDistance** (*float*) – Distance used to determine the visible range of this object. Default value 50.
- **viewRayDensity** (*float*) – By default determines the number of rays used during visibility checks. This value is the density of rays per degree of visible range in one dimension. The total number of rays sent will be this value squared per square degree of this object's view angles. This value determines the default value for `viewRayCount`, so if `viewRayCount` is overwritten this value is ignored. Default value 5.
- **viewRayCount** (*None | tuple[float, float]*) – The total number of horizontal and vertical view angles to be sent, or None if this value should be computed automatically. Default value `None`.
- **viewRayDistanceScaling** (*bool*) – Whether or not the number of rays should scale with the distance to the object. Ignored if `viewRayCount` is passed. Default value `False`.
- **mutationScale** (*float*) – Overall scale of mutations, as set by the *mutate* statement. Default value 0 (mutations disabled).
- **positionStdDev** (*tuple[float, float, float]*) – Standard deviation of Gaussian noise for each dimension (x,y,z) to be added to this object's `position` when mutation is enabled with scale 1. Default value (1,1,0), mutating only the x,y values of the point.

property visibleRegion

The visible region of this object.

The visible region of a *Point* is a sphere centered at its `position` with radius `visibleDistance`.

canSee(*other*, *occludingObjects=()*, *debug=False*)

Whether or not this *Point* can see `other`.

Parameters

- **other** – A *Point*, *OrientedPoint*, or *Object* to check for visibility.
- **occludingObjects** – A list of objects that can occlude visibility.

Return type
bool

class OrientedPoint <specifiers>

Bases: *Point*

The Scenic class `OrientedPoint`.

The default mutator for *OrientedPoint* adds Gaussian noise to `yaw` while leaving `pitch` and `roll` unchanged, using the three standard deviations (for yaw/pitch/roll respectively) given by the `orientationStdDev` property. It then also applies the mutator for *Point*.

The default mutator for *OrientedPoint* adds Gaussian noise to `yaw`, `pitch` and `roll` according to `orientationStdDev`. By default the standard deviations for `pitch` and `roll` are zero so that, by default, only `yaw` is mutated.

Properties

- **yaw** (*float; dynamic*) – Yaw of the *OrientedPoint* in radians in the local coordinate system provided by `parentOrientation`. Default value 0.
- **pitch** (*float; dynamic*) – Pitch of the *OrientedPoint* in radians in the local coordinate system provided by `parentOrientation`. Default value 0.
- **roll** (*float; dynamic*) – Roll of the *OrientedPoint* in radians in the local coordinate system provided by `parentOrientation`. Default value 0.
- **parentOrientation** (*Orientation*) – The local coordinate system that the *OrientedPoint*'s `yaw`, `pitch`, and `roll` are interpreted in. Default value is the global coordinate system, where an object is flat in the XY plane, facing North.
- **orientation** (*Orientation*; dynamic; final) – The orientation of the *OrientedPoint* relative to the global coordinate system. Derived from the `yaw`, `pitch`, `roll`, and `parentOrientation` of this *OrientedPoint* and non-overridable.
- **heading** (*float; dynamic; final*) – Yaw value of this *OrientedPoint* in the global coordinate system. Derived from `orientation` and non-overridable.
- **viewAngles** (*tuple[float,float]*) – Horizontal and vertical view angles of this *OrientedPoint* in radians. Horizontal view angle can be up to 2 and vertical view angle can be up to . Values greater than these will be truncated. Default value is (2,)
- **orientationStdDev** (*tuple[float,float,float]*) – Standard deviation of Gaussian noise to add to this object's Euler angles (yaw, pitch, roll) when mutation is enabled with scale 1. Default value (5°, 0, 0), mutating only the `yaw` of this *OrientedPoint*.

property visibleRegion

The visible region of this object.

The visible region of an *OrientedPoint* restricts that of *Point* (a sphere with radius visibleDistance) based on the value of viewAngles. In general, it is a capped rectangular pyramid subtending an angle of viewAngles[0] horizontally and viewAngles[1] vertically, as long as those angles are less than /2; larger angles yield various kinds of wrap-around regions. See *ViewRegion* for details.

canSee(*other*, *occludingObjects=()*, *debug=False*)

Whether or not this *OrientedPoint* can see other.

Parameters

- **other** – A *Point*, *OrientedPoint*, or *Object* to check for visibility.
- **occludingObjects** – A list of objects that can occlude visibility.

Return type
bool

distancePast(*vec*)

Distance past a given point, assuming we've been moving in a straight line.

class Object <specifiers>

Bases: *OrientedPoint*

The Scenic class Object.

This is the default base class for Scenic classes.

Properties

- **width** (*float*) – Width of the object, i.e. extent along its X axis. Default value of 1 inherited from the object's shape.
- **length** (*float*) – Length of the object, i.e. extent along its Y axis. Default value of 1 inherited from the object's shape.
- **height** (*float*) – Height of the object, i.e. extent along its Z axis. Default value of 1 inherited from the object's shape.
- **shape** (*Shape*) – The shape of the object, which must be an instance of *Shape*. The default shape is a box, with default unit dimensions.
- **allowCollisions** (*bool*) – Whether the object is allowed to intersect other objects. Default value False.
- **regionContainedIn** (*Region* or None) – A *Region* the object is required to be contained in. If None, the object need only be contained in the scenario's workspace.
- **baseOffset** (*Vector*) – An offset from the position of the Object to the base of the object, used by the *on region* specifier. Default value is (0, 0, -self.height/2), placing the base of the Object at the bottom center of the Object's bounding box.
- **contactTolerance** (*float*) – The maximum distance this object can be away from a surface to be considered on the surface. Objects are placed at half this distance away from a point when the *on region* specifier or a directional specifier like *(left | right) of Object [by scalar]* is used. Default value 1e-4.
- **sideComponentThresholds** (*DimensionLimits*) – Used to determine the various sides of an object (when using the default implementation). The three interior 2-tuples represent the maximum and minimum bounds for each dimension's (x,y,z) surface. See

defaultSideSurface for details. Default value ((-0.5, 0.5), (-0.5, 0.5), (-0.5, 0.5)).

- **cameraOffset** (*Vector*) – Position of the camera for the *can see* operator, relative to the object's position. Default (0, 0, 0).
- **requireVisible** (*bool*) – Whether the object is required to be visible from the ego object. Default value False.
- **occluding** (*bool*) – Whether or not this object can occlude other objects. Default value True.
- **showVisibleRegion** (*bool*) – Whether or not to display the visible region in the Scenic internal visualizer.
- **color** (tuple[float, float, float, float] or tuple[float, float, float] or None) – An optional color (with optional alpha) property that is used by the internal visualizer, or possibly simulators. All values should be between 0 and 1. Default value None
- **velocity** (*Vector*; *dynamic*) – Velocity in dynamic simulations. Default value is the velocity determined by speed and orientation.
- **speed** (*float; dynamic*) – Speed in dynamic simulations. Default value 0.
- **angularVelocity** (*Vector*; *dynamic*)
- **angularSpeed** (*float; dynamic*) – Angular speed in dynamic simulations. Default value 0.
- **behavior** – Behavior for dynamic agents, if any (see *Dynamic Scenarios*). Default value None.
- **lastActions** – Tuple of actions taken by this agent in the last time step (or None if the object is not an agent or this is the first time step).

startDynamicSimulation()

Hook called when the object is created in a dynamic simulation.

Does nothing by default; provided for objects to do simulator-specific initialization as needed.

Changed in version 3.0: This method is called on objects created in the middle of dynamic simulations, not only objects present in the initial scene.

containsPoint(*point*)

Whether or not the space this object occupies contains a point

distanceTo(*point*)

The minimal distance from the space this object occupies to a given point

intersects(*other*)

Whether or not this object intersects another object

property visibleRegion

The visible region of this object.

The visible region of an *Object* is the same as that of an *OrientedPoint* (see *OrientedPoint.visibleRegion*) except that it is offset by the value of cameraOffset (which is the zero vector by default).

canSee(*other*, *occludingObjects=()*, *debug=False*)

Whether or not this *Object* can see other.

Parameters

- **other** – A *Point*, *OrientedPoint*, or *Object* to check for visibility.
- **occludingObjects** – A list of objects that can occlude visibility.

Return type
bool

property corners

A tuple containing the corners of this object's bounding box

property occupiedSpace

A region representing the space this object occupies

property _isConvex

Whether this object's shape is convex

property boundingBox

A region representing this object's bounding box

property inradius

A lower bound on the inradius of this object

property surface

A region containing the entire surface of this object

property onSurface

The surface used by the on specifier.

This region is used to sample position when another object is placed on this object. By default the top surface of this object (*topSurface*), but can be overwritten by subclasses.

property topSurface

A region containing the top surface of this object

For how this surface is computed, see *defaultSideSurface*.

property rightSurface

A region containing the right surface of this object

For how this surface is computed, see *defaultSideSurface*.

property leftSurface

A region containing the left surface of this object

For how this surface is computed, see *defaultSideSurface*.

property frontSurface

A region containing the front surface of this object

For how this surface is computed, see *defaultSideSurface*.

property backSurface

A region containing the back surface of this object

For how this surface is computed, see *defaultSideSurface*.

property bottomSurface

A region containing the bottom surface of this object

For how this surface is computed, see *defaultSideSurface*.

property _isPlanarBox

Whether this object is a box aligned with the XY plane.

defaultSideSurface(*occupiedSpace*, *dimension*, *positive*, *thresholds*)

Extracts a side surface from the occupiedSpace of an object.

This function is the default implementation for computing a region representing a side surface of an object. This is done by keeping only the faces of the object's `occupiedSpace` mesh that have normal vectors with a large/small enough x,y, or z component. For example, for the front surface of an object we would would keep all faces that had a normal vector with y component greater than `thresholds[1][1]` and for the back surface of an object we would keep all faces that had a normal vector with y component less than `thresholds[1][0]`.

Parameters

- **occupiedSpace** – The *occupiedSpace* region of the object to extract the side surface from.
- **dimension** – The target dimension who's component will be checked.
- **positive** – If `False`, the target component must be less than the first value in the appropriate tuple. If `True`, the component must be greater than the second value in the appropriate tuple.
- **thresholds** – A 3-tuple of 2-tuples, one for each dimension (x,y,z), with each tuple containing the thresholds for a non-positive and positive side, respectively, in each dimension.
- **on_dimension** – The on_dimension to be passed to the created surface.

Return type
MeshSurfaceRegion

class Point2D <specifiers>

Bases: *Point*

A 2D version of *Point*, used for backwards compatibility with Scenic 2.0

_3DClass

alias of *Point*

property visibleRegion

The visible region of this 2D point.

The visible region of a *Point* is a disc centered at its `position` with radius `visibleDistance`.

class OrientedPoint2D <specifiers>

Bases: *Point2D*, *OrientedPoint*

A 2D version of *OrientedPoint*, used for backwards compatibility with Scenic 2.0

_3DClass

alias of *OrientedPoint*

property visibleRegion

The visible region of this 2D oriented point.

The visible region of an *OrientedPoint* is a sector of the disc centered at its `position` with radius `visibleDistance`, oriented along `heading` and subtending an angle of `viewAngle`.

class Object2D <specifiers>

Bases: *OrientedPoint2D*, *Object*

A 2D version of *Object*, used for backwards compatibility with Scenic 2.0

_3DClass

alias of *Object*

property visibleRegion

The visible region of this 2D object.

The visible region of a 2D *Object* is a circular sector as for *OrientedPoint*, except that the base of the sector may be offset from `position` by the `cameraOffset` property (to allow modeling cameras which are not located at the center of the object).

scenic.core.propositions

Objects representing propositions that can be used to specify conditions

Summary of Module Members

Classes

Always	
And	
Atomic	
Eventually	
Implies	
Next	
Not	
Or	
PropositionMonitor	
PropositionNode	Base class for temporal and non-temporal propositions
UnaryProposition	Base class for temporal unary operators
Until	

Member Details

class PropositionNode(*ltl_node*)

Base class for temporal and non-temporal propositions

is_temporal

tells if the proposition is temporal

`check_constrains_sampling()`

Checks if the proposition can be used for pruning.

A requirement can be used for pruning if it is evaluated on the scene generation phase before simulation, and violation in that phase immediately results in discarding the scene and regenerating a new one. For simplicity, we currently check two special cases: 1. requirements with no temporal requirement 2. requirements with only one always operator on top-level

Returns
bool – True if the requirement is one of the forms above. False otherwise.

`property children: List[PropositionNode]`

Returns all children of proposition tree.

Returns
list – proposition nodes that are directly under this node

`flatten()`

Flattens the tree and return the list of nodes.

Returns
list – list of all children nodes

Return type
List[PropositionNode]

`class UnaryProposition(ltl_node)`

Bases: *PropositionNode*

Base class for temporal unary operators

scenic.core.pruning

Pruning parts of the sample space which violate requirements.

The top-level function here, *prune*, is called as the very last step of scenario compilation (from *translator.constructScenarioFrom*).

Summary of Module Members

Functions

currentPropValue	Get the current value of an object's property, taking into account prior pruning.
feasibleRHPolygon	Find where objects aligned to the given fields can satisfy the given RH bounds.
isFunctionCall	Match calls to a given function, taking into account distribution decorators.
isMethodCall	Match calls to a given method, taking into account distribution decorators.
matchInRegion	Match uniform samples from a *Region*
matchPolygonalField	Match orientation yaw defined by a *PolygonalVectorField* at the given position.
maxDistanceBetween	Upper bound the distance between the given Objects.
prune	Prune a *Scenario*, removing infeasible parts of the space.
pruneContainment	Prune based on the requirement that individual Objects fit within their container.
pruneRelativeHeading	Prune based on requirements bounding the relative heading of an Object.
relativeHeadingRange	Lower/upper bound the possible RH between two headings with bounded disturbances.
visibilityBound	Upper bound the distance from an Object to another it can see.

Member Details

currentPropValue(*obj*, *prop*)

Get the current value of an object's property, taking into account prior pruning.

isMethodCall(*thing*, *method*)

Match calls to a given method, taking into account distribution decorators.

isFunctionCall(*thing*, *function*)

Match calls to a given function, taking into account distribution decorators.

matchInRegion(*position*)

Match uniform samples from a *Region*

Returns the Region, if any, and a lower and upper bound on the distance the object will be placed along with any offset that should be added to the base.

matchPolygonalField(*heading*, *position*)

Match orientation yaw defined by a *PolygonalVectorField* at the given position.

Matches the yaw attribute of orientations exactly equal to a *PolygonalVectorField*, or offset by a bounded disturbance. Returns a triple consisting of the matched field if any, together with lower/upper bounds on the disturbance.

prune(*scenario*, *verbosity=1*)

Prune a *Scenario*, removing infeasible parts of the space.

This function directly modifies the Distributions used in the Scenario, but leaves the conditional distribution under the scenario's requirements unchanged. See *Samplable.conditionTo*.

Currently, the following pruning techniques are applied in order:

- Pruning based on containment (*pruneContainment*)
- Pruning based on relative heading bounds (*pruneRelativeHeading*)

pruneContainment(*scenario*, *verbosity*)

Prune based on the requirement that individual Objects fit within their container.

Specifically, if O is positioned uniformly (with a possible offset) in region B and has container C, then we can instead pick a position uniformly in their intersection. If we can also lower bound the radius of O, then we can first erode C by that distance minus that maximum offset distance.

pruneRelativeHeading(*scenario*, *verbosity*)

Prune based on requirements bounding the relative heading of an Object.

Specifically, if an object O is:

- positioned uniformly within a polygonal region B;
- aligned to a polygonal vector field F (up to a bounded offset);

and another object O' is:

- aligned to a polygonal vector field F' (up to a bounded offset);
- at most some finite maximum distance from O;
- required to have relative heading within a bounded offset of that of O;

then we can instead position O uniformly in the subset of B intersecting the cells of F which satisfy the relative heading requirements w.r.t. some cell of F' which is within the distance bound.

maxDistanceBetween(*scenario*, *obj*, *target*)

Upper bound the distance between the given Objects.

visibilityBound(*obj*, *target*)

Upper bound the distance from an Object to another it can see.

feasibleRHPolygon(*field*, *offsetL*, *offsetR*, *tField*, *tOffsetL*, *tOffsetR*, *lowerBound*, *upperBound*, *maxDist*)

Find where objects aligned to the given fields can satisfy the given RH bounds.

relativeHeadingRange(*baseHeading*, *offsetL*, *offsetR*, *targetHeading*, *tOffsetL*, *tOffsetR*)

Lower/upper bound the possible RH between two headings with bounded disturbances.

scenic.core.regions

Objects representing regions in space.

Manipulations of polygons and line segments are done using the shapely package.

Manipulations of meshes is done using the trimesh package.

Summary of Module Members

Module Attributes

everywhere	A *Region* containing all points.
nowhere	A *Region* containing no points.

Functions

convertToFootprint	Recursively convert a region into it's footprint.
orientationFor	
regionFromShapelyObject	Build a 'Region' from Shapely geometry.
toPolygon	

Classes

AllRegion	Region consisting of all space.
BoxRegion	Region in the shape of a rectangular cuboid, i.e. a box.
CircularRegion	A circular region with a possibly-random center and radius.
CylinderSectionRegion	
DifferenceRegion	
EmptyRegion	Region containing no points.
GridRegion	A Region given by an obstacle grid.
IntersectionRegion	
MeshRegion	Region given by a scaled, positioned, and rotated mesh.
MeshSurfaceRegion	A region representing the surface of a mesh.
MeshVolumeRegion	A region representing the volume of a mesh.
PathRegion	A region composed of multiple polylines in 3D space.
PointInRegionDistribution	Uniform distribution over points in a Region
PointSetRegion	Region consisting of a set of discrete points.
PolygonalFootprintRegion	Region that contains all points in a polygonal footprint, regardless of their z value.
PolygonalRegion	Region given by one or more polygons (possibly with holes) at a fixed z coordinate.
PolylineRegion	Region given by one or more polylines (chain of line segments).
RectangularRegion	A rectangular region with a possibly-random position, heading, and size.
Region	An abstract base class for Scenic Regions
SectorRegion	A sector of a *CircularRegion*.
SpheroidRegion	Region in the shape of a spheroid.
SurfaceCollisionTrimesh	A Trimesh object that always returns non-convex.
UnionRegion	
ViewRegion	The viewing volume of a camera defined by a radius and horizontal/vertical view angles.
ViewSectionRegion	

Exceptions

UndefinedSamplingException

Member Details

class Region(*name*, **dependencies*, *orientation=None*)

Bases: *Samplable*, ABC

An abstract base class for Scenic Regions

abstract uniformPointInner()

Do the actual random sampling. Implemented by subclasses.

abstract containsPoint(*point*)

Check if the *Region* contains a point. Implemented by subclasses.

Return type
bool

abstract containsObject(*obj*)

Check if the *Region* contains an *Object*

Return type
bool

abstract containsRegionInner(*reg*, *tolerance*)

Check if the *Region* contains a *Region*

Return type
bool

abstract distanceTo(*point*)

Distance to this region from a given point.

Return type
float

abstract projectVector(*point*, *onDirection*)

Returns point projected onto this region along onDirection.

abstract property AABB

Axis-aligned bounding box for this *Region*.

intersects(*other*)

Check if this *Region* intersects another.

Return type
bool

intersect(*other*, *triedReversed=False*)

Get a *Region* representing the intersection of this one with another.

If both regions have a preferred orientation, the one of `self` is inherited by the intersection.

Return type
Region

union(*other*, *triedReversed=False*)

Get a *Region* representing the union of this one with another.

Not supported by all region types.

Return type
Region

difference(*other*)

Get a *Region* representing the difference of this one and another.

Not supported by all region types.

Return type
Region

static uniformPointIn(*region*)

Get a uniform *Distribution* over points in a *Region*.

orient(*vec*)

Orient the given vector along the region's orientation, if any.

class PointInRegionDistribution(*region*)

Bases: *VectorDistribution*

Uniform distribution over points in a Region

class AllRegion(*name*, **dependencies*, *orientation=None*)

Bases: *Region*

Region consisting of all space.

class EmptyRegion(*name*, **dependencies*, *orientation=None*)

Bases: *Region*

Region containing no points.

everywhere = <AllRegion everywhere>

A *Region* containing all points.

Points may not be sampled from this region, as no uniform distribution over it exists.

nowhere = <EmptyRegion nowhere>

A *Region* containing no points.

Attempting to sample from this region causes the sample to be rejected.

regionFromShapelyObject(*obj*, *orientation=None*)

Build a 'Region' from Shapely geometry.

class SurfaceCollisionTrimesh(*vertices=None*, *faces=None*, *face_normals=None*, *vertex_normals=None*, *face_colors=None*, *vertex_colors=None*, *face_attributes=None*, *vertex_attributes=None*, *metadata=None*, *process=True*, *validate=False*, *merge_tex=None*, *merge_norm=None*, *use_embree=True*, *initial_cache=None*, *visual=None*, ***kwargs*)

Bases: Trimesh

A Trimesh object that always returns non-convex.

Used so that fcl doesn't find collision without an actual surface intersection.

class MeshRegion(*mesh*, *dimensions=None*, *position=None*, *rotation=None*, *orientation=None*, *tolerance=1e-06*, *centerMesh=True*, *onDirection=None*, *engine='blender'*, *name=None*, *additionalDeps=[]*)

Bases: *Region*

Region given by a scaled, positioned, and rotated mesh.

This is an abstract class and cannot be instantiated directly. Instead a subclass should be used, like *MeshVolumeRegion* or *MeshSurfaceRegion*.

The mesh is first placed so the origin is at the center of the bounding box (unless `centerMesh` is `False`). The mesh is scaled to `dimensions`, translated so the center of the bounding box of the mesh is at `positon`, and then rotated to `rotation`.

Meshes are centered by default (since `centerMesh` is true by default). If you disable this operation, do note that scaling and rotation transformations may not behave as expected, since they are performed around the origin.

Parameters

- **mesh** – The base mesh for this MeshRegion.
- **name** – An optional name to help with debugging.
- **dimensions** – An optional 3-tuple, with the values representing width, length, height respectively. The mesh will be scaled such that the bounding box for the mesh has these dimensions.
- **position** – An optional position, which determines where the center of the region will be.
- **rotation** – An optional Orientation object which determines the rotation of the object in space.
- **orientation** – An optional vector field describing the preferred orientation at every point in the region.
- **tolerance** – Tolerance for internal computations.
- **centerMesh** – Whether or not to center the mesh after copying and before transformations.
- **onDirection** – The direction to use if an object being placed on this region doesn't specify one.
- **engine** – Which engine to use for mesh operations. Either "blender" or "scad".
- **additionalDeps** – Any additional sampling dependencies this region relies on.

classmethod fromFile(*path*, *filetype=None*, *compressed=None*, *binary=False*, ***kwargs*)

Load a mesh region from a file, attempting to infer filetype and compression.

For example: "foo.obj.bz2" is assumed to be a compressed .obj file. "foo.obj" is assumed to be an uncompressed .obj file. "foo" is an unknown filetype, so unless a filetype is provided an exception will be raised.

Parameters

- **path** (*str*) – Path to the file to import.
- **filetype** (*str*) – Filetype of file to be imported. This will be inferred if not provided. The filetype must be one compatible with `trimesh.load`.
- **compressed** (*bool*) – Whether or not this file is compressed (with bz2). This will be inferred if not provided.
- **binary** (*bool*) – Whether or not to open the file as a binary file.
- **kwargs** – Additional arguments to the MeshRegion initializer.

projectVector(*point*, *onDirection*)

Find the nearest point in the region following the onDirection or its negation.

Returns None if no such points exist.

property circumcircle

Compute an upper bound on the radius of the region

property boundingPolygon

A PolygonalRegion bounding the mesh

class MeshVolumeRegion(**args*, ***kwargs*)

Bases: *MeshRegion*

A region representing the volume of a mesh.

The mesh passed must be a `trimesh.base.Trimesh` object that represents a well defined volume (i.e. the `is_volume` property must be true), meaning the mesh must be watertight, have consistent winding and have outward facing normals.

The mesh is first placed so the origin is at the center of the bounding box (unless `centerMesh` is `False`). The mesh is scaled to `dimensions`, translated so the center of the bounding box of the mesh is at `positon`, and then rotated to `rotation`.

Meshes are centered by default (since `centerMesh` is true by default). If you disable this operation, do note that scaling and rotation transformations may not behave as expected, since they are performed around the origin.

Parameters

- **mesh** – The base mesh for this region.
- **name** – An optional name to help with debugging.
- **dimensions** – An optional 3-tuple, with the values representing width, length, height respectively. The mesh will be scaled such that the bounding box for the mesh has these dimensions.
- **position** – An optional position, which determines where the center of the region will be.
- **rotation** – An optional Orientation object which determines the rotation of the object in space.
- **orientation** – An optional vector field describing the preferred orientation at every point in the region.
- **tolerance** – Tolerance for internal computations.
- **centerMesh** – Whether or not to center the mesh after copying and before transformations.
- **onDirection** – The direction to use if an object being placed on this region doesn't specify one.
- **engine** – Which engine to use for mesh operations. Either "blender" or "scad".

intersects(*other*, *triedReversed=False*)

Check if this region intersects another.

This function handles intersect calculations for *MeshVolumeRegion* with: * *MeshVolumeRegion* * *MeshSurfaceRegion* * *PolygonalFootprintRegion*

containsPoint(*point*)

Check if this region's volume contains a point.

containsObject(*obj*)

Check if this region's volume contains an *Object*.

intersect(*other*, *triedReversed=False*)

Get a *Region* representing the intersection of this region with another.

This function handles intersection computation for *MeshVolumeRegion* with: * *MeshVolumeRegion* * *PolygonalFootprintRegion* * *PolygonalRegion* * *PathRegion* * *PolylineRegion*

union(*other*, *triedReversed=False*)

Get a *Region* representing the union of this region with another.

This function handles union computation for *MeshVolumeRegion* with:

- *MeshVolumeRegion*

difference(*other*, *debug=False*)

Get a *Region* representing the difference of this region with another.

This function handles union computation for *MeshVolumeRegion* with: * *MeshVolumeRegion* * *PolygonalFootprintRegion*

distanceTo(*point*)

Get the minimum distance from this region to the specified point.

getSurfaceRegion()

Return a region equivalent to this one, except as a MeshSurfaceRegion

getVolumeRegion()

Returns this object, as it is already a MeshVolumeRegion

class MeshSurfaceRegion(**args*, ***kwargs*)

Bases: *MeshRegion*

A region representing the surface of a mesh.

The mesh is first placed so the origin is at the center of the bounding box (unless `centerMesh` is `False`). The mesh is scaled to `dimensions`, translated so the center of the bounding box of the mesh is at `positon`, and then rotated to `rotation`.

Meshes are centered by default (since `centerMesh` is true by default). If you disable this operation, do note that scaling and rotation transformations may not behave as expected, since they are performed around the origin.

If an orientation is not passed to this mesh, a default orientation is provided which provides an orientation that aligns an object's z axis with the normal vector of the face containing that point, and has a yaw aligned with a yaw of 0 in the global coordinate system.

Parameters

- **mesh** – The base mesh for this region.
- **name** – An optional name to help with debugging.
- **dimensions** – An optional 3-tuple, with the values representing width, length, height respectively. The mesh will be scaled such that the bounding box for the mesh has these dimensions.
- **position** – An optional position, which determines where the center of the region will be.
- **rotation** – An optional Orientation object which determines the rotation of the object in space.

- **orientation** – An optional vector field describing the preferred orientation at every point in the region.
- **tolerance** – Tolerance for internal computations.
- **centerMesh** – Whether or not to center the mesh after copying and before transformations.
- **onDirection** – The direction to use if an object being placed on this region doesn't specify one.

intersects(*other*, *triedReversed=False*)

Check if this region's surface intersects another.

This function handles intersection computation for *MeshSurfaceRegion* with: * *MeshSurfaceRegion* * *PolygonalFootprintRegion*

containsPoint(*point*)

Check if this region's surface contains a point.

distanceTo(*point*)

Get the minimum distance from this object to the specified point.

getFlatOrientation(*pos*)

Get a flat orientation at a point in the region.

Given a point on the surface of the mesh, returns an orientation that aligns an instance's z axis with the normal vector of the face containing that point. Since there are infinitely many such orientations, the orientation returned has yaw aligned with a global yaw of 0.

If `pos` is not within `self.tolerance` of the surface of the mesh, a `RejectionException` is raised.

getVolumeRegion()

Return a region equivalent to this one, except as a MeshVolumeRegion

getSurfaceRegion()

Returns this object, as it is already a MeshSurfaceRegion

class BoxRegion(**args*, ***kwargs*)

Bases: *MeshVolumeRegion*

Region in the shape of a rectangular cuboid, i.e. a box.

By default the unit box centered at the origin and aligned with the axes is used.

Parameters are the same as *MeshVolumeRegion*, with the exception of the `mesh` parameter which is excluded.

class SpheroidRegion(**args*, ***kwargs*)

Bases: *MeshVolumeRegion*

Region in the shape of a spheroid.

By default the unit sphere centered at the origin and aligned with the axes is used.

Parameters are the same as *MeshVolumeRegion*, with the exception of the `mesh` parameter which is excluded.

class PolygonalFootprintRegion(*polygon*, *name=None*)

Bases: *Region*

Region that contains all points in a polygonal footprint, regardless of their z value.

This region cannot be sampled from, as it has infinite height and therefore infinite volume.

Parameters

- **polygon** – A `shapely` `Polygon` or `MultiPolygon`, that defines the footprint of this region.
- **name** – An optional name to help with debugging.

intersect(*other*, *triedReversed=False*)

Get a *Region* representing the intersection of this region with another.

This function handles intersection computation for *PolygonalFootprintRegion* with: * *PolygonalFootprintRegion* * *PolygonalRegion*

union(*other*, *triedReversed=False*)

Get a *Region* representing the union of this region with another.

This function handles union computation for *PolygonalFootprintRegion* with: * *PolygonalFootprintRegion*

difference(*other*)

Get a *Region* representing the difference of this region with another.

This function handles difference computation for *PolygonalFootprintRegion* with: * *PolygonalFootprintRegion*

containsPoint(*point*)

Checks if a point is contained in the polygonal footprint.

Equivalent to checking if the (x, y) values are contained in the polygon.

Parameters
point – A point to be checked for containment.

containsObject(*obj*)

Checks if an object is contained in the polygonal footprint.

Parameters
obj – An object to be checked for containment.

distanceTo(*point*)

Minimum distance from this polygonal footprint to the target point

approxBoundFootprint(*centerZ*, *height*)

Returns an overapproximation of boundFootprint

Returns a volume that is guaranteed to contain the result of boundFootprint(centerZ, height), but may be taller. Used to save time on recomputing boundFootprint.

boundFootprint(*centerZ*, *height*)

Cap the footprint of the object to a given height, centered at a given z.

Parameters

- **centerZ** – The resulting mesh will be vertically centered at this height.
- **height** – The resulting mesh will have this height.

class PathRegion(*points=None*, *polylines=None*, *tolerance=1e-08*, *name=None*)

Bases: *Region*

A region composed of multiple polylines in 3D space.

One of points or polylines should be provided.

Parameters

- **points** – A list of points defining a single polyline.

- **polylines** – A list of list of points, defining multiple polylines.
- **tolerance** – Tolerance used internally.

class PolygonalRegion(*points=None*, *polygon=None*, *z=0*, *orientation=None*, *name=None*, *additionalDeps=[]*)

Bases: *Region*

Region given by one or more polygons (possibly with holes) at a fixed z coordinate.

The region may be specified by giving either a sequence of points defining the boundary of the polygon, or a collection of `shapely` polygons (a `Polygon` or `MultiPolygon`).

Parameters

- **points** – sequence of points making up the boundary of the polygon (or `None` if using the **polygon** argument instead).
- **polygon** – `shapely` polygon or collection of polygons (or `None` if using the **points** argument instead).
- **z** – The z coordinate the polygon is located at.
- **orientation** (*VectorField*; optional) – preferred orientation to use.
- **name** (*str; optional*) – name for debugging.

property boundary: *PolylineRegion*

Get the boundary of this region as a *PolylineRegion*.

class CircularRegion(*center*, *radius*, *resolution=32*, *name=None*)

Bases: *PolygonalRegion*

A circular region with a possibly-random center and radius.

Parameters

- **center** (*Vector*) – center of the disc.
- **radius** (*float*) – radius of the disc.
- **resolution** (*int; optional*) – number of vertices to use when approximating this region as a polygon.
- **name** (*str; optional*) – name for debugging.

class SectorRegion(*center*, *radius*, *heading*, *angle*, *resolution=32*, *name=None*)

Bases: *PolygonalRegion*

A sector of a *CircularRegion*.

This region consists of a sector of a disc, i.e. the part of a disc subtended by a given arc.

Parameters

- **center** (*Vector*) – center of the corresponding disc.
- **radius** (*float*) – radius of the disc.
- **heading** (*float*) – heading of the centerline of the sector.
- **angle** (*float*) – angle subtended by the sector.
- **resolution** (*int; optional*) – number of vertices to use when approximating this region as a polygon.
- **name** (*str; optional*) – name for debugging.

class RectangularRegion(*position*, *heading*, *width*, *length*, *name=None*)

Bases: *PolygonalRegion*

A rectangular region with a possibly-random position, heading, and size.

Parameters

- **position** (*Vector*) – center of the rectangle.
- **heading** (*float*) – the heading of the `length` axis of the rectangle.
- **width** (*float*) – width of the rectangle.
- **length** (*float*) – length of the rectangle.
- **name** (*str; optional*) – name for debugging.

class PolylineRegion(*points=None*, *polyline=None*, *orientation=True*, *name=None*)

Bases: *Region*

Region given by one or more polylines (chain of line segments).

The region may be specified by giving either a sequence of points or `shapely` polylines (a `LineString` or `MultiLineString`).

Parameters

- **points** – sequence of points making up the polyline (or `None` if using the **polyline** argument instead).
- **polyline** – `shapely` polyline or collection of polylines (or `None` if using the **points** argument instead).
- **orientation** (*optional*) – preferred orientation to use, or `True` to use an orientation aligned with the direction of the polyline (the default).
- **name** (*str; optional*) – name for debugging.

property start

Get an *OrientedPoint* at the start of the polyline.

The OP's orientation will be aligned with the orientation of the region, if there is one (the default orientation pointing along the polyline).

property end

Get an *OrientedPoint* at the end of the polyline.

The OP's orientation will be aligned with the orientation of the region, if there is one (the default orientation pointing along the polyline).

signedDistanceTo(*point*)

Compute the signed distance of the PolylineRegion to a point.

The distance is positive if the point is left of the nearest segment, and negative otherwise.

Return type
float

pointAlongBy(*distance*, *normalized=False*)

Find the point a given distance along the polyline from its start.

If **normalized** is true, then distance should be between 0 and 1, and is interpreted as a fraction of the length of the polyline. So for example `pointAlongBy(0.5, normalized=True)` returns the polyline's midpoint.

Return type
Vector

class PointSetRegion(*name*, *points*, *kdTree=None*, *orientation=None*, *tolerance=1e-06*)

Bases: *Region*

Region consisting of a set of discrete points.

No *Object* can be contained in a *PointSetRegion*, since the latter is discrete. (This may not be true for subclasses, e.g. *GridRegion*.)

Parameters

- **name** (*str*) – name for debugging
- **points** (*arraylike*) – set of points comprising the region
- **kdTree** (scipy.spatial.KDTree, optional) – k-D tree for the points (one will be computed if none is provided)
- **orientation** (*VectorField*; optional) – preferred orientation for the region
- **tolerance** (*float; optional*) – distance tolerance for checking whether a point lies in the region

convertToFootprint(*region*)

Recursively convert a region into it's footprint.

For a polygonal region, returns the footprint. For composed regions, recursively reconstructs them using the footprints of their sub regions.

class GridRegion(*name*, *grid*, *Ax*, *Ay*, *Bx*, *By*, *orientation=None*)

Bases: *PointSetRegion*

A Region given by an obstacle grid.

A point is considered to be in a *GridRegion* if the nearest grid point is not an obstacle.

Parameters

- **name** (*str*) – name for debugging
- **grid** – 2D list, tuple, or NumPy array of 0s and 1s, where 1 indicates an obstacle and 0 indicates free space
- **Ax** (*float*) – spacing between grid points along X axis
- **Ay** (*float*) – spacing between grid points along Y axis
- **Bx** (*float*) – X coordinate of leftmost grid column
- **By** (*float*) – Y coordinate of lowest grid row
- **orientation** (*VectorField*; optional) – orientation of region

class ViewRegion(*visibleDistance*, *viewAngles*, *name=None*, *position=Vector(0, 0, 0)*, *rotation=None*, *orientation=None*, *angleCutoff=0.01*, *tolerance=1e-08*)

Bases: *MeshVolumeRegion*

The viewing volume of a camera defined by a radius and horizontal/vertical view angles.

The default view region can take several forms, depending on the viewAngles parameter:

- Case 1: viewAngles[1] = 180 degrees
  - Case 2.a viewAngles[0] = 360 degrees => Sphere

  - Case 2.b viewAngles[0] < 360 degrees => Sphere & CylinderSectionRegion
- Case 2: viewAngles[1] < 180 degrees
  - Case 2.a viewAngles[0] = 360 degrees => Sphere - (Cone + Cone) (Cones on z axis expanding from origin)
  - Case 2.b viewAngles[0] < 360 degrees => Sphere & ViewSectionRegion

When making changes to this class you should run `pytest -k test_viewRegion --exhaustive`.

Parameters

- **visibleDistance** – The view distance for this region.
- **viewAngles** – The view angles for this region.
- **name** – An optional name to help with debugging.
- **position** – An optional position, which determines where the center of the region will be.
- **rotation** – An optional Orientation object which determines the rotation of the object in space.
- **orientation** – An optional vector field describing the preferred orientation at every point in the region.
- **angleCutoff** – How close to 180/360 degrees an angle has to be to be mapped to that value.
- **tolerance** – Tolerance for collision computations.

scenic.core.requirements

Support for hard and soft requirements.

Summary of Module Members

Functions

getAllGlobals	Find all names the given lambda depends on, along with their current bindings.

Classes

BlanketCollisionRequirement	
BoundRequirement	
CompiledRequirement	
ContainmentRequirement	
DynamicMonitorRequirement	
DynamicRequirement	
IntersectionRequirement	
MonitorRequirement	MonitorRequirement is a BoundRequirement with temporal proposition monitor
NonVisibilityRequirement	
PendingRequirement	
RequirementType	An enumeration.
SamplingRequirement	A requirement to be checked to validate a sample.
VisibilityRequirement	

Member Details

class RequirementType(*value*)

Bases: Enum

An enumeration.

getAllGlobals(*req*, *restrictTo=None*)

Find all names the given lambda depends on, along with their current bindings.

class MonitorRequirement(*compiledReq*, *sample*, *proposition*)

Bases: BoundRequirement

MonitorRequirement is a BoundRequirement with temporal proposition monitor

class SamplingRequirement(*optional*)

Bases: ABC

A requirement to be checked to validate a sample.

Parameters

optional – Whether or not this requirement must be checked to validate the sample. Optional samples can be checked, and if False imply that the sample is invalid, but do not need to be checked if all non-optional requirements are satisfied.

abstract falsifiedByInner(*sample*)

Returns False if the requirement is falsifed, True otherwise

abstract property violationMsg

Message to be printed if the requirement is violated

scenic.core.sample_checking

The SampleChecker class and it's implementations.

Summary of Module Members

Classes

BasicChecker	Basic requirement checker.
SampleChecker	
WeightedAcceptanceChecker	Picks the requirement with the lowest time-weighted acceptance chance.

Member Details

class BasicChecker(*initialCollisionCheck*)

Bases: SampleChecker

Basic requirement checker.

Evaluates requirements in order, with a tiny bit of tuning.

class WeightedAcceptanceChecker(*bufferSize=10*)

Bases: SampleChecker

Picks the requirement with the lowest time-weighted acceptance chance.

Incentivizes exploration by initializing all buffer values to 0.

Parameters

bufferSize – Max samples to use when calculating time-weighted rejection chance.

sortedRequirements()

Return the list of requirements in sorted order

updateMetrics(*req*, *new_metrics*)

Update the metrics for a given requirement

scenic.core.scenarios

Scenario and scene objects.

Summary of Module Members

Classes

Scenario	A compiled Scenic scenario, from which scenes can be sampled.
Scene	A scene generated from a Scenic scenario.

Member Details

class Scene

A scene generated from a Scenic scenario.

To run a dynamic simulation from a scene, create an instance of *Simulator* for the simulator you want to use, and pass the scene to its *simulate* method.

Attributes

- **objects** (tuple of *Object*) – All objects in the scene. The ego object is first, if there is one.
- **egoObject** (*Object* or None) – The ego object, if any.
- **params** (*dict*) – Dictionary mapping the name of each global parameter to its value.
- **workspace** (*Workspace*) – The workspace for the scenario.

Changed in version 3.0: The egoObject attribute can now be None.

dumpAsScenicCode(*stream=sys.stdout*)

Dump Scenic code reproducing this scene to the given stream.

For non-human-readable but complete serialization of scenes see *Scenario.sceneToBytes* and *Scenario.sceneFromBytes*.

Note: This function does not currently reproduce parts of the original Scenic program defining behaviors, functions, etc. used in the scene. Also, if the scene involves any user-defined types, they must provide a suitable __repr__ for this function to print them properly.

Parameters

stream (text file) – Where to print the code (default sys.stdout).

show3D(*axes*)

Render a 3D schematic of the scene for debugging.

show2D(*zoom=None*, *block=True*)

Render a 2D schematic of the scene for debugging.

class Scenario

A compiled Scenic scenario, from which scenes can be sampled.

generate(*maxIterations=2000*, *verbosity=0*, *feedback=None*)

Sample a *Scene* from this scenario.

For a description of how scene generation is done, see *Scene Generation*.

Parameters

- **maxIterations** (*int*) – Maximum number of rejection sampling iterations.
- **verbosity** (*int*) – Verbosity level.
- **feedback** (*float*) – Feedback to pass to external samplers doing active sampling. See *scenic.core.external_params*.

Returns
A pair with the sampled *Scene* and the number of iterations used.

Raises
RejectionException – if no valid sample is found in **maxIterations** iterations.

generateBatch(*numScenes*, *maxIterations=inf*, *verbosity=0*, *feedback=None*)

Sample several *Scene* objects from this scenario.

For a description of how scene generation is done, see *Scene Generation*.

Parameters

- **numScenes** (*int*) – Number of scenes to generate.
- **maxIterations** (*int*) – Maximum number of rejection sampling iterations (over all scenes).
- **verbosity** (*int*) – Verbosity level.
- **feedback** (*float*) – Feedback to pass to external samplers doing active sampling. See *scenic.core.external_params*.

Returns
A pair with a list of the sampled *Scene* objects and the total number of iterations used.

Raises
RejectionException – if not enough valid samples are found in **maxIterations** iterations.

resetExternalSampler()

Reset the scenario's external sampler, if any.

If the Python random seed is reset before calling this function, this should cause the sequence of generated scenes to be deterministic.

conditionOn(*scene=None*, *objects=()*, *params={}*)

Condition the scenario on particular values for some objects or parameters.

This method changes the distribution of the scenario and should be used with care: it does not attempt to check that the new distribution is equivalent to the old one or that it has nonzero probability of satisfying the scenario's requirements.

For example, to sample object #5 in the scenario once and then leave it fixed in all subsequent samples:

```
sceneA, _ = scenario.generate()
scenario.conditionOn(scene=sceneA, objects=(5,))
sceneB, _ = scenario.generate()     # will have the same object 5 as sceneA
```

Parameters

- **scene** (Scene) – Scene from which to take values for the given **objects**, if any.
- **objects** – Sequence of indices specifying which objects in this scenario should be conditioned on the corresponding objects in **scene** (i.e. those with the same index in the list of objects).

- **params** (*dict*) – Dictionary of global parameters to condition and their new values (which may be constants or distributions).

sceneToBytes(*scene*, *allowPickle=False*)

Encode a *Scene* sampled from this scenario to a bytes object.

The serialized scene may be reconstituted with *sceneFromBytes*. The format used is suitable for long-term storage of scenes, although it is not guaranteed to be compatible across major versions of Scenic. For further discussion and usage examples, see *Storing Scenes/Simulations for Later Use*.

Raises

SerializationError – if the scene could not be properly encoded. This should not happen unless your scenario includes a user-defined *Distribution* subclass with an unusual value type. If you get this exception, see the documentation for the internal class *Serializer* for solutions.

sceneFromBytes(*data*, *verify=True*, *allowPickle=False*)

Decode a *Scene* serialized with *sceneToBytes*.

Parameters

- **data** (*bytes*) – Encoding of a *Scene* sampled from this scenario.
- **verify** (*bool*) – If true (the default), raise an exception if the scene appears to have been generated from a different scenario (meaning it will almost certainly not decode correctly).
- **allowPickle** (*bool*) – Enable using pickle to deserialize custom object types. False by default because it allows malicious data to trigger arbitrary code execution (see the pickle documentation). Use this option only if you trust the source of the data and it is not practical to implement serialization for the datatypes you need.

Raises

SerializationError – if the scene could not be properly decoded.

simulationToBytes(*simulation*, *allowPickle=False*)

Encode a *Simulation* sampled from this scenario to a bytes object.

The serialized simulation may be replayed with *simulationFromBytes*. As with *sceneToBytes*, the format used is suitable for long-term storage but is not guaranteed to be compatible across major versions of Scenic.

Raises

SerializationError – if the simulation could not be properly encoded. This should not happen unless your scenario includes a user-defined *Distribution* subclass with an unusual value type. If you get this exception, see the documentation for the internal class *Serializer* for solutions.

Note: The returned data encodes both the scene comprising the initial condition for the simulation and the simulation itself. If you will be running many simulations starting from the same scene, you can save space by separately encoding the scene and the various simulations: use *sceneToBytes* and *Simulation.getReplay* for encoding, and the **replay** argument of *Simulator.simulate* for decoding.

---

simulationFromBytes(*data*, *simulator*, *, *verify=True*, *allowPickle=False*, **kwargs*)

Replay a *Simulation* serialized with *simulationToBytes*.

Parameters

- **data** (*bytes*) – Encoding of a *Simulation* sampled from this scenario.

- **simulator** (Simulator) – Simulator in which to run the simulation. Using a different simulator configuration than that used for the original simulation may cause errors or unexpected behavior. If you need to do this, see the **enableDivergenceCheck** option of *Simulator.simulate*.
- **verify** (*bool*) – As in *sceneFromBytes*.
- **allowPickle** (*bool*) – As in *sceneFromBytes*.
- **kwargs** – All additional keyword arguments are passed through to the simulator; see *Simulator.simulate* for the available configuration options.

Returns

A *Simulation* object representing the completed simulation.

Raises

- ***SerializationError*** – if the simulation could not be properly decoded.
- ***DivergenceError*** – if the replayed simulation has diverged from the original (requires the original to have been run with divergence-checking support; see *Simulator.simulate*).

scenic.core.serialization

Utilities to help serialize Scenic objects.

The functions in this module usually do not need to be used directly. For high-level serialization APIs, see *Scenario.sceneToBytes*, *Scenario.simulationToBytes*, and *Scene.dumpAsScenicCode*.

Summary of Module Members

Functions

dumpAsScenicCode	Utility function to help export Scenic objects as Scenic code.
readBool	
readBytes	
readFloat	
readInt	
readStr	
scenicToJSON	Utility function to help serialize Scenic objects to JSON.
writeBool	
writeBytes	
writeFloat	
writeInt	
writeStr	

Classes

Serializer	Class for (de)serializing scenes, etc.

Exceptions

SerializationError	An error occurring during serialization/deserialization of Scenic objects.

Member Details

scenicToJSON(*obj*)

Utility function to help serialize Scenic objects to JSON.

Suitable for passing as the `default` argument to `json.dump`. At the moment this only supports very basic types like scalars and vectors: it does not allow encoding of an entire *Object*.

dumpAsScenicCode(*value*, *stream*)

Utility function to help export Scenic objects as Scenic code.

exception SerializationError

Bases: `Exception`

An error occurring during serialization/deserialization of Scenic objects.

class Serializer(*data=b''*, *allowPickle=False*, *detectEnd=False*)

Class for (de)serializing scenes, etc.

Ordinary Scenic users do not need to know about this class: they can use public APIs such as *Scenario.sceneToBytes*. If you have defined a custom type of *Distribution* whose **valueType** isn't one of the types used by the built-in primitive distributions (i.e. `int`, `float`, *Vector*), read on.

The sampled value of a *Distribution* is encoded as follows:

1. If the *Distribution* is *_deterministic*, recursively encode the sampled values of its dependencies.
2. If its *valueType* is a type for which we have a "codec" (like `int` or `float`), use the encoding function provided by the codec.
3. If the *valueType* provides a `encodeTo` method, use that.
4. If the user has allowed the use of `pickle`, pickle the value.
5. Otherwise raise a *SerializationError*.

Thus, you need only extend the serialization mechanism if your *Distribution* cannot be made deterministic (by adding appropriate dependencies with simpler valueTypes) and it has an unusual **valueType**. In that case, it's best to have your **valueType** implement `encodeTo` and `decodeFrom` methods: see *Vector* for example. If for some reason you can't add those methods to the class in question, you can use *Serializer.addCodec* to register encoder/decoder functions. Finally, if you're only using serialization internally and aren't concerned about security issues or making the encoding as compact as possible, you can turn on the **allowPickle** option: this will use `pickle` to encode any objects for which no specialized encoder is known.

classmethod sceneFormatVersion()

Current version of the *Scene* serialization format.

Must be incremented if the *writeScene* method or any of its helper methods (e.g. *writeValue*) change, or if a new codec is added.

classmethod replayFormatVersion()

Current version of the *Simulation* replay serialization format.

Must be incremented if the *writeReplayHeader* or *writeValue* methods change, or if a new codec is added.

writeScene(*scenario*, *scene*)

Serialize a *Scene*.

writeReplayHeader(*flags*)

Begin the encoding of a *Simulation* replay.

classmethod addCodec(*ty*, *encoder*, *decoder*)

Register encoder and decoder functions for the given type.

The encoder function should have signature `encoder(value, stream)` with *stream* a binary file-like object. The decoder function should have signature `decoder(stream)` and return the decoded value.

writeValue(*value*, *ty*)

Serialize a value of the given type.

scenic.core.shapes

Module containing the Shape class and its subclasses, which represent shapes of Objects

Summary of Module Members

Classes

BoxShape	A box shape with all dimensions 1 by default.
ConeShape	A cone shape with all dimensions 1 by default.
CylinderShape	A cylinder shape with all dimensions 1 by default.
MeshShape	A Shape subclass defined by a `trimesh.base.Trimesh` object.
Shape	An abstract base class for Scenic shapes.
SpheroidShape	A spheroid shape with all dimensions 1 by default.

Member Details

class Shape(*dimensions*, *scale*)

Bases: `ABC`

An abstract base class for Scenic shapes.

Represents a physical shape in Scenic. Does not encode position or orientation, which are handled by the *Region* class. Does contain dimension information, which is used as a default value by any *Object* with this shape and can be overwritten.

If dimensions and scale are both specified the dimensions are first set by dimensions, and then scaled by scale.

Parameters

- **dimensions** – The raw (before scaling) dimensions of the shape.
- **scale** – Scales all the dimensions of the shape by a multiplicative factor.

property containsCenter

Whether or not this object contains its central point

class MeshShape(*mesh*, *dimensions=None*, *scale=1*, *initial_rotation=None*)

Bases: *Shape*

A Shape subclass defined by a `trimesh.base.Trimesh` object.

The mesh passed must be a `trimesh.base.Trimesh` object that represents a well defined volume (i.e. the `is_volume` property must be true), meaning the mesh must be watertight, have consistent winding and have outward facing normals.

Parameters

- **mesh** – A mesh object.
- **dimensions** – The raw (before scaling) dimensions of the shape. If dimensions and scale are both specified the dimensions are first set by dimensions, and then scaled by scale.
- **scale** – Scales all the dimensions of the shape by a multiplicative factor. If dimensions and scale are both specified the dimensions are first set by dimensions, and then scaled by scale.

- **initial_rotation** – A 3-tuple containing the yaw, pitch, and roll respectively to apply when loading the mesh. Note the initial_rotation must be fixed.

classmethod fromFile(*path*, *filetype=None*, *compressed=None*, *binary=False*, ***kwargs*)

Load a mesh shape from a file, attempting to infer filetype and compression.

For example: "foo.obj.bz2" is assumed to be a compressed .obj file. "foo.obj" is assumed to be an uncompressed .obj file. "foo" is an unknown filetype, so unless a filetype is provided an exception will be raised.

Parameters

- **path** (*str*) – Path to the file to import.
- **filetype** (*str*) – Filetype of file to be imported. This will be inferred if not provided. The filetype must be one compatible with `trimesh.load`.
- **compressed** (*bool*) – Whether or not this file is compressed (with bz2). This will be inferred if not provided.
- **binary** (*bool*) – Whether or not to open the file as a binary file.
- **kwargs** – Additional arguments to the MeshShape initializer.

class BoxShape(*dimensions=(1, 1, 1)*, *scale=1*, *initial_rotation=None*)

Bases: *MeshShape*

A box shape with all dimensions 1 by default.

class CylinderShape(*dimensions=(1, 1, 1)*, *scale=1*, *initial_rotation=None*, *sections=24*)

Bases: *MeshShape*

A cylinder shape with all dimensions 1 by default.

class ConeShape(*dimensions=(1, 1, 1)*, *scale=1*, *initial_rotation=None*)

Bases: *MeshShape*

A cone shape with all dimensions 1 by default.

class SpheroidShape(*dimensions=(1, 1, 1)*, *scale=1*, *initial_rotation=None*)

Bases: *MeshShape*

A spheroid shape with all dimensions 1 by default.

scenic.core.simulators

Interface between Scenic and simulators.

This module defines the core classes *Simulator* and *Simulation* which orchestrate dynamic simulations. Each simulator interface defines subclasses of these classes for their particular simulator.

Ordinary Scenic users only need to know about the top-level simulation API *Simulator.simulate* and the attributes of the *Simulation* class (in particular the `result` attribute, which captures information about the result of the simulation as a *SimulationResult* object).

Summary of Module Members

Classes

Action	An action which can be taken by an agent for one step of a simulation.
DummySimulation	Minimal *Simulation* subclass for *DummySimulator*.
DummySimulator	Simulator which does (almost) nothing, for testing and debugging purposes.
EndScenarioAction	Special action indicating it is time to end the current scenario.
EndSimulationAction	Special action indicating it is time to end the simulation.
ReplayMode	An enumeration.
Simulation	A single simulation run.
SimulationResult	Result of running a simulation.
Simulator	A simulator which can execute dynamic simulations from Scenic scenes.
TerminationType	Enum describing the possible ways a simulation can end.

Exceptions

DivergenceError	Exception indicating simulation replay failed due to simulator nondeterminism.
RejectSimulationException	Exception indicating a requirement was violated at runtime.
SimulationCreationError	Exception indicating a simulation could not be run from the given scene.
SimulatorInterfaceWarning	Warning indicating an issue with the interface to an external simulator.

Member Details

exception SimulatorInterfaceWarning

Bases: UserWarning

Warning indicating an issue with the interface to an external simulator.

exception SimulationCreationError

Bases: Exception

Exception indicating a simulation could not be run from the given scene.

Can also be issued during a simulation if dynamic object creation fails.

exception DivergenceError

Bases: Exception

Exception indicating simulation replay failed due to simulator nondeterminism.

exception RejectSimulationException

Bases: Exception

Exception indicating a requirement was violated at runtime.

class Simulator

Bases: ABC

A simulator which can execute dynamic simulations from Scenic scenes.

Simulator interfaces which support dynamic simulations should implement a subclass of *Simulator*. An instance of the class represents a connection to the simulator suitable for running multiple simulations (not necessarily of the same Scenic program). For a simple example of how to implement this class, and its counterpart *Simulation* for individual simulations, see *scenic.simulators.lgsvl.simulator*.

Users who create an instance of *Simulator* should call its *destroy* method when they are finished running simulations to allow the interface to do any necessary cleanup.

simulate(*scene*, *maxSteps=None*, *maxIterations=1*, ***, *timestep=None*, *verbosity=None*, *raiseGuardViolations=False*, *replay=None*, *enableReplay=True*, *enableDivergenceCheck=False*, *divergenceTolerance=0*, *continueAfterDivergence=False*, *allowPickle=False*)

Run a simulation for a given scene.

For details on how simulations are run, see *Execution of Dynamic Scenarios*.

Parameters

- **scene** (Scene) – Scene from which to start the simulation (sampled using *Scenario.generate*).
- **maxSteps** (*int*) – Maximum number of time steps for the simulation, or None to not impose a time bound.
- **maxIterations** (*int*) – Maximum number of rejection sampling iterations.
- **timestep** (*float*) – Length of a time step in seconds, or None to use a default provided by the simulator interface. Some interfaces may not allow arbitrary time step lengths or may require the timestep to be set when creating the *Simulator* and not customized per-simulation.
- **verbosity** (*int*) – If not None, override Scenic's global verbosity level (from the *--verbosity* option or *scenic.setDebuggingOptions*).
- **raiseGuardViolations** (*bool*) – Whether violations of preconditions/invariants of scenarios/behaviors should cause this method to raise an exception, instead of only rejecting the simulation (the default behavior).
- **replay** (*bytes*) – If not None, must be replay data output by *Simulation.getReplay*: we will then replay the saved simulation rather than randomly generating one as usual. If **maxSteps** is larger than that of the original simulation, then once the replay is exhausted the simulation will continue to run in the usual randomized manner.
- **enableReplay** (*bool*) – Whether to save data from the simulation so that it can be serialized for later replay using *Scenario.simulationToBytes* or *Simulation.getReplay*. Enabled by default as the overhead is generally low.
- **enableDivergenceCheck** (*bool*) – Whether to save the values of every dynamic property at each time step, so that when the simulation is replayed, nondeterminism in the simulator (or replaying the simulation in the wrong simulator) can be detected. Disabled by default as this option greatly increases the size of replay objects (~100 bytes per object per step).
- **divergenceTolerance** (*float*) – Amount by which a dynamic property can deviate in a replay from its original value before we consider the replay to have diverged. The default value is zero: no deviation is allowed. If finer control over divergences is required, see *Simulation.valuesHaveDiverged*.

- **continueAfterDivergence** (*bool*) – Whether to continue simulating after a divergence is detected instead of raising a *DivergenceError*. If this is true, then a divergence ends the replaying of the saved scenario but the simulation will continue in the usual randomized manner (i.e., it is as if the replay data ran out at the moment of the divergence).
- **allowPickle** (*bool*) – Whether to use `pickle` to (de)serialize custom object types. See *sceneFromBytes* for a discussion of when this may be needed (rarely) and its security implications.

Returns

A *Simulation* object representing the completed simulation, or `None` if no simulation satisfying the requirements could be found within **maxIterations** iterations.

Raises

- ***SimulationCreationError*** – if an error occurred while trying to run a simulation (e.g. some assumption made by the simulator was violated, like trying to create an object inside another).
- ***GuardViolation*** – if **raiseGuardViolations** is true and a precondition or invariant was violated during the simulation.
- ***DivergenceError*** – if replaying a simulation (via the **replay** option) and the replay has diverged from the original; requires the original simulation to have been run with **enableDivergenceCheck**.
- ***SerializationError*** – if writing or reading replay data fails. This could happen if your scenario uses an unusual custom distribution (see *sceneToBytes*) or if the replayed scenario has diverged without divergence-checking enabled.

Changed in version 3.0: **maxIterations** is now 1 by default.

New in version 3.0: The **timestep** argument.

replay(*scene*, *replay*, ***kwargs*)

Replay a simulation.

This convenience method simply calls *simulate* (and so takes all the same arguments), but makes the **replay** argument positional so you can write `simulator.replay(scene, replay)` instead of `simulator.simulate(scene, replay=replay)`.

abstract createSimulation(*scene*, ***kwargs*)

Create a *Simulation* from a Scenic scene.

This should be overridden by subclasses to return instances of their own specialized subclass of *Simulation*. The given **scene** and **kwargs** (together making up all the arguments passed to *simulate* except for **maxIterations**) should be passed through to the initializer of that instance.

Changed in version 3.0: This method is now called with all the arguments to *simulate* except for **maxIterations**; these should be passed through as described above.

destroy()

Clean up as needed when shutting down the simulator interface.

Subclasses should call the parent implementation, which will catch this method being called twice on the same *Simulator*.

class Simulation(*scene*, *, *maxSteps*, *name*, *timestep*, *replay=None*, *enableReplay=True*, *allowPickle=False*, *enableDivergenceCheck=False*, *divergenceTolerance=0*, *continueAfterDivergence=False*, *verbosity=0*)

Bases: `ABC`

A single simulation run.

These objects are not manipulated manually, but are created by a *Simulator*. Simulator interfaces should subclass this class, implementing various abstract methods to call the appropriate simulator APIs. In particular, the following methods must be implemented:

- *createObjectInSimulator*, to create an object;
- *step*, to run the simulation for one time step;
- *getProperties*, to read back the new state of an object.

Other methods can be overridden if necessary, e.g. *setup* for initialization at the start of the simulation and *destroy* for cleanup afterward.

Changed in version 3.0: The `__init__` method of subclasses should no longer create objects; the *createObjectInSimulator* method will be called instead. Other initialization which needs to take place after object creation should be done in *setup* after calling the superclass implementation.

The arguments to `__init__` are the same as those to *simulate*, except that `maxIterations` is omitted.

Attributes

- **currentTime** (*int*) – Number of time steps elapsed so far.
- **timestep** (*float*) – Length of each time step in seconds.
- **objects** – List of Scenic objects (instances of *Object*) existing in the simulation. This list will change if objects are created dynamically.
- **agents** – List of agents in the simulation.
- **result** (*SimulationResult*) – Result of the simulation, or `None` if it has not yet completed. This is the primary object which should be inspected to get data out of the simulation: the other undocumented attributes of this class are for internal use only.

Raises

RejectSimulationException – if a requirement is violated.

setup()

Set up the simulation to run in the simulator.

Subclasses may override this method to perform custom initialization, but should call the parent implementation to create the objects in the initial scene (through *createObjectInSimulator*).

abstract createObjectInSimulator(*obj*)

Create the given object in the simulator.

Implemented by subclasses. Should raise *SimulationCreationError* if creating the object fails.

Parameters

obj (`Object`) – the Scenic object to create.

Raises

SimulationCreationError – if unable to create the object in the simulator.

scheduleForAgents()

Compute the order for the agents to run in the next time step.

The default order is the order in which the agents were created.

Returns

An iterable which is a permutation of `self.agents`.

actionsAreCompatible(*agent*, *actions*)

Check whether the given actions can be taken simultaneously by an agent.

The default is to consider all actions compatible with each other, and to call *Action.canBeTakenBy* to determine if an agent can take an action. Subclasses should override this method as appropriate.

Parameters

- **agent** (Object) – the agent which wants to take the given actions.
- **actions** (*tuple*) – tuple of actions to be taken.

executeActions(*allActions*)

Execute the actions selected by the agents.

The default implementation calls the *applyTo* method of each Action to apply it to the appropriate agent. Subclasses may override this method to make additional simulator API calls as needed, but should call this implementation too or otherwise emulate its functionality.

Parameters

allActions – an OrderedDict mapping each agent to a tuple of actions. The order of agents in the dict should be respected in case the order of actions matters.

abstract step()

Run the simulation for one step and return the next trajectory element.

Implemented by subclasses. This should cause the simulator to simulate physics for `self.timestep` seconds.

updateObjects()

Update the positions and other properties of objects from the simulation.

Subclasses likely do not need to override this method: they should implement its subroutine *getProperties* below.

valuesHaveDiverged(*obj*, *prop*, *expected*, *actual*)

Decide whether the value of a dynamic property has diverged from the replay.

The default implementation considers scalar and vector properties to have diverged if the distance between the actual and expected values is greater than `self.divergenceTolerance` (which is 0 by default); other types of properties use the != operator.

Subclasses may override this function to provide more specialized criteria (e.g. allowing some properties to diverge more than others).

Parameters

- **obj** (Object) – The object being considered.
- **prop** (*str*) – The name of the dynamic property being considered.
- **expected** – The value of the property saved in the replay currently being run.
- **actual** – The value of the property in the current simulation.

Returns

True if the actual value should be considered as having diverged from the expected one; otherwise False.

abstract getProperties(*obj*, *properties*)

Read the values of the given properties of the object from the simulator.

Implemented by subclasses.

Parameters

- **obj** (Object) – Scenic object in question.
- **properties** (set) – Set of names of properties to read from the simulator. It is safe to destructively iterate through the set if you want.

Returns

A dict mapping each of the given properties to its current value.

currentState()

Return the current state of the simulation.

The definition of 'state' is up to the simulator; the 'state' is simply saved at each time step to define the 'trajectory' of the simulation.

The default implementation returns a tuple of the positions of all objects.

property currentRealTime

Current simulation time, in seconds.

destroy()

Perform any cleanup necessary to reset the simulator after a simulation.

The default implementation does nothing by default; it may be overridden by subclasses.

getReplay()

Encode this simulation to a bytes object for future replay.

Requires that the simulation was run with `enableReplay=True` (the default).

class ReplayMode(*value*)

Bases: IntFlag

An enumeration.

class DummySimulator(*drift=0*)

Bases: *Simulator*

Simulator which does (almost) nothing, for testing and debugging purposes.

To allow testing the change of dynamic properties over time, all objects drift upward by **drift** every time step.

class DummySimulation(*scene*, *drift=0*, ***kwargs*)

Bases: *Simulation*

Minimal *Simulation* subclass for *DummySimulator*.

class Action

Bases: ABC

An action which can be taken by an agent for one step of a simulation.

canBeTakenBy(*agent*)

Whether this action is allowed to be taken by the given agent.

The default implementation always returns True.

abstract applyTo(*agent*, *simulation*)

Apply this action to the given agent in the given simulation.

This method should call simulator APIs so that the agent will take this action during the next simulated time step. Depending on the simulator API, it may be necessary to batch each agent's actions into a

single update: in that case you can have this method set some state on the agent, then apply the actual update in an overridden implementation of *Simulation.executeActions*. For examples, see the CARLA interface: *scenic.simulators.carla.actions* has some CARLA-specific actions which directly call CARLA APIs, while the generic steering and braking actions from *scenic.domains.driving.actions* are implemented using the batching approach (see for example the `setThrottle` method of the class *scenic.simulators.carla.model.Vehicle*, which sets state later read by `CarlaSimulation.executeActions` in *scenic.simulators.carla.simulator*).

class EndSimulationAction(*line*)

Bases: *Action*

Special action indicating it is time to end the simulation.

Only for internal use.

class EndScenarioAction(*scenario*, *line*)

Bases: *Action*

Special action indicating it is time to end the current scenario.

Only for internal use.

class TerminationType(*value*)

Bases: Enum

Enum describing the possible ways a simulation can end.

timeLimit = 'reached simulation time limit'

Simulation reached the specified time limit.

scenarioComplete = 'the top-level scenario finished'

The top-level scenario finished executing.

(Either its *compose* block completed, one of its termination conditions was met, or it was terminated with *terminate*.)

simulationTerminationCondition = 'a simulation termination condition was met'

A user-specified simulation termination condition was met.

terminatedByMonitor = 'a monitor terminated the simulation'

A monitor used *terminate simulation* to end the simulation.

terminatedByBehavior = 'a behavior terminated the simulation'

A dynamic behavior used *terminate simulation* to end the simulation.

class SimulationResult(*trajectory*, *actions*, *terminationType*, *terminationReason*, *records*)

Result of running a simulation.

Attributes

- **trajectory** – A tuple giving for each time step the simulation's 'state': by default the positions of every object. See *Simulation.currentState*.
- **finalState** – The last 'state' of the simulation, as above.
- **actions** – A tuple giving for each time step a dict specifying for each agent the (possibly-empty) tuple of actions it took at that time step.
- **terminationType** (*TerminationType*) – The way the simulation ended.
- **terminationReason** (*str*) – A human-readable string giving the reason why the simulation ended, possibly including debugging info.

- **records** (*dict*) – For each *record* statement, the value or time series of values its expression took during the simulation.

scenic.core.specifiers

Specifiers and associated objects.

Summary of Module Members

Classes

ModifyingSpecifier	Specifier providing values (or modifying) properties.
PropertyDefault	A default value, possibly with dependencies.
Specifier	Specifier providing values for properties.

Member Details

class Specifier(*name*, *priorities*, *value*, *deps=None*)

Specifier providing values for properties.

Each property is set to a value, at a given priority, given dependencies.

Parameters

- **name** – The name of this specifier.
- **priorities** – A dictionary mapping properties to the priority they are being specified with.
- **value** – A dictionary mapping properties to the values they are being specified as.
- **deps** – An iterable containing all properties that this specifier relies on.

getValuesFor(*obj*)

Get the values specified for a given object.

class ModifyingSpecifier(*name*, *priorities*, *value*, *modifiable_props*, *deps=None*)

Bases: *Specifier*

Specifier providing values (or modifying) properties.

Parameters

- **name** – The name of this specifier.
- **priorities** – A dictionary mapping properties to the priority they are being specified with.
- **value** – A dictionary mapping properties to the values they are being specified as.
- **modifiable_props** – What properties specified by this specifier can be modified.
- **deps** – An iterable containing all properties that this specifier relies on.

class PropertyDefault(*requiredProperties*, *attributes*, *value*)

A default value, possibly with dependencies.

resolveFor(*prop*, *overriddenDefs*)

Create a Specifier for a property from this default and any superclass defaults.

scenic.core.type_support

Support for checking Scenic types.

This module provides a system for checking that values passed to Scenic operators and functions have the expected types. The top-level function *toTypes* and its specializations *toType*, *toVector*, *toScalar*, etc. can also *coerce* closely-related types into the desired type in some cases. For lazily-evaluated values (random values and delayed arguments of specifiers), it may not be possible to determine the type at object creation time: in such cases these functions return a lazily-evaluated object that performs the type check either during specifier resolution or sampling as needed.

In general, the only objects which are coercible to a type T are instances of that type, together with *Distribution* objects whose **_valueType** is a type coercible to T (and therefore whose sampled value can be coerced to T). However, we also have the following exceptional rules:

- **Coercible to a scalar (type `float`):**
  - Instances of `numbers.Real` (coerced by calling `float` on them); this includes NumPy types such as `numpy.single`
- **Coercible to a heading (type *Heading*):**
  - Anything coercible to a scalar
  - Any type with a **toHeading** method (including *OrientedPoint*)
- **Coercible to a vector (type *Vector*):**
  - Tuples and lists of length 2 or 3
  - Any type with a **toVector** method (including *Point*)
- **Coercible to a `Behavior`:**
  - Subclasses of `Behavior` (coerced by calling them with no arguments)
  - `None` (considered to have type `Behavior` for convenience)

Summary of Module Members

Functions

canCoerce	Can this value be coerced into the given type?
canCoerceType	Can values of typeA be coerced into typeB?
coerce	Coerce something into the given type.
coerceToAny	Coerce something into any of the given types, raising an error if impossible.
coerceToFloat	
coerceToHeading	
evaluateRequiringEqualTypes	Evaluate the func, assuming thingA and thingB have the same type.
isA	Is this guaranteed to evaluate to a member of the given Scenic type?
is_typing_generic	Whether this is a pre-3.9 generic type from the typing module.
toHeading	Convert something to a heading, raising an error if impossible.
toOrientation	Convert something to an orientation, raising an error if impossible.
toScalar	Convert something to a scalar, raising an error if impossible.
toType	Convert something to a given type, raising an error if impossible.
toTypes	Convert something to any of the given types, raising an error if impossible.
toVector	Convert something to a vector, raising an error if impossible.
underlyingType	What type this value ultimately evaluates to, if we can tell.
unifierOfTypes	Most specific type unifying the given types.
unifyingType	Most specific type unifying the given values.

Classes

Heading	Dummy class used as a target for type coercions to headings.
TypeChecker	Checks that a given lazy value has one of a given list of types.
TypeEqualityChecker	Evaluates a function after checking that two lazy values have the same type.
TypecheckedDistribution	Distribution which typechecks its value at sampling time.

Exceptions

CoercionFailure	Raised by coercion functions when coercion is impossible.

Member Details

class Heading(*x=0*, */*)

Bases: float

Dummy class used as a target for type coercions to headings.

underlyingType(*thing*)

What type this value ultimately evaluates to, if we can tell.

isA(*thing*, *ty*)

Is this guaranteed to evaluate to a member of the given Scenic type?

unifyingType(*opts*)

Most specific type unifying the given values.

unifierOfTypes(*types*)

Most specific type unifying the given types.

canCoerceType(*typeA*, *typeB*)

Can values of typeA be coerced into typeB?

canCoerce(*thing*, *ty*)

Can this value be coerced into the given type?

coerce(*thing*, *ty*, *error='wrong type'*)

Coerce something into the given type.

Used internally by *toType*, etc.; this function should not otherwise be called directly.

exception CoercionFailure

Bases: Exception

Raised by coercion functions when coercion is impossible.

Only used internally; will be converted to a parse error for reporting to the user.

class TypecheckedDistribution(*dist*, *ty*, *errorMessage*, *coercer=None*)

Bases: *Distribution*

Distribution which typechecks its value at sampling time.

Only for internal use by the typechecking system; introduced by *coerce* when it is unable to guarantee that a random value will have the correct type after sampling. Note that the type check is not a purely passive operation, and may actually transform the sampled value according to the coercion rules above (e.g. a sampled *Point* will be converted to a *Vector* in a context which expects the latter).

coerceToAny(*thing*, *types*, *error*)

Coerce something into any of the given types, raising an error if impossible.

Only for internal use by the typechecking system; called from *toTypes*.

Raises
TypeError – if it is impossible to coerce the value into any of the types.

toTypes(*thing*, *types*, *typeError='wrong type'*)

Convert something to any of the given types, raising an error if impossible.

Types are tried in the order they are given: the first one compatible with the given value is used. Coercions of closely-related types may take place as described in the module documentation above.

If the given value requires lazy evaluation, this function returns a *TypeChecker* object that performs the type conversion after specifier resolution.

Parameters

- **thing** – Value to convert.
- **types** – Sequence of one or more destination types.
- **typeError** (*str*) – Message included in exception raised on failure.

Raises
TypeError – if the given value is not one of the given types and cannot be converted to any of them.

toType(*thing*, *ty*, *typeError='wrong type'*)

Convert something to a given type, raising an error if impossible.

Equivalent to *toTypes* with a single destination type.

toScalar(*thing*, *typeError='non-scalar in scalar context'*)

Convert something to a scalar, raising an error if impossible.

See *toTypes* for details.

toHeading(*thing*, *typeError='non-heading in heading context'*)

Convert something to a heading, raising an error if impossible.

See *toTypes* for details.

toOrientation(*thing*, *typeError='non-orientation in orientation context'*)

Convert something to an orientation, raising an error if impossible.

See *toTypes* for details.

toVector(*thing*, *typeError='non-vector in vector context'*)

Convert something to a vector, raising an error if impossible.

See *toTypes* for details.

evaluateRequiringEqualTypes(*func*, *thingA*, *thingB*, *typeError='type mismatch'*)

Evaluate the func, assuming thingA and thingB have the same type.

If func produces a lazy value, it should not have any required properties beyond those of thingA and thingB.

Raises
TypeError – if thingA and thingB do not have the same type.

class TypeChecker(**args*, *_internal=False*, ***kwargs*)

Bases: *DelayedArgument*

Checks that a given lazy value has one of a given list of types.

class TypeEqualityChecker(**args*, *_internal=False*, ***kwargs*)

Bases: *DelayedArgument*

Evaluates a function after checking that two lazy values have the same type.

is_typing_generic(*tp*)

Whether this is a pre-3.9 generic type from the typing module.

scenic.core.utils

Assorted utility functions.

Summary of Module Members

Functions

alarm	
argsToString	
batched	
cached	Decorator for making a method with no arguments cache its result
cached_method	Decorator for making a method cache its result on a per-object basis.
cached_property	
loadMesh	

Classes

DefaultIdentityDict	Dictionary which is the identity map by default.

Member Details

cached(*oldMethod*)

Decorator for making a method with no arguments cache its result

cached_method(*oldMethod*)

Decorator for making a method cache its result on a per-object basis.

Like `functools.lru_cache(maxsize=None)` except using a separate cache for each object, with the cache automatically deallocated when the object is garbage collected.

class DefaultIdentityDict

Dictionary which is the identity map by default.

The map works on all objects, even unhashable ones, but doesn't support all of the standard mapping operations.

scenic.core.vectors

Scenic vectors and vector fields.

Summary of Module Members

Functions

alwaysGlobalOrientation	Whether this orientation is always aligned with the global coordinate system.
makeVectorOperatorHandler	
scalarOperator	Decorator for vector operators that yield scalars.
vectorDistributionMethod	Decorator for methods that produce vectors.
vectorOperator	Decorator for vector operators that yield vectors.
zeroIdentityVectorOperator	
zeroPreservingVectorOperator	

Classes

Orientation	An orientation in 3D space.
OrientedVector	
PiecewiseVectorField	A vector field defined by patching together several regions.
PolygonalVectorField	A piecewise-constant vector field defined over polygonal cells.
PolyhedronVectorField	
Vector	A 3D vector, whose coordinates can be distributions.
VectorDistribution	A distribution over Vectors.
VectorField	A vector field, providing an orientation at every point.
VectorMethodDistribution	Vector version of MethodDistribution.
VectorOperatorDistribution	Vector version of OperatorDistribution.

Member Details

class VectorDistribution(**dependencies*, *valueType=None*)

Bases: *Distribution*

A distribution over Vectors.

_defaultValueType

alias of *Vector*

class VectorOperatorDistribution(*operator*, *obj*, *operands*)

Bases: *VectorDistribution*

Vector version of OperatorDistribution.

class VectorMethodDistribution(*method*, *obj*, *args*, *kwargs*)

Bases: *VectorDistribution*

Vector version of MethodDistribution.

scalarOperator(*method*)

Decorator for vector operators that yield scalars.

vectorOperator(*method*, *preservesZero=False*, *zeroIdentity=False*)

Decorator for vector operators that yield vectors.

vectorDistributionMethod(*method*)

Decorator for methods that produce vectors. See distributionMethod.

class Orientation(*rotation*)

An orientation in 3D space.

classmethod fromQuaternion(*quaternion*)

Create an *Orientation* from a quaternion (of the form (x,y,z,w))

Return type
Orientation

classmethod fromEuler(*yaw*, *pitch*, *roll*)

Create an *Orientation* from yaw, pitch, and roll angles (in radians).

Return type
Orientation

property yaw: float

Yaw in the global coordinate system.

property pitch: float

Pitch in the global coordinate system.

property roll: float

Roll in the global coordinate system.

property eulerAngles: Tuple[float, float, float]

Global intrinsic Euler angles yaw, pitch, roll.

localAnglesFor(*orientation*)

Get local Euler angles for an orientation w.r.t. this orientation.

That is, considering `self` as the parent orientation, find the Euler angles expressing the given orientation.

Return type
Tuple[float, float, float]

globalToLocalAngles(*yaw*, *pitch*, *roll*)

Convert global Euler angles to local angles w.r.t. this orientation.

Equivalent to *localAnglesFor* but takes Euler angles as input.

Return type
Tuple[float, float, float]

alwaysGlobalOrientation(*orientation*)

Whether this orientation is always aligned with the global coordinate system.

Returns False if the orientation is a distribution or delayed argument, since then the value cannot be known at this time.

class Vector(*x*, *y*, *z=0*)

Bases: *Samplable*, Sequence

A 3D vector, whose coordinates can be distributions.

sphericalCoordinates()

Returns this vector in spherical coordinates (rho, theta, phi)

rotatedBy(*angleOrOrientation*)

Return a vector equal to this one rotated counterclockwise by angle/orientation.

Return type
Vector

angleWith(*other*)

Compute the signed angle between self and other.

The angle is positive if other is counterclockwise of self (considering the smallest possible rotation to align them).

Return type
float

class VectorField(*name*, *value*, *minSteps=4*, *defaultStepSize=5*)

A vector field, providing an orientation at every point.

Parameters

- **name** (*str*) – name for debugging.
- **value** – function computing the heading at the given *Vector*.
- **minSteps** (*int*) – Minimum number of steps for *followFrom*; default 4.
- **defaultStepSize** (*float*) – Default step size for *followFrom*; default 5. This is an upper bound: more steps will be taken as needed to ensure that no single step is longer than this value, but if the distance to travel is small then the steps may be smaller.

followFrom(*pos*, *dist*, *steps=None*, *stepSize=None*)

Follow the field from a point for a given distance.

Uses the forward Euler approximation, covering the given distance with equal-size steps. The number of steps can be given manually, or computed automatically from a desired step size.

Parameters

- **pos** (*Vector*) – point to start from.
- **dist** (*float*) – distance to travel.
- **steps** (*int*) – number of steps to take, or None to compute the number of steps based on the distance (default None).
- **stepSize** (*float*) – length used to compute how many steps to take, or None to use the field's default step size.

static forUnionOf(*regions*, *tolerance=0*)

Creates a *PiecewiseVectorField* from the union of the given regions.

If none of the regions have an orientation, returns None instead.

class PolygonalVectorField(*name*, *cells*, *headingFunction=None*, *defaultHeading=None*)

Bases: *VectorField*

A piecewise-constant vector field defined over polygonal cells.

Parameters

- **name** (*str*) – name for debugging.
- **cells** – a sequence of cells, with each cell being a pair consisting of a Shapely geometry and a heading. If the heading is None, we call the given **headingFunction** for points in the cell instead.
- **headingFunction** – function computing the heading for points in cells without specified headings, if any (default None).
- **defaultHeading** – heading for points not contained in any cell (default None, meaning reject such points).

class PiecewiseVectorField(*name*, *regions*, *tolerance=0*, *defaultHeading=None*)

Bases: *VectorField*

A vector field defined by patching together several regions.

The heading at a point is determined by checking each region in turn to see if it has an orientation and contains the point, returning the corresponding heading if so. If we get through all the regions, and **tolerance** is nonzero, we try again, this time allowing the point to be up to **tolerance** away from each region. If we still fail to find a region "containing" the point, then we return the **defaultHeading**, if any, and otherwise reject the scene.

Parameters

- **name** (*str*) – name for debugging.
- **regions** (sequence of *Region* objects) – the regions making up the field.
- **tolerance** (*float*) – maximum distance at which to consider a point as being in one of the regions, if it is not otherwise contained (default 0).
- **defaultHeading** (*float*) – the heading for points not in any region with an orientation (default None, meaning reject such points).

scenic.core.visibility

Implementations of Scenic's visibility functions.

Summary of Module Members

Functions

canSee	Perform visibility checks on Points, OrientedPoints, or Objects, accounting for occlusion.

Member Details

canSee(*position*, *orientation*, *visibleDistance*, *viewAngles*, *rayCount*, *rayDensity*, *distanceScaling*, *target*, *occludingObjects*, *debug=False*)

Perform visibility checks on Points, OrientedPoints, or Objects, accounting for occlusion.

For visibilty of Objects:

1. Do several quick checks to see if the object is naively visible or not visible:
   - If the object contains its position and its position is visible, the object is visible.
   - If the viewer is inside the object, the object is visible.
   - If the closest distance from the object to the viewer is greater than the visible distance, the object is not visible.
2. Check if the object crosses the back and/or front of the viewing object.
3. Compute the spherical coordinates of all vertices in the mesh of the region we are trying to view, with the goal of using this to send rays only where they have a chance of hitting the region.
4. Compute 2 ranges of angles (horizontal/vertical) in which rays have a chance of hitting the object, as follows:
   - If the object does not cross behind the viewer, take the min and max of the the spherical coordinate angles, while noting that this range is centered on the front of the viewer.
   - If the object crosses behind the viewer but not in front, transform the spherical angles so they are coming from the back of the object, while noting that this range is centered on the back of the object.
   - If it crosses both, we do not optimize the amount of rays sent.
5. Compute the intersection of the optimizated range from step 4 and the viewAngles range, accounting for where the optimization range is centered. If it is empty, the object cannot be visible. If it is not empty, shoot rays at the desired density in the intersection region. Keep all rays that intersect the object (candidate rays).
6. If there are no candidate rays, the object is not visible.
7. For each occluding object in occludingObjects: check if any candidate rays intersect the occluding object at a distance less than the distance they intersected the target object. If they do, remove them from the candidate rays.
8. If any candidate rays remain, the object is visible. If not, it is occluded and not visible.

For visibility of Points/OrientedPoints:

1. Check if distance from the viewer to the point is greater than visibleDistance. If so, the point cannot be visible
2. Create a single candidate ray, using the vector from the viewer to the target. If this ray is outside of the bounds of viewAngles, the point cannot be visible.

3. For each occluding object in occludingObjets: check if the candidate ray hits the occluding object at a distance less than the distance from the viewer to the target point. If so, then the object is not visible. Otherwise, the object is visible.

Parameters

- **position** – Position of the viewer, accounting for any offsets.
- **orientation** – Orientation of the viewer.
- **visibleDistance** – The maximum distance the viewer can view objects from.
- **viewAngles** – The horizontal and vertical view angles, in radians, of the viewer.
- **rayCount** – The total number of rays in each dimension used in visibility calculations..
- **target** – The target being viewed. Currently supports Point, OrientedPoint, and Object.
- **occludingObjects** – An optional list of objects which can occlude the target.

scenic.core.workspaces

Workspaces.

Summary of Module Members

Classes

Workspace	A workspace describing the fixed world of a scenario.

Member Details

class Workspace(*region=<AllRegion everywhere>*)

Bases: *Region*

A workspace describing the fixed world of a scenario.

Parameters
region (Region) – The region defining the extent of the workspace (default *everywhere*).

show3D(*viewer*)

Render a schematic of the workspace (in 3D) for debugging

show2D(*plt*)

Render a schematic of the workspace (in 2D) for debugging

zoomAround(*plt*, *objects*, *expansion=1*)

Zoom the schematic around the specified objects

scenicToSchematicCoords(*coords*)

Convert Scenic coordinates to those used for schematic rendering.

scenic.domains

General scenario domains used across simulators.

driving	Domain for driving scenarios.

scenic.domains.driving

Domain for driving scenarios.

The *world model* defines Scenic classes for cars, pedestrians, etc., actions for dynamic agents which walk or drive, as well as simple behaviors like lane-following. Scenarios for the driving domain should import the model as follows:

```
model scenic.domains.driving.model
```

Scenarios written for the driving domain should work without changes[1] in any of the following simulators:

- CARLA, using the model *scenic.simulators.carla.model*
- LGSVL, using the model *scenic.simulators.lgsvl.model*
- the built-in Newtonian simulator, using the model *scenic.simulators.newtonian.driving_model*

For example, the `examples/driving/badlyParkedCarPullingIn.scenic` scenario is written for the driving domain and can be run in multiple simulators:

- no simulator, for viewing the initial scene:

```
$ scenic examples/driving/badlyParkedCarPullingIn.scenic
```

- the built-in Newtonian simulator, for quick debugging without having to install an external simulator:

```
$ scenic -S --model scenic.simulators.newtonian.driving_model \
    examples/driving/badlyParkedCarPullingIn.scenic
```

- CARLA, using the default map specified in the scenario:

```
$ scenic -S --model scenic.simulators.carla.model \
    examples/driving/badlyParkedCarPullingIn.scenic
```

- LGSVL, specifying a map which it supports:

```
$ scenic -S --model scenic.simulators.lgsvl.model \
    --param map tests/formats/opendrive/maps/LGSVL/borregasave.xodr \
    --param lgsvl_map BorregasAve \
    examples/driving/badlyParkedCarPullingIn.scenic
```

[1] Assuming the simulator supports the selected map. If necessary, the map may be changed from the command line using the `--param` option; see the *model documentation* for details.

actions	Actions for dynamic agents in the driving domain.
behaviors	Library of useful behaviors for dynamic agents in driving scenarios.
controllers	Low-level controllers useful for vehicles.
model	Scenic world model for scenarios using the driving domain.
roads	Library for representing road network geometry and traffic information.
simulators	Abstract interface to simulators supporting the driving domain.
workspace	Workspaces for the driving domain.

scenic.domains.driving.actions

Actions for dynamic agents in the driving domain.

These actions are automatically imported when using the driving domain.

The *RegulatedControlAction* is based on code from the CARLA project, licensed under the following terms:

> Copyright (c) 2018-2020 CVC.
>
> This work is licensed under the terms of the MIT license. For a copy, see <https://opensource.org/licenses/MIT>.

Summary of Module Members

Classes

OffsetAction	Teleports actor forward (in direction of its heading) by some offset.
RegulatedControlAction	Regulated control of throttle, braking, and steering.
SetBrakeAction	Set the amount of brake.
SetHandBrakeAction	Set or release the hand brake.
SetPositionAction	Teleport an agent to the given position.
SetReverseAction	Engage or release reverse gear.
SetSpeedAction	Set the speed of an agent (keeping its heading fixed).
SetSteerAction	Set the steering 'angle'.
SetThrottleAction	Set the throttle.
SetVelocityAction	Set the velocity of an agent.
SetWalkingDirectionAction	Set the walking direction.
SetWalkingSpeedAction	Set the walking speed.
SteeringAction	Abstract class for actions usable by agents which can steer.
Steers	Mixin protocol for agents which can steer.
WalkingAction	Abstract class for actions usable by agents which can walk.
Walks	Mixin protocol for agents which can walk with a given direction and speed.

Member Details

class Steers

Mixin protocol for agents which can steer.

Specifically, agents must support throttling, braking, steering, setting the hand brake, and going into reverse.

class Walks

Mixin protocol for agents which can walk with a given direction and speed.

We provide a simplistic implementation which directly sets the velocity of the agent. This implementation needs to be explicitly opted-into, since simulators may provide a more sophisticated API that properly animates pedestrians.

class SetPositionAction(*pos*)

Bases: *Action*

Teleport an agent to the given position.

Parameters
pos (Vector) –

class OffsetAction(*offset*)

Bases: *Action*

Teleports actor forward (in direction of its heading) by some offset.

Parameters
offset (Vector) –

class SetVelocityAction(*xVel*, *yVel*, *zVel=0*)

Bases: *Action*

Set the velocity of an agent.

Parameters

- **xVel** (*float*) –
- **yVel** (*float*) –
- **zVel** (*float*) –

class SetSpeedAction(*speed*)

Bases: *Action*

Set the speed of an agent (keeping its heading fixed).

Parameters
speed (*float*) –

class SteeringAction

Bases: *Action*

Abstract class for actions usable by agents which can steer.

Such agents must implement the *Steers* protocol.

class SetThrottleAction(*throttle*)

Bases: *SteeringAction*

Set the throttle.

Parameters
throttle (*float*) – Throttle value between 0 and 1.

class SetSteerAction(*steer*)

Bases: *SteeringAction*

Set the steering 'angle'.

Parameters
steer (*float*) – Steering 'angle' between -1 and 1.

class SetBrakeAction(*brake*)

Bases: *SteeringAction*

Set the amount of brake.

Parameters
brake (*float*) – Amount of braking between 0 and 1.

class SetHandBrakeAction(*handBrake*)

Bases: *SteeringAction*

Set or release the hand brake.

Parameters
handBrake (*bool*) – Whether or not the hand brake is set.

class SetReverseAction(*reverse*)

Bases: *SteeringAction*

Engage or release reverse gear.

Parameters
reverse (*bool*) – Whether or not the car is in reverse.

class RegulatedControlAction(*throttle*, *steer*, *past_steer*, *max_throttle=0.5*, *max_brake=0.5*, *max_steer=0.8*)

Bases: *SteeringAction*

Regulated control of throttle, braking, and steering.

Controls throttle and braking using one signal that may be positive or negative. Useful with simple controllers that output a single value.

Parameters

- **throttle** (*float*) – Control signal for throttle and braking (will be clamped as below).
- **steer** (*float*) – Control signal for steering (also clamped).
- **past_steer** (*float*) – Previous steering signal, for regulating abrupt changes.
- **max_throttle** (*float*) – Maximum value for **throttle**, when positive.
- **max_brake** (*float*) – Maximum (absolute) value for **throttle**, when negative.
- **max_steer** (*float*) – Maximum absolute value for **steer**.

class WalkingAction

Bases: *Action*

Abstract class for actions usable by agents which can walk.

Such agents must implement the *Walks* protocol.

class SetWalkingDirectionAction(*heading*)

Bases: *WalkingAction*

Set the walking direction.

class SetWalkingSpeedAction(*speed*)

Bases: *WalkingAction*

Set the walking speed.

scenic.domains.driving.behaviors

Library of useful behaviors for dynamic agents in driving scenarios.

These behaviors are automatically imported when using the driving domain.

Summary of Module Members

Functions

concatenateCenterlines

Behaviors

AccelerateForwardBehavior
ConstantThrottleBehavior
DriveAvoidingCollisions

FollowLaneBehavior	Follow's the lane on which the vehicle is at, unless the laneToFollow is specified.
FollowTrajectoryBehavior	Follows the given trajectory.
LaneChangeBehavior	is_oppositeTraffic should be specified as True only if the laneSectionToSwitch to has the opposite traffic direction to the initial lane from which the vehicle started LaneChangeBehavior e.g.
TurnBehavior	This behavior uses a controller specifically tuned for turning at an intersection.
WalkForwardBehavior	Walk forward behavior for pedestrians.

Member Details

behavior FollowLaneBehavior(*target_speed=10*, *laneToFollow=None*, *is_oppositeTraffic=False*)

Follow's the lane on which the vehicle is at, unless the laneToFollow is specified. Once the vehicle reaches an intersection, by default, the vehicle will take the straight route. If straight route is not available, then any availble turn route will be taken, uniformly randomly. If turning at the intersection, the vehicle will slow down to make the turn, safely.

This behavior does not terminate. A recommended use of the behavior is to accompany it with condition, e.g. do FollowLaneBehavior() until ...

Parameters

- **target_speed** – Its unit is in m/s. By default, it is set to 10 m/s
- **laneToFollow** – If the lane to follow is different from the lane that the vehicle is on, this parameter can be used to specify that lane. By default, this variable will be set to None, which means that the vehicle will follow the lane that it is currently on.

behavior FollowTrajectoryBehavior(*target_speed=10*, *trajectory=None*, *turn_speed=None*)

Follows the given trajectory. The behavior terminates once the end of the trajectory is reached.

Parameters

- **target_speed** – Its unit is in m/s. By default, it is set to 10 m/s
- **trajectory** – It is a list of sequential lanes to track, from the lane that the vehicle is initially on to the lane it should end up on.

behavior LaneChangeBehavior(*laneSectionToSwitch*, *is_oppositeTraffic=False*, *target_speed=10*)

is_oppositeTraffic should be specified as True only if the laneSectionToSwitch to has the opposite traffic direction to the initial lane from which the vehicle started LaneChangeBehavior e.g. refer to the use of this flag in examples/carla/Carla_Challenge/carlaChallenge6.scenic

behavior TurnBehavior(*trajectory*, *target_speed=6*)

This behavior uses a controller specifically tuned for turning at an intersection. This behavior is only operational within an intersection, it will terminate if the vehicle is outside of an intersection.

behavior WalkForwardBehavior()

Walk forward behavior for pedestrians.

It will uniformly randomly choose either end of the sidewalk that the pedestrian is on, and have the pedestrian walk towards the endpoint.

scenic.domains.driving.controllers

Low-level controllers useful for vehicles.

The Lateral/Longitudinal PID controllers are adapted from CARLA's PID controllers, which are licensed under the following terms:

Summary of Module Members

Classes

PIDLateralController	Lateral control using a PID to track a trajectory.
PIDLongitudinalController	Longitudinal control using a PID to reach a target speed.

Member Details

class PIDLongitudinalController(*K_P=0.5*, *K_D=0.1*, *K_I=0.2*, *dt=0.1*)

Longitudinal control using a PID to reach a target speed.

Parameters

- **K_P** – Proportional gain
- **K_D** – Derivative gain
- **K_I** – Integral gain
- **dt** – time step

run_step(*speed_error*)

Estimate the throttle/brake of the vehicle based on the PID equations.

Parameters
speed_error – target speed minus current speed

Returns
a signal between -1 and 1, with negative values indicating braking.

class PIDLateralController(*K_P=0.3*, *K_D=0.2*, *K_I=0*, *dt=0.1*)

Lateral control using a PID to track a trajectory.

Parameters

- **K_P** – Proportional gain
- **K_D** – Derivative gain
- **K_I** – Integral gain
- **dt** – time step

run_step(*cte*)

Estimate the steering angle of the vehicle based on the PID equations.

Parameters
cte – cross-track error (distance to right of desired trajectory)

Returns
a signal between -1 and 1, with -1 meaning maximum steering to the left.

scenic.domains.driving.model

Scenic world model for scenarios using the driving domain.

Imports actions and behaviors for dynamic agents from *scenic.domains.driving.actions* and *scenic.domains.driving.behaviors*.

The map file to use for the scenario must be specified before importing this model by defining the global parameter `map`. This path is passed to the *Network.fromFile* function to create a *Network* object representing the road network. Extra options may be passed to the function by defining the global parameter `map_options`, which should be a dictionary of keyword arguments. For example, we could write:

```
param map = localPath('mymap.xodr')
param map_options = { 'tolerance': 0.1 }
model scenic.domains.driving.model
```

If you are writing a generic scenario that supports multiple maps, you may leave the `map` parameter undefined; then running the scenario will produce an error unless the user uses the *--param* command-line option to specify the map.

Note: If you are using a simulator, you may have to also define simulator-specific global parameters to tell the simulator which world to load. For example, our LGSVL interface uses a parameter `lgsvl_map` to specify the name of the Unity scene. See the *documentation* of the simulator interfaces for details.

Summary of Module Members

Module Attributes

network	The road network being used for the scenario, as a *Network* object.
road	The union of all drivable roads, including intersections but not shoulders or parking lanes.
curb	The union of all curbs.
sidewalk	The union of all sidewalks.
shoulder	The union of all shoulders, including parking lanes.
roadOrShoulder	All drivable areas, including both ordinary roads and shoulders.
intersection	The union of all intersections.
roadDirection	A `VectorField` representing the nominal traffic direction at a given point.

Functions

`is2DMode`

withinDistanceToAnyCars	returns boolean
withinDistanceToAnyObjs	checks whether there exists any obj (1) in front of the vehicle, (2) within thresholdDistance
withinDistanceToObjsInLane	checks whether there exists any obj (1) in front of the vehicle, (2) on the same lane, (3) within thresholdDistance

Classes

Car	A car.
DrivingObject	Abstract class for objects in a road network.
NPCCar	Car for which accurate physics is not required.
Pedestrian	A pedestrian.
Vehicle	Vehicles which drive, such as cars.

Member Details

network: *Network*

The road network being used for the scenario, as a *Network* object.

road: *Region*

The union of all drivable roads, including intersections but not shoulders or parking lanes.

curb: *Region*

The union of all curbs.

sidewalk: *Region*

The union of all sidewalks.

shoulder: *Region*

The union of all shoulders, including parking lanes.

roadOrShoulder: *Region*

All drivable areas, including both ordinary roads and shoulders.

intersection: *Region*

The union of all intersections.

roadDirection: *VectorField*

A `VectorField` representing the nominal traffic direction at a given point.

Inside intersections or anywhere else where there can be multiple nominal traffic directions, the choice is arbitrary. At such points, the function *Network.nominalDirectionsAt* can be used to get all nominal directions.

class DrivingObject <specifiers>

Bases: *Object2D*

Abstract class for objects in a road network.

Provides convenience properties for the lane, road, intersection, etc. at the object's current position (if any).

Also defines the `elevation` property as a standard way to access the Z component of an object's position, since the Scenic built-in property `position` is only 2D. If `elevation` is set to `None`, the simulator is responsible for choosing an appropriate Z coordinate so that the object is on the ground, then updating the property. 2D simulators should set the property to zero.

Properties

- **elevation** (*float or None; dynamic*) – default `None` (see above).
- **requireVisible** (*bool*) – Default value `False` (overriding the default from *Object*).

property lane: *Lane*

The *Lane* at the object's current position.

The simulation is rejected if the object is not in a lane. (Use *DrivingObject._lane* to get None instead.)

property _lane: Optional[*Lane*]

The *Lane* at the object's current position, if any.

property laneSection: *LaneSection*

The *LaneSection* at the object's current position.

The simulation is rejected if the object is not in a lane.

property _laneSection: Optional[*LaneSection*]

The *LaneSection* at the object's current position, if any.

property laneGroup: *LaneGroup*

The *LaneGroup* at the object's current position.

The simulation is rejected if the object is not in a lane.

property _laneGroup: Optional[*LaneGroup*]

The *LaneGroup* at the object's current position, if any.

property oppositeLaneGroup: *LaneGroup*

The *LaneGroup* on the other side of the road from the object.

The simulation is rejected if the object is not on a two-way road.

property road: *Road*

The *Road* at the object's current position.

The simulation is rejected if the object is not on a road.

property _road: Optional[*Road*]

The *Road* at the object's current position, if any.

property intersection: *Intersection*

The *Intersection* at the object's current position.

The simulation is rejected if the object is not in an intersection.

property _intersection: Optional[*Intersection*]

The *Intersection* at the object's current position, if any.

property crossing: *PedestrianCrossing*

The *PedestrianCrossing* at the object's current position.

The simulation is rejected if the object is not in a crosswalk.

property _crossing: Optional[*PedestrianCrossing*]

The *PedestrianCrossing* at the object's current position, if any.

property element: *NetworkElement*

The highest-level *NetworkElement* at the object's current position.

See *Network.elementAt* for the details of how this is determined. The simulation is rejected if the object is not in any network element.

property _element: Optional[*NetworkElement*]

The highest-level *NetworkElement* at the object's current position, if any.

`distanceToClosest`(*type*)

Compute the distance to the closest object of the given type.

For example, one could write *self*.distanceToClosest(Car) in a behavior.

Parameters
type (*type*) –

Return type
Object2D

class Vehicle <specifiers>

Bases: *DrivingObject*

Vehicles which drive, such as cars.

Properties

- **position** – The default position is uniformly random over the *road*.
- **heading** – The default heading is aligned with *roadDirection*, plus an offset given by **roadDeviation**.
- **roadDeviation** (*float*) – Relative heading with respect to the road direction at the *Vehicle*'s position. Used by the default value for **heading**.
- **regionContainedIn** – The default container is *roadOrShoulder*.
- **viewAngle** – The default view angle is 90 degrees.
- **width** – The default width is 2 meters.
- **length** – The default length is 4.5 meters.
- **color** (Color or RGB tuple) – Color of the vehicle. The default value is a distribution derived from car color popularity statistics; see Color.defaultCarColor.

class Car <specifiers>

Bases: *Vehicle*

A car.

class NPCCar <specifiers>

Bases: *Car*

Car for which accurate physics is not required.

class Pedestrian <specifiers>

Bases: *DrivingObject*

A pedestrian.

Properties

- **position** – The default position is uniformly random over sidewalks and crosswalks.
- **heading** – The default heading is uniformly random.
- **viewAngle** – The default view angle is 90 degrees.
- **width** – The default width is 0.75 m.
- **length** – The default length is 0.75 m.
- **color** – The default color is turquoise. Pedestrian colors are not necessarily used by simulators, but do appear in the debugging diagram.

withinDistanceToAnyCars(*car*, *thresholdDistance*)

returns boolean

withinDistanceToAnyObjs(*vehicle*, *thresholdDistance*)

checks whether there exists any obj (1) in front of the vehicle, (2) within thresholdDistance

withinDistanceToObjsInLane(*vehicle*, *thresholdDistance*)

checks whether there exists any obj (1) on the same lane, (3) within thresholdDistance

scenic.domains.driving.roads

Library for representing road network geometry and traffic information.

A road network is represented by an instance of the *Network* class, which can be created from a map file using *Network.fromFile*.

Note: This library is a prototype under active development. We will try not to make backwards-incompatible changes, but the API may not be entirely stable.

Summary of Module Members

Module Attributes

Vectorlike	Alias for types which can be interpreted as positions in Scenic.

Classes

Intersection	An intersection where multiple roads meet.
Lane	A lane for cars, bicycles, or other vehicles.
LaneGroup	A group of parallel lanes with the same type and direction.
LaneSection	Part of a lane in a single *RoadSection*.
LinearElement	A part of a road network with (mostly) linear 1- or 2-way flow.
Maneuver	A maneuver which can be taken upon reaching the end of a lane.
ManeuverType	A type of *Maneuver*, e.g., going straight or turning left.
Network	A road network.
NetworkElement	Abstract class for part of a road network.
PedestrianCrossing	A pedestrian crossing (crosswalk).
Road	A road consisting of one or more lanes.
RoadSection	Part of a road with a fixed number of lanes.
Shoulder	A shoulder of a road, including parking lanes by default.
Sidewalk	A sidewalk.
Signal	Traffic lights, stop signs, etc.
VehicleType	A type of vehicle, including pedestrians.

Member Details

Vectorlike

Alias for types which can be interpreted as positions in Scenic.

This includes instances of *Point* and *Object*, and pairs of numbers.

alias of Union[*Vector*, *Point2D*, Tuple[Real, Real]]

class VehicleType(*value*)

Bases: Enum

A type of vehicle, including pedestrians. Used to classify lanes.

class ManeuverType(*value*)

Bases: Enum

A type of *Maneuver*, e.g., going straight or turning left.

STRAIGHT = 1

Straight, including one lane merging into another.

LEFT_TURN = 2

Left turn.

RIGHT_TURN = 3

Right turn.

U_TURN = 4

U-turn.

static guessTypeFromLanes(*start*, *end*, *connecting*, *turnThreshold=0.3490658503988659*)

For formats lacking turn information, guess it from the geometry.

Parameters

- **start** (Lane) – starting lane of the maneuver.
- **end** (Lane) – ending lane of the maneuver.
- **connecting** (*Optional*[Lane]) – connecting lane of the maneuver, if any.
- **turnThreshold** (*float*) – angle beyond which to consider a maneuver a turn.

class Maneuver

A maneuver which can be taken upon reaching the end of a lane.

Parameters

- **type** (ManeuverType) –
- **startLane** (Lane) –
- **endLane** (Lane) –
- **connectingLane** (*Optional*[Lane]) –
- **intersection** (*Optional*[Intersection]) –

type: *ManeuverType*

type of maneuver (straight, left turn, etc.)

startLane: *Lane*

starting lane of the maneuver

endLane: *Lane*

ending lane of the maneuver

connectingLane: Optional[*Lane*]

connecting lane from the start to the end lane, if any (None for lane mergers)

intersection: Optional[*Intersection*]

intersection where the maneuver takes place, if any (None for lane mergers)

property conflictingManeuvers: Tuple[*Maneuver*]

Maneuvers whose connecting lanes intersect this one's.

property reverseManeuvers: Tuple[*Maneuver*]

Maneuvers whose start and end roads are the reverse of this one's.

class NetworkElement

Bases: *PolygonalRegion*

Abstract class for part of a road network.

Includes roads, lane groups, lanes, sidewalks, pedestrian crossings, and intersections.

This is a subclass of *Region*, so you can do things like `Car in lane` or `Car on road` if `lane` and `road` are elements, as well as computing distances to an element, etc.

Parameters

- **polygon** (*Union[Polygon, MultiPolygon]*) –
- **orientation** (*Optional[VectorField]*) –
- **name** (*str*) –
- **uid** (*str*) –
- **id** (*Optional[str]*) –
- **network** (Network) –
- **vehicleTypes** (*FrozenSet[VehicleType]*) –
- **speedLimit** (*Optional[float]*) –
- **tags** (*FrozenSet[str]*) –

name: str

Human-readable name, if any.

uid: str

Unique identifier; from underlying format, if possible. (In OpenDRIVE, for example, ids are not necessarily unique, so we invent our own.)

id: Optional[str]

Identifier from underlying format, if any.

network: *Network*

Link to parent network.

vehicleTypes: FrozenSet[*VehicleType*]

Which types of vehicles (car, bicycle, etc.) can be here.

speedLimit: Optional[float]

Optional speed limit, which may be inherited from parent.

tags: FrozenSet[str]

Uninterpreted semantic tags, e.g. 'roundabout'.

nominalDirectionsAt(*point*)

Get nominal traffic direction(s) at a point in this element.

There must be at least one such direction. If there are multiple, we pick one arbitrarily to be the orientation of the element as a *Region*. (So `Object in element` will align by default to that orientation.)

Parameters

point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type

Tuple[Orientation]

class LinearElement

Bases: *NetworkElement*

A part of a road network with (mostly) linear 1- or 2-way flow.

Includes roads, lane groups, lanes, sidewalks, and pedestrian crossings, but not intersections.

LinearElements have a direction, namely from the first point on their centerline to the last point. This is called 'forward', even for 2-way roads. The 'left' and 'right' edges are interpreted with respect to this direction.

The left/right edges are oriented along the direction of traffic near them; so for 2-way roads they will point opposite directions.

Parameters

- **polygon** (*Union[Polygon, MultiPolygon]*) –
- **orientation** (*Optional[VectorField]*) –
- **name** (*str*) –
- **uid** (*str*) –
- **id** (*Optional[str]*) –
- **network** (Network) –
- **vehicleTypes** (*FrozenSet[VehicleType]*) –
- **speedLimit** (*Optional[float]*) –
- **tags** (*FrozenSet[str]*) –
- **centerline** (PolylineRegion) –
- **leftEdge** (PolylineRegion) –
- **rightEdge** (PolylineRegion) –
- **successor** (*Union[NetworkElement, None]*) –
- **predecessor** (*Union[NetworkElement, None]*) –

flowFrom(*point*, *distance*, *steps=None*, *stepSize=5*)

Advance a point along this element by a given distance.

Equivalent to `follow element.orientation from point for distance`, but possibly more accurate. The default implementation uses the forward Euler approximation with a step size of 5 meters; subclasses may ignore the **steps** and **stepSize** parameters if they can compute the flow exactly.

Parameters

- **point** (*`scenic.domains.driving.roads.Vectorlike`*) – point to start from.
- **distance** (*float*) – distance to travel.
- **steps** (*Optional[int]*) – number of steps to take, or None to compute the number of steps based on the distance (default None).
- **stepSize** (*float*) – length used to compute how many steps to take, if **steps** is not specified (default 5 meters).

Return type
Vector

class Road

Bases: *LinearElement*

A road consisting of one or more lanes.

Lanes are grouped into 1 or 2 instances of *LaneGroup*:

- **forwardLanes**: the lanes going the same direction as the road
- **backwardLanes**: the lanes going the opposite direction

One of these may be None if there are no lanes in that direction.

Because of splits and mergers, the Lanes of a *Road* do not necessarily start or end at the same point as the *Road*. Such intermediate branching points cause the *Road* to be partitioned into multiple road sections, within which the configuration of lanes is fixed.

Parameters

- **polygon** (*Union[Polygon, MultiPolygon]*) –
- **orientation** (*Optional[VectorField]*) –
- **name** (*str*) –
- **uid** (*str*) –
- **id** (*Optional[str]*) –
- **network** (Network) –
- **vehicleTypes** (*FrozenSet[VehicleType]*) –
- **speedLimit** (*Optional[float]*) –
- **tags** (*FrozenSet[str]*) –
- **centerline** (PolylineRegion) –
- **leftEdge** (PolylineRegion) –
- **rightEdge** (PolylineRegion) –
- **successor** (*Union[NetworkElement, None]*) –
- **predecessor** (*Union[NetworkElement, None]*) –
- **lanes** (*Tuple[Lane]*) –
- **forwardLanes** (*Optional[LaneGroup]*) –
- **backwardLanes** (*Optional[LaneGroup]*) –
- **laneGroups** (*Tuple[LaneGroup]*) –

- **sections** (*Tuple[RoadSection]*) –
- **signals** (*Tuple[Signal]*) –
- **crossings** (*Tuple[PedestrianCrossing]*) –
- **sidewalks** (*Tuple[Sidewalk]*) –
- **sidewalkRegion** (PolygonalRegion) –

lanes: Tuple[*Lane*]

All lanes of this road, in either direction.

The order of the lanes is arbitrary. To access lanes in order according to their geometry, use *LaneGroup.lanes*.

forwardLanes: Optional[*LaneGroup*]

Group of lanes aligned with the direction of the road, if any.

backwardLanes: Optional[*LaneGroup*]

Group of lanes going in the opposite direction, if any.

laneGroups: Tuple[*LaneGroup*]

All LaneGroups of this road, with *forwardLanes* being first if it exists.

sections: Tuple[*RoadSection*]

All sections of this road, ordered from start to end.

crossings: Tuple[*PedestrianCrossing*]

All crosswalks of this road, ordered from start to end.

sidewalks: Tuple[*Sidewalk*]

All sidewalks of this road, with the one adjacent to *forwardLanes* being first.

sidewalkRegion: *PolygonalRegion*

Possibly-empty region consisting of all sidewalks of this road.

sectionAt(*point*, *reject=False*)

Get the *RoadSection* passing through a given point.

Parameters
point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type
Optional[RoadSection]

laneSectionAt(*point*, *reject=False*)

Get the *LaneSection* passing through a given point.

Parameters
point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type
Optional[LaneSection]

laneAt(*point*, *reject=False*)

Get the *Lane* passing through a given point.

Parameters
point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type
Optional[Lane]

laneGroupAt(*point*, *reject=False*)

Get the *LaneGroup* passing through a given point.

Parameters
point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type
Optional[LaneGroup]

crossingAt(*point*, *reject=False*)

Get the *PedestrianCrossing* passing through a given point.

Parameters
point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type
Optional[PedestrianCrossing]

shiftLanes(*point*, *offset*)

Find the point equivalent to this one but shifted over some # of lanes.

Parameters

- **point** (*`scenic.domains.driving.roads.Vectorlike`*) –
- **offset** (*int*) –

Return type
Optional[Vector]

class LaneGroup

Bases: *LinearElement*

A group of parallel lanes with the same type and direction.

Parameters

- **polygon** (*Union[Polygon, MultiPolygon]*) –
- **orientation** (*Optional[VectorField]*) –
- **name** (*str*) –
- **uid** (*str*) –
- **id** (*Optional[str]*) –
- **network** (Network) –
- **vehicleTypes** (*FrozenSet[VehicleType]*) –
- **speedLimit** (*Optional[float]*) –
- **tags** (*FrozenSet[str]*) –
- **centerline** (PolylineRegion) –
- **leftEdge** (PolylineRegion) –
- **rightEdge** (PolylineRegion) –
- **successor** (*Union[NetworkElement, None]*) –
- **predecessor** (*Union[NetworkElement, None]*) –
- **road** (Road) –

- **lanes** (*Tuple[Lane]*) –
- **curb** (PolylineRegion) –
- **sidewalk** (*Union[Sidewalk, None]*) –
- **bikeLane** (*Union[Lane, None]*) –
- **shoulder** (*Union[Shoulder, None]*) –
- **opposite** (*Union[LaneGroup, None]*) –

road: *Road*

Parent road.

lanes: Tuple[*Lane*]

Lanes, partially ordered with lane 0 being closest to the curb.

curb: *PolylineRegion*

Region representing the associated curb, which is not necessarily adjacent if there are parking lanes or some other kind of shoulder.

_sidewalk: Optional[*Sidewalk*]

Adjacent sidewalk, if any.

_shoulder: Optional[*Shoulder*]

Adjacent shoulder, if any.

_opposite: Optional[*LaneGroup*]

Opposite lane group of the same road, if any.

property sidewalk: *Sidewalk*

The adjacent sidewalk; rejects if there is none.

property shoulder: *Shoulder*

The adjacent shoulder; rejects if there is none.

property opposite: *LaneGroup*

The opposite lane group of the same road; rejects if there is none.

laneAt(*point*, *reject=False*)

Get the *Lane* passing through a given point.

Parameters
point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type
Optional[Lane]

class Lane

Bases: *LinearElement*

A lane for cars, bicycles, or other vehicles.

Parameters

- **polygon** (*Union[Polygon, MultiPolygon]*) –
- **orientation** (*Optional[VectorField]*) –
- **name** (*str*) –
- **uid** (*str*) –

- **id** (*Optional[str]*) –
- **network** (Network) –
- **vehicleTypes** (*FrozenSet[VehicleType]*) –
- **speedLimit** (*Optional[float]*) –
- **tags** (*FrozenSet[str]*) –
- **centerline** (PolylineRegion) –
- **leftEdge** (PolylineRegion) –
- **rightEdge** (PolylineRegion) –
- **successor** (*Union[NetworkElement, None]*) –
- **predecessor** (*Union[NetworkElement, None]*) –
- **group** (LaneGroup) –
- **road** (Road) –
- **sections** (*Tuple[LaneSection]*) –
- **adjacentLanes** (*Tuple[Lane]*) –
- **maneuvers** (*Tuple[Maneuver]*) –

sectionAt(*point*, *reject=False*)

Get the LaneSection passing through a given point.

Parameters

point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type

Optional[LaneSection]

class RoadSection

Bases: *LinearElement*

Part of a road with a fixed number of lanes.

A RoadSection has a fixed number of lanes: when a lane begins or ends, we move to a new section (which will be the successor of the current one).

Parameters

- **polygon** (*Union[Polygon, MultiPolygon]*) –
- **orientation** (*Optional[VectorField]*) –
- **name** (*str*) –
- **uid** (*str*) –
- **id** (*Optional[str]*) –
- **network** (Network) –
- **vehicleTypes** (*FrozenSet[VehicleType]*) –
- **speedLimit** (*Union[float, None]*) –
- **tags** (*FrozenSet[str]*) –
- **centerline** (PolylineRegion) –

- **leftEdge** (PolylineRegion) –
- **rightEdge** (PolylineRegion) –
- **successor** (*Union[NetworkElement, None]*) –
- **predecessor** (*Union[NetworkElement, None]*) –
- **road** (Road) –
- **lanes** (*Tuple[LaneSection]*) –
- **forwardLanes** (*Tuple[LaneSection]*) –
- **backwardLanes** (*Tuple[LaneSection]*) –
- **lanesByOpenDriveID** (*Dict[LaneSection]*) –

laneAt(*point*, *reject=False*)

Get the lane section passing through a given point.

Parameters

point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type

Optional[LaneSection]

class LaneSection

Bases: *LinearElement*

Part of a lane in a single *RoadSection*.

Since the lane configuration in a *RoadSection* is fixed, a *LaneSection* can have at most one adjacent lane to left or right. These are accessible using the *laneToLeft* and *laneToRight* properties, which for convenience reject the simulation if the desired lane does not exist. If rejection is not desired (for example if you want to handle the case where there is no lane to the left yourself), you can use the *_laneToLeft* and *_laneToRight* properties instead.

Parameters

- **polygon** (*Union[Polygon, MultiPolygon]*) –
- **orientation** (*Optional[VectorField]*) –
- **name** (*str*) –
- **uid** (*str*) –
- **id** (*Optional[str]*) –
- **network** (Network) –
- **vehicleTypes** (*FrozenSet[VehicleType]*) –
- **speedLimit** (*Optional[float]*) –
- **tags** (*FrozenSet[str]*) –
- **centerline** (PolylineRegion) –
- **leftEdge** (PolylineRegion) –
- **rightEdge** (PolylineRegion) –
- **successor** (*Union[NetworkElement, None]*) –
- **predecessor** (*Union[NetworkElement, None]*) –

- **lane** (Lane) –
- **group** (LaneGroup) –
- **road** (Road) –
- **openDriveID** (*int*) –
- **isForward** (*bool*) –
- **adjacentLanes** (*Tuple[LaneSection]*) –
- **laneToLeft** (*Union[LaneSection, None]*) –
- **laneToRight** (*Union[LaneSection, None]*) –
- **fasterLane** (*Union[LaneSection, None]*) –
- **slowerLane** (*Union[LaneSection, None]*) –

lane: *Lane*

Parent lane.

group: *LaneGroup*

Grandparent lane group.

road: *Road*

Great-grandparent road.

isForward: bool

Whether this lane has the same direction as its parent road.

adjacentLanes: Tuple[*LaneSection*]

Adjacent lanes of the same type, if any.

_laneToLeft: Optional[*LaneSection*]

Adjacent lane of same type to the left, if any.

_laneToRight: Optional[*LaneSection*]

Adjacent lane of same type to the right, if any.

_fasterLane: Optional[*LaneSection*]

Faster adjacent lane of same type, if any. Could be to left or right depending on the country.

_slowerLane: Optional[*LaneSection*]

Slower adjacent lane of same type, if any.

property laneToLeft: *LaneSection*

The adjacent lane of the same type to the left; rejects if there is none.

property laneToRight: *LaneSection*

The adjacent lane of the same type to the right; rejects if there is none.

property fasterLane: *LaneSection*

The faster adjacent lane of the same type; rejects if there is none.

property slowerLane: *LaneSection*

The slower adjacent lane of the same type; rejects if there is none.

shiftedBy(*offset*)

Find the lane a given number of lanes over from this lane.

Parameters
offset (*int*) –

Return type
Optional[LaneSection]

class Sidewalk

Bases: *LinearElement*

A sidewalk.

Parameters

- **polygon** (*Union[Polygon, MultiPolygon]*) –
- **orientation** (*Optional[VectorField]*) –
- **name** (*str*) –
- **uid** (*str*) –
- **id** (*Optional[str]*) –
- **network** (Network) –
- **vehicleTypes** (*FrozenSet[VehicleType]*) –
- **speedLimit** (*Optional[float]*) –
- **tags** (*FrozenSet[str]*) –
- **centerline** (PolylineRegion) –
- **leftEdge** (PolylineRegion) –
- **rightEdge** (PolylineRegion) –
- **successor** (*Union[NetworkElement, None]*) –
- **predecessor** (*Union[NetworkElement, None]*) –
- **road** (Road) –
- **crossings** (*Tuple[PedestrianCrossing]*) –

class PedestrianCrossing

Bases: *LinearElement*

A pedestrian crossing (crosswalk).

Parameters

- **polygon** (*Union[Polygon, MultiPolygon]*) –
- **orientation** (*Optional[VectorField]*) –
- **name** (*str*) –
- **uid** (*str*) –
- **id** (*Optional[str]*) –
- **network** (Network) –
- **vehicleTypes** (*FrozenSet[VehicleType]*) –

- **speedLimit** (*Optional[float]*) –
- **tags** (*FrozenSet[str]*) –
- **centerline** (PolylineRegion) –
- **leftEdge** (PolylineRegion) –
- **rightEdge** (PolylineRegion) –
- **successor** (*Union[NetworkElement, None]*) –
- **predecessor** (*Union[NetworkElement, None]*) –
- **parent** (*Union[Road, Intersection]*) –
- **startSidewalk** (Sidewalk) –
- **endSidewalk** (Sidewalk) –

class Shoulder

Bases: *LinearElement*

A shoulder of a road, including parking lanes by default.

Parameters

- **polygon** (*Union[Polygon, MultiPolygon]*) –
- **orientation** (*Optional[VectorField]*) –
- **name** (*str*) –
- **uid** (*str*) –
- **id** (*Optional[str]*) –
- **network** (Network) –
- **vehicleTypes** (*FrozenSet[VehicleType]*) –
- **speedLimit** (*Optional[float]*) –
- **tags** (*FrozenSet[str]*) –
- **centerline** (PolylineRegion) –
- **leftEdge** (PolylineRegion) –
- **rightEdge** (PolylineRegion) –
- **successor** (*Union[NetworkElement, None]*) –
- **predecessor** (*Union[NetworkElement, None]*) –
- **road** (Road) –

class Intersection

Bases: *NetworkElement*

An intersection where multiple roads meet.

Parameters

- **polygon** (*Union[Polygon, MultiPolygon]*) –
- **orientation** (*Optional[VectorField]*) –
- **name** (*str*) –

- **uid** (*str*) –
- **id** (*Optional[str]*) –
- **network** (Network) –
- **vehicleTypes** (*FrozenSet[VehicleType]*) –
- **speedLimit** (*Optional[float]*) –
- **tags** (*FrozenSet[str]*) –
- **roads** (*Tuple[Road]*) –
- **incomingLanes** (*Tuple[Lane]*) –
- **outgoingLanes** (*Tuple[Lane]*) –
- **maneuvers** (*Tuple[Maneuver]*) –
- **signals** (*Tuple[Signal]*) –
- **crossings** (*Tuple[PedestrianCrossing]*) –

property is3Way: bool

Whether or not this is a 3-way intersection.

Type
bool

property is4Way: bool

Whether or not this is a 4-way intersection.

Type
bool

property isSignalized: bool

Whether or not this is a signalized intersection.

Type
bool

maneuversAt(*point*)

Get all maneuvers possible at a given point in the intersection.

Parameters
point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type
List[Maneuver]

class Signal(*, *uid=None*, *openDriveID*, *country*, *type*)

Traffic lights, stop signs, etc.

> **Warning:** Signal parsing is a work in progress and the API is likely to change in the future.

Parameters

- **uid** (*str*) –
- **openDriveID** (*int*) –
- **country** (*str*) –

- **type** (*str*) –

openDriveID: int

ID number as in OpenDRIVE (unique ID of the signal within the database)

country: str

Country code of the signal

type: str

Type identifier according to country code.

property isTrafficLight: bool

Whether or not this signal is a traffic light.

class Network

A road network.

Networks are composed of roads, intersections, sidewalks, etc., which are all instances of *NetworkElement*.

Road networks can be loaded from standard formats using *Network.fromFile*.

Parameters

- **elements** (*Dict[str, NetworkElement]*) –
- **roads** (*Tuple[Road]*) –
- **connectingRoads** (*Tuple[Road]*) –
- **allRoads** (*Tuple[Road]*) –
- **laneGroups** (*Tuple[LaneGroup]*) –
- **lanes** (*Tuple[Lane]*) –
- **intersections** (*Tuple[Intersection]*) –
- **crossings** (*Tuple[PedestrianCrossing]*) –
- **sidewalks** (*Tuple[Sidewalk]*) –
- **shoulders** (*Tuple[Shoulder]*) –
- **roadSections** (*Tuple[RoadSection]*) –
- **laneSections** (*Tuple[LaneSection]*) –
- **driveOnLeft** (*bool*) –
- **tolerance** (*float*) –
- **drivableRegion** (PolygonalRegion) –
- **walkableRegion** (PolygonalRegion) –
- **roadRegion** (PolygonalRegion) –
- **laneRegion** (PolygonalRegion) –
- **intersectionRegion** (PolygonalRegion) –
- **crossingRegion** (PolygonalRegion) –
- **sidewalkRegion** (PolygonalRegion) –
- **curbRegion** (PolylineRegion) –
- **shoulderRegion** (PolygonalRegion) –

- **roadDirection** (VectorField) –

elements: Dict[str, *NetworkElement*]

All network elements, indexed by unique ID.

roads: Tuple[*Road*]

All ordinary roads in the network (i.e. those not part of an intersection).

connectingRoads: Tuple[*Road*]

All roads connecting one exit of an intersection to another.

allRoads: Tuple[*Road*]

All roads of either type.

laneGroups: Tuple[*LaneGroup*]

All lane groups in the network.

lanes: Tuple[*Lane*]

All lanes in the network.

intersections: Tuple[*Intersection*]

All intersections in the network.

crossings: Tuple[*PedestrianCrossing*]

All pedestrian crossings in the network.

sidewalks: Tuple[*Sidewalk*]

All sidewalks in the network.

shoulders: Tuple[*Shoulder*]

All shoulders in the network (by default, includes parking lanes).

roadSections: Tuple[*RoadSection*]

All sections of ordinary roads in the network.

laneSections: Tuple[*LaneSection*]

All sections of lanes in the network.

driveOnLeft: bool

Whether or not cars drive on the left in this network.

tolerance: float

Distance tolerance for testing inclusion in network elements.

roadDirection: *VectorField*

Traffic flow vector field aggregated over all roads (0 elsewhere).

pickledExt = '.snet'

File extension for cached versions of processed networks.

exception DigestMismatchError

Bases: Exception

Exception raised when loading a cached map not matching the original file.

classmethod fromFile(*path*, *useCache=True*, *writeCache=True*, ***kwargs*)

Create a *Network* from a map file.

This function calls an appropriate parsing routine based on the extension of the given file. Supported map formats are:

- OpenDRIVE (`.xodr`): *Network.fromOpenDrive*

See the functions listed above for format-specific options to this function. If no file extension is given in **path**, this function searches for any file with the given name in one of the formats above (in order).

Parameters

- **path** – A string or other path-like object giving a path to a file. If no file extension is included, we search for any file type we know how to parse.
- **useCache** (*bool*) – Whether to use a cached version of the map, if one exists and matches the given map file (default true; note that if the map file changes, the cached version will still not be used).
- **writeCache** (*bool*) – Whether to save a cached version of the processed map after parsing has finished (default true).
- **kwargs** – Additional keyword arguments specific to particular map formats.

Raises

- **FileNotFoundError** – no readable map was found at the given path.
- **ValueError** – the given map is of an unknown format.

classmethod fromOpenDrive(*path*, *ref_points=20*, *tolerance=0.05*, *fill_gaps=True*, *fill_intersections=True*, *elide_short_roads=False*)

Create a *Network* from an OpenDRIVE file.

Parameters

- **path** – Path to the file, as in *Network.fromFile*.
- **ref_points** (*int*) – Number of points to discretize continuous reference lines into.
- **tolerance** (*float*) – Tolerance for merging nearby geometries.
- **fill_gaps** (*bool*) – Whether to attempt to fill gaps between adjacent lanes.
- **fill_intersections** (*bool*) – Whether to attempt to fill gaps inside intersections.
- **elide_short_roads** (*bool*) – Whether to attempt to fix geometry artifacts by eliding roads with length less than **tolerance**.

findPointIn(*point*, *elems*, *reject*)

Find the first of the given elements containing the point.

Elements which *actually* contain the point have priority; if none contain the point, then we search again allowing an error of up to **tolerance**. If there are still no matches, we return None, unless **reject** is true, in which case we reject the current sample.

Parameters

- **point** (*`scenic.domains.driving.roads.Vectorlike`*) –
- **elems** (*Sequence[NetworkElement]*) –
- **reject** (*Union[bool, str]*) –

Return type
Optional[NetworkElement]

elementAt(*point*, *reject=False*)

Get the highest-level *NetworkElement* at a given point, if any.

If the point lies in an *Intersection*, we return that; otherwise if the point lies in a *Road*, we return that; otherwise we return None, or reject the simulation if **reject** is true (default false).

Parameters
point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type
Optional[NetworkElement]

roadAt(*point*, *reject=False*)

Get the *Road* passing through a given point.

Parameters
point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type
Optional[Road]

laneAt(*point*, *reject=False*)

Get the *Lane* passing through a given point.

Parameters
point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type
Optional[Lane]

laneSectionAt(*point*, *reject=False*)

Get the *LaneSection* passing through a given point.

Parameters
point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type
Optional[LaneSection]

laneGroupAt(*point*, *reject=False*)

Get the *LaneGroup* passing through a given point.

Parameters
point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type
Optional[LaneGroup]

crossingAt(*point*, *reject=False*)

Get the *PedestrianCrossing* passing through a given point.

Parameters
point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type
Optional[PedestrianCrossing]

intersectionAt(*point*, *reject=False*)

Get the *Intersection* at a given point.

Parameters
point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type
Optional[Intersection]

nominalDirectionsAt(*point*, *reject=False*)

Get the nominal traffic direction(s) at a given point, if any.

There can be more than one such direction in an intersection, for example: a car at a given point could be going straight, turning left, etc.

Parameters
point (*`scenic.domains.driving.roads.Vectorlike`*) –

Return type
Tuple[Orientation]

show(*labelIncomingLanes=False*)

Render a schematic of the road network for debugging.

If you call this function directly, you'll need to subsequently call `matplotlib.pyplot.show` to actually display the diagram.

Parameters
labelIncomingLanes (*bool*) – Whether to label the incoming lanes of intersections with their indices in `incomingLanes`.

scenic.domains.driving.simulators

Abstract interface to simulators supporting the driving domain.

Summary of Module Members

Classes

DrivingSimulation	A *Simulation* with a simulator supporting the driving domain.
DrivingSimulator	A *Simulator* supporting the driving domain.

Member Details

class DrivingSimulator

Bases: *Simulator*

A *Simulator* supporting the driving domain.

class DrivingSimulation(*scene*, *, *maxSteps*, *name*, *timestep*, *replay=None*, *enableReplay=True*, *allowPickle=False*, *enableDivergenceCheck=False*, *divergenceTolerance=0*, *continueAfterDivergence=False*, *verbosity=0*)

Bases: *Simulation*

A *Simulation* with a simulator supporting the driving domain.

This subclass of *Simulation* provides no special behavior by itself; it just provides convenience methods for creating controllers to be used by *FollowLaneBehavior* and related behaviors, so that the parameters of these controllers can be customized for different simulators.

getLaneFollowingControllers(*agent*)

Get longitudinal and lateral controllers for lane following.

The default controllers are simple PID controllers with parameters that work reasonably well for cars in simulators with realistic physics. See the classes *PIDLongitudinalController* and *PIDLateralController* for details, and *NewtonianSimulation* for an example of how to override this function.

Returns

A pair of controllers for throttle and steering respectively.

getTurningControllers(*agent*)

Get longitudinal and lateral controllers for turning.

getLaneChangingControllers(*agent*)

Get longitudinal and lateral controllers for lane changing.

scenic.domains.driving.workspace

Workspaces for the driving domain.

Summary of Module Members

Classes

DrivingWorkspace	Workspace created from a road *Network*.

Member Details

class DrivingWorkspace(*network*)

Bases: *Workspace*

Workspace created from a road *Network*.

scenic.formats

Support for file formats not specific to particular simulators.

opendrive	Support for loading OpenDRIVE maps.

scenic.formats.opendrive

Support for loading OpenDRIVE maps.

workspace	Workspaces based on OpenDRIVE maps.
xodr_parser	Parser for OpenDRIVE (.xodr) files.

scenic.formats.opendrive.workspace

Workspaces based on OpenDRIVE maps.

Summary of Module Members

Classes

OpenDriveWorkspace

Member Details

scenic.formats.opendrive.xodr_parser

Parser for OpenDRIVE (.xodr) files.

Summary of Module Members

Functions

buffer_union
warn

Classes

Clothoid	An Euler spiral with curvature varying linearly between CURV0 and CURV1.
Cubic	A curve defined by the cubic polynomial a + bu + cu^2 + du^3.
Curve	Geometric elements which compose road reference lines.
Junction	
Lane	
LaneSection	
Line	A line segment between (x0, y0) and (x1, y1).
ParamCubic	A curve defined by the parametric equations u = a_u + b_up + c_up^2 + d_up^3, v = a_v + b_vp + c_vp^2 + d_up^3, with p in [0, p_range].
Poly3	Cubic polynomial.
Road	
RoadLink	Indicates Roads a and b, with ids id_a and id_b respectively, are connected.
RoadMap	
Signal	Traffic lights, stop signs, etc.
SignalReference	

Exceptions

OpenDriveWarning

Member Details

class Poly3(*a*, *b*, *c*, *d*)

Cubic polynomial.

class Curve(*x0*, *y0*, *hdg*, *length*)

Geometric elements which compose road reference lines. See the OpenDRIVE Format Specification for coordinate system details.

to_points(*num*, *extra_points=[]*)

Sample NUM evenly-spaced points from curve.

Points are tuples of (x, y, s) with (x, y) absolute coordinates and s the arc length along the curve. Additional points at s values in extra_points are included if they are contained in the curve (unless they are extremely close to one of the equally-spaced points).

abstract point_at(*s*)

Get an (x, y, s) point along the curve at the given s coordinate.

rel_to_abs(*point*)

Convert from relative coordinates of curve to absolute coordinates. I.e. rotate counterclockwise by self.hdg and translate by (x0, x1).

class Cubic(*x0*, *y0*, *hdg*, *length*, *a*, *b*, *c*, *d*)

Bases: *Curve*

A curve defined by the cubic polynomial a + bu + cu^2 + du^3. The curve starts at (X0, Y0) in direction HDG, with length LENGTH.

class ParamCubic(*x0*, *y0*, *hdg*, *length*, *au*, *bu*, *cu*, *du*, *av*, *bv*, *cv*, *dv*, *p_range=1*)

Bases: *Curve*

A curve defined by the parametric equations u = a_u + b_up + c_up^2 + d_up^3, v = a_v + b_vp + c_vp^2 + d_up^3, with p in [0, p_range]. The curve starts at (X0, Y0) in direction HDG, with length LENGTH.

class Clothoid(*x0*, *y0*, *hdg*, *length*, *curv0*, *curv1*)

Bases: *Curve*

An Euler spiral with curvature varying linearly between CURV0 and CURV1. The spiral starts at (X0, Y0) in direction HDG, with length LENGTH.

class Line(*x0*, *y0*, *hdg*, *length*)

Bases: *Curve*

A line segment between (x0, y0) and (x1, y1).

class RoadLink(*id_a*, *id_b*, *contact_a*, *contact_b*)

Indicates Roads a and b, with ids id_a and id_b respectively, are connected.

class Signal(*id_*, *country*, *type_*, *subtype*, *orientation*, *validity=None*)

Traffic lights, stop signs, etc.

scenic.simulators

World models and interfaces for particular simulators.

carla	Interface to the CARLA driving simulator.
gta	Scenic world model for Grand Theft Auto V (GTAV).
lgsvl	Interface to the LGSVL driving simulator.
newtonian	Simple Newtonian physics simulator.
utils	Various utilities useful across multiple simulators.
webots	Scenic world models for the Webots robotics simulator.
xplane	Scenic world model for the X-Plane flight simulator.

scenic.simulators.carla

Interface to the CARLA driving simulator.

This interface has been tested with CARLA versions 0.9.9, 0.9.10, and 0.9.11. It supports dynamic scenarios involving vehicles, pedestrians, and props.

The interface implements the *scenic.domains.driving* abstract domain, so any object types, behaviors, utility functions, etc. from that domain may be used freely. For details of additional CARLA-specific functionality, see the world model *scenic.simulators.carla.model*.

actions	Actions for dynamic agents in CARLA scenarios.
behaviors	Behaviors for dynamic agents in CARLA scenarios.
blueprints	CARLA blueprints for cars, pedestrians, etc.
misc	Module with auxiliary functions.
model	Scenic world model for traffic scenarios in CARLA.
simulator	Simulator interface for CARLA.

scenic.simulators.carla.actions

Actions for dynamic agents in CARLA scenarios.

Summary of Module Members

Classes

PedestrianAction	
SetAngularVelocityAction	
SetAutopilotAction	
SetGearAction	
SetJumpAction	
SetManualFirstGearShiftAction	
SetManualGearShiftAction	
SetTrafficLightAction	Set the traffic light to desired color.
SetTransformAction	
SetVehicleLightStateAction	Set the vehicle lights' states.
SetWalkAction	
TrackWaypointsAction	
VehicleAction	

Member Details

class SetTrafficLightAction(*color*, *distance=100*, *group=False*)

Bases: VehicleAction

Set the traffic light to desired color. It will only take effect if the car is within a given distance of the traffic light.

Parameters

- **color** – the string red/yellow/green/off/unknown
- **distance** – the maximum distance to search for traffic lights from the current position

class SetVehicleLightStateAction(*vehicleLightState*)

Bases: VehicleAction

Set the vehicle lights' states.

Parameters
vehicleLightState – Which lights are on.

scenic.simulators.carla.behaviors

Behaviors for dynamic agents in CARLA scenarios.

behavior AutopilotBehavior()

Behavior causing a vehicle to use CARLA's built-in autopilot.

behavior CrossingBehavior(*reference_actor*, *min_speed=1*, *threshold=10*, *final_speed=None*)

This behavior dynamically controls the speed of an actor that will perpendicularly (or close to) cross the road, so that it arrives at a spot in the road at the same time as a reference actor.

Parameters

- **min_speed** (*float*) – minimum speed of the crossing actor. As this is a type of "synchronization action", a minimum speed is needed, to allow the actor to keep moving even if the reference actor has stopped
- **threshold** (*float*) – starting distance at which the crossing actor starts moving
- **final_speed** (*float*) – speed of the crossing actor after the reference one surpasses it

scenic.simulators.carla.blueprints

CARLA blueprints for cars, pedestrians, etc.

Summary of Module Members

Module Attributes

oldBlueprintNames	Mapping from current names of blueprints to ones in old CARLA versions.
carModels	blueprints for cars
bicycleModels	blueprints for bicycles
motorcycleModels	blueprints for motorcycles
truckModels	blueprints for trucks
trashModels	blueprints for trash cans
coneModels	blueprints for traffic cones
debrisModels	blueprints for road debris
vendingMachineModels	blueprints for vending machines
chairModels	blueprints for chairs
busStopModels	blueprints for bus stops
advertisementModels	blueprints for roadside billboards
garbageModels	blueprints for pieces of trash
containerModels	blueprints for containers
tableModels	blueprints for tables
barrierModels	blueprints for traffic barriers
plantpotModels	blueprints for flowerpots
mailboxModels	blueprints for mailboxes
gnomeModels	blueprints for garden gnomes
creasedboxModels	blueprints for creased boxes
caseModels	blueprints for briefcases, suitcases, etc.
boxModels	blueprints for boxes
benchModels	blueprints for benches
barrelModels	blueprints for barrels
atmModels	blueprints for ATMs
kioskModels	blueprints for kiosks
ironplateModels	blueprints for iron plates
trafficwarningModels	blueprints for traffic warning signs
walkerModels	blueprints for pedestrians

Member Details

oldBlueprintNames = {'vehicle.dodge.charger_police': ('vehicle.dodge_charger.police',), 'vehicle.ford.mustang': ('vehicle.mustang.mustang',), 'vehicle.lincoln.mkz_2017': ('vehicle.lincoln.mkz2017',), 'vehicle.mercedes.coupe': ('vehicle.mercedes-benz.coupe',), 'vehicle.mini.cooper_s': ('vehicle.mini.cooperst',)}

Mapping from current names of blueprints to ones in old CARLA versions.

We provide a tuple of old names in case they change more than once.

carModels = ['vehicle.audi.a2', 'vehicle.audi.etron', 'vehicle.audi.tt', 'vehicle.bmw.grandtourer', 'vehicle.chevrolet.impala', 'vehicle.citroen.c3', 'vehicle.dodge.charger_police', 'vehicle.jeep.wrangler_rubicon', 'vehicle.lincoln.mkz_2017', 'vehicle.mercedes.coupe', 'vehicle.mini.cooper_s', 'vehicle.ford.mustang', 'vehicle.nissan.micra', 'vehicle.nissan.patrol', 'vehicle.seat.leon', 'vehicle.tesla.model3', 'vehicle.toyota.prius', 'vehicle.volkswagen.t2']

blueprints for cars

`bicycleModels = ['vehicle.bh.crossbike', 'vehicle.diamondback.century', 'vehicle.gazelle.omafiets']`

blueprints for bicycles

`motorcycleModels = ['vehicle.harley-davidson.low_rider', 'vehicle.kawasaki.ninja', 'vehicle.yamaha.yzf']`

blueprints for motorcycles

`truckModels = ['vehicle.carlamotors.carlacola', 'vehicle.tesla.cybertruck']`

blueprints for trucks

`trashModels = ['static.prop.trashcan01', 'static.prop.trashcan02', 'static.prop.trashcan03', 'static.prop.trashcan04', 'static.prop.trashcan05', 'static.prop.bin']`

blueprints for trash cans

`coneModels = ['static.prop.constructioncone', 'static.prop.trafficcone01', 'static.prop.trafficcone02']`

blueprints for traffic cones

`debrisModels = ['static.prop.dirtdebris01', 'static.prop.dirtdebris02', 'static.prop.dirtdebris03']`

blueprints for road debris

`vendingMachineModels = ['static.prop.vendingmachine']`

blueprints for vending machines

`chairModels = ['static.prop.plasticchair']`

blueprints for chairs

`busStopModels = ['static.prop.busstop']`

blueprints for bus stops

`advertisementModels = ['static.prop.advertisement', 'static.prop.streetsign', 'static.prop.streetsign01', 'static.prop.streetsign04']`

blueprints for roadside billboards

`garbageModels = ['static.prop.colacan', 'static.prop.garbage01', 'static.prop.garbage02', 'static.prop.garbage03', 'static.prop.garbage04', 'static.prop.garbage05', 'static.prop.garbage06', 'static.prop.plasticbag', 'static.prop.trashbag']`

blueprints for pieces of trash

`containerModels = ['static.prop.container', 'static.prop.clothcontainer', 'static.prop.glasscontainer']`

blueprints for containers

`tableModels = ['static.prop.table', 'static.prop.plastictable']`

blueprints for tables

`barrierModels = ['static.prop.streetbarrier', 'static.prop.chainbarrier', 'static.prop.chainbarrierend']`

blueprints for traffic barriers

`plantpotModels = ['static.prop.plantpot01', 'static.prop.plantpot02', 'static.prop.plantpot03', 'static.prop.plantpot04', 'static.prop.plantpot05', 'static.prop.plantpot06', 'static.prop.plantpot07', 'static.prop.plantpot08']`

blueprints for flowerpots

`mailboxModels = ['static.prop.mailbox']`

blueprints for mailboxes

`gnomeModels = ['static.prop.gnome']`

blueprints for garden gnomes

`creasedboxModels = ['static.prop.creasedbox01', 'static.prop.creasedbox02', 'static.prop.creasedbox03']`

blueprints for creased boxes

`caseModels = ['static.prop.travelcase', 'static.prop.briefcase', 'static.prop.guitarcase']`

blueprints for briefcases, suitcases, etc.

`boxModels = ['static.prop.box01', 'static.prop.box02', 'static.prop.box03']`

blueprints for boxes

`benchModels = ['static.prop.bench01', 'static.prop.bench02', 'static.prop.bench03']`

blueprints for benches

`barrelModels = ['static.prop.barrel']`

blueprints for barrels

`atmModels = ['static.prop.atm']`

blueprints for ATMs

`kioskModels = ['static.prop.kiosk_01']`

blueprints for kiosks

`ironplateModels = ['static.prop.ironplank']`

blueprints for iron plates

`trafficwarningModels = ['static.prop.trafficwarning']`

blueprints for traffic warning signs

`walkerModels = ['walker.pedestrian.0001', 'walker.pedestrian.0002', 'walker.pedestrian.0003', 'walker.pedestrian.0004', 'walker.pedestrian.0005', 'walker.pedestrian.0006', 'walker.pedestrian.0007', 'walker.pedestrian.0008', 'walker.pedestrian.0009', 'walker.pedestrian.0010', 'walker.pedestrian.0011', 'walker.pedestrian.0012', 'walker.pedestrian.0013', 'walker.pedestrian.0014']`

blueprints for pedestrians

scenic.simulators.carla.misc

Module with auxiliary functions.

Summary of Module Members

Functions

compute_distance	Euclidean distance between 3D points
compute_magnitude_angle	Compute relative angle and distance between a target_location and a current_location
distance_vehicle	Returns the 2D distance from a waypoint to a vehicle
draw_waypoints	Draw a list of waypoints at a certain height given in z.
get_speed	Compute speed of a vehicle in Km/h.
is_within_distance	Check if a target object is within a certain distance from a reference object.
is_within_distance_ahead	Check if a target object is within a certain distance in front of a reference object.
positive	Return the given number if positive, else 0
vector	Returns the unit vector from location_1 to location_2

Member Details

draw_waypoints(*world*, *waypoints*, *z=0.5*)

Draw a list of waypoints at a certain height given in z.

param world
carla.world object

param waypoints
list or iterable container with the waypoints to draw

param z
height in meters

get_speed(*vehicle*)

Compute speed of a vehicle in Km/h.

param vehicle
the vehicle for which speed is calculated

return
speed as a float in Km/h

is_within_distance_ahead(*target_transform*, *current_transform*, *max_distance*)

Check if a target object is within a certain distance in front of a reference object.

Parameters

- **target_transform** – location of the target object
- **current_transform** – location of the reference object
- **orientation** – orientation of the reference object
- **max_distance** – maximum allowed distance

Returns
True if target object is within max_distance ahead of the reference object

`is_within_distance`(*target_location*, *current_location*, *orientation*, *max_distance*, *d_angle_th_up*, *d_angle_th_low=0*)

Check if a target object is within a certain distance from a reference object. A vehicle in front would be something around 0 deg, while one behind around 180 deg.

param target_location
location of the target object

param current_location
location of the reference object

param orientation
orientation of the reference object

param max_distance
maximum allowed distance

param d_angle_th_up
upper thereshold for angle

param d_angle_th_low
low thereshold for angle (optional, default is 0)

return
True if target object is within max_distance ahead of the reference object

`compute_magnitude_angle`(*target_location*, *current_location*, *orientation*)

Compute relative angle and distance between a target_location and a current_location

param target_location
location of the target object

param current_location
location of the reference object

param orientation
orientation of the reference object

return
a tuple composed by the distance to the object and the angle between both objects

`distance_vehicle`(*waypoint*, *vehicle_transform*)

Returns the 2D distance from a waypoint to a vehicle

param waypoint
actual waypoint

param vehicle_transform
transform of the target vehicle

`vector`(*location_1*, *location_2*)

Returns the unit vector from location_1 to location_2

param location_1, location_2
carla.Location objects

`compute_distance`(*location_1*, *location_2*)

Euclidean distance between 3D points

param location_1, location_2
3D points

`positive`(*num*)

Return the given number if positive, else 0

param num
value to check

scenic.simulators.carla.model

Scenic world model for traffic scenarios in CARLA.

The model currently supports vehicles, pedestrians, and props. It implements the basic *Car* and *Pedestrian* classes from the *scenic.domains.driving* domain, while also providing convenience classes for specific types of objects like bicycles, traffic cones, etc. Vehicles and pedestrians support the basic actions and behaviors from the driving domain; several more are automatically imported from *scenic.simulators.carla.actions* and *scenic.simulators.carla.behaviors*.

The model defines several global parameters, whose default values can be overridden in scenarios using the `param` statement or on the command line using the *--param* option:

Global Parameters

- **carla_map** (*str*) – Name of the CARLA map to use, e.g. 'Town01'. Can also be set to `None`, in which case CARLA will attempt to create a world from the **map** file used in the scenario (which must be an `.xodr` file).
- **timestep** (*float*) – Timestep to use for simulations (i.e., how frequently Scenic interrupts CARLA to run behaviors, check requirements, etc.), in seconds. Default is 0.1 seconds.
- **weather** (*str or dict*) – Weather to use for the simulation. Can be either a string identifying one of the CARLA weather presets (e.g. 'ClearSunset') or a dictionary specifying all the weather parameters (see carla.WeatherParameters). Default is a uniform distribution over all the weather presets.
- **address** (*str*) – IP address at which to connect to CARLA. Default is localhost (127.0.0.1).
- **port** (*int*) – Port on which to connect to CARLA. Default is 2000.
- **timeout** (*float*) – Maximum time to wait when attempting to connect to CARLA, in seconds. Default is 10.
- **render** (*int*) – Whether or not to have CARLA create a window showing the simulations from the point of view of the ego object: 1 for yes, 0 for no. Default 1.
- **record** (*str*) – If nonempty, folder in which to save CARLA record files for replaying the simulations.

Summary of Module Members

Functions

freezeTrafficLights	Freezes all traffic lights in the scene.
getClosestTrafficLightStatus	
getTrafficLightStatus	
setAllIntersectionTrafficLightsStatus	
setClosestTrafficLightStatus	
setTrafficLightStatus	
unfreezeTrafficLights	Unfreezes all traffic lights in the scene.
withinDistanceToRedYellowTrafficLight	
withinDistanceToTrafficLight	

Classes

ATM	
Advertisement	
Barrel	
Barrier	
Bench	
Bicycle	
Box	
BusStop	
Car	A car.
CarlaActor	Abstract class for CARLA objects.
Case	
Chair	
Cone	
Container	
CreasedBox	

continues on next page

Table 1 – continued from previous page

Debris	
Garbage	
Gnome	
IronPlate	
Kiosk	
Mailbox	
Motorcycle	
NPCCar	
Pedestrian	A pedestrian.
PlantPot	
Prop	Abstract class for props, i.e. non-moving objects.
Table	
TrafficWarning	
Trash	
Truck	
Vehicle	Abstract class for steerable vehicles.
VendingMachine	

Member Details

class CarlaActor <specifiers>

Bases: *DrivingObject*

Abstract class for CARLA objects.

Properties

- **carlaActor** (*dynamic*) – Set during simulations to the `carla.Actor` representing this object.
- **blueprint** (*str*) – Identifier of the CARLA blueprint specifying the type of object.
- **rolename** (*str*) – Can be used to differentiate specific actors during runtime. Default value `None`.
- **physics** (*bool*) – Whether physics is enabled for this object in CARLA. Default true.

class Vehicle <specifiers>

Bases: *Vehicle*, *CarlaActor*, *Steers*

Abstract class for steerable vehicles.

class Car <specifiers>

Bases: *Vehicle*

A car.

The default `blueprint` (see *CarlaActor*) is a uniform distribution over the blueprints listed in *scenic.simulators.carla.blueprints.carModels*.

class Pedestrian <specifiers>

Bases: *Pedestrian*, *CarlaActor*, *Walks*

A pedestrian.

The default `blueprint` (see *CarlaActor*) is a uniform distribution over the blueprints listed in *scenic.simulators.carla.blueprints.walkerModels*.

class Prop <specifiers>

Bases: *CarlaActor*

Abstract class for props, i.e. non-moving objects.

Properties

- **heading** (*float*) – Default value overridden to be uniformly random.
- **physics** (*bool*) – Default value overridden to be false.

freezeTrafficLights()

Freezes all traffic lights in the scene.

Frozen traffic lights can be modified by the user but the time will not update them until unfrozen.

unfreezeTrafficLights()

Unfreezes all traffic lights in the scene.

_getClosestTrafficLight(*vehicle*, *distance=100*)

Returns the closest traffic light affecting 'vehicle', up to a maximum of 'distance'

scenic.simulators.carla.simulator

Simulator interface for CARLA.

Summary of Module Members

Classes

CarlaSimulation	
CarlaSimulator	Implementation of *Simulator* for CARLA.

Member Details

class CarlaSimulator(*carla_map*, *map_path*, *address='127.0.0.1'*, *port=2000*, *timeout=10*, *render=True*, *record=''*, *timestep=0.1*, *traffic_manager_port=None*)

Bases: *DrivingSimulator*

Implementation of *Simulator* for CARLA.

scenic.simulators.gta

Scenic world model for Grand Theft Auto V (GTAV).

Importing scenes generated using this model into GTA V requires a GTA V plugin, which you can find here.

center_detection	This file contains helper functions
img_modf	This file has basic image modification functions
interface	Python supporting code for the GTA model.
map	
messages	
model	World model for GTA.

scenic.simulators.gta.center_detection

This file contains helper functions

Summary of Module Members

Functions

compute_bb	
compute_gradient_sobel	
compute_midpoints	
find_center	Find which edge x lies in
generate_circle	
generate_connected_edges	
generate_neighbors	
transform_center	

Classes

EdgeData

Member Details

find_center(*x*, *theta*, *collected_edges*, *all_edges*, *num_samples*, *bw_image*)

Find which edge x lies in

class EdgeData(*init_theta*, *tangent*, *opp_loc*, *mid_loc*)

Bases: NamedTuple

Parameters

- **init_theta** (*float*) –
- **tangent** (*float*) –
- **opp_loc** (*Tuple[float, float]*) –
- **mid_loc** (*Tuple[float, float]*) –

init_theta: float

Alias for field number 0

tangent: float

Alias for field number 1

opp_loc: Tuple[float, float]

Alias for field number 2

mid_loc: Tuple[float, float]

Alias for field number 3

_asdict()

Return a new dict which maps field names to their values.

classmethod _make(*iterable*)

Make a new EdgeData object from a sequence or iterable

_replace(***kwds*)

Return a new EdgeData object replacing specified fields with new values

scenic.simulators.gta.img_modf

This file has basic image modification functions

Summary of Module Members

Functions

convert_black_white
get_edges

Member Details

scenic.simulators.gta.interface

Python supporting code for the GTA model.

Summary of Module Members

Classes

CarModel	A model of car in GTA.
GTA	
Map	Represents roads and obstacles in GTA, extracted from a map image.
MapWorkspace	Workspace whose rendering is handled by a Map

Member Details

class Map(*imagePath*, *Ax*, *Ay*, *Bx*, *By*)

Represents roads and obstacles in GTA, extracted from a map image.

This code handles images from the GTA V Interactive Map, rendered with the "Road" setting.

Parameters

- **imagePath** (*str*) – path to image file
- **Ax** (*float*) – width of one pixel in GTA coordinates
- **Ay** (*float*) – height of one pixel in GTA coordinates
- **Bx** (*float*) – GTA X-coordinate of bottom-left corner of image
- **By** (*float*) – GTA Y-coordinate of bottom-left corner of image

class MapWorkspace(*mappy*, *region*)

Bases: *Workspace*

Workspace whose rendering is handled by a Map

class CarModel(*name*, *width*, *length*, *viewAngle=1.5707963267948966*)

A model of car in GTA.

Attributes

- **name** (*str*) – name of model in GTA
- **width** (*float*) – width of this model of car
- **length** (*float*) – length of this model of car
- **viewAngle** (*float*) – view angle in radians (default is 90 degrees)

Class Attributes

models – dict mapping model names to the corresponding *CarModel*

Parameters

- **name** (*str*) –
- **width** (*float*) –
- **length** (*float*) –
- **viewAngle** (*float*) –

scenic.simulators.gta.map

Summary of Module Members

Functions

setLocalMap

Member Details

scenic.simulators.gta.messages

Summary of Module Members

Functions

frame2numpy
obj_dict

Classes

Commands
Config
Dataset
Formal_Config
Formal_Configs
Scenario
Start
Stop
Vehicle

Member Details

scenic.simulators.gta.model

World model for GTA.

Summary of Module Members

Module Attributes

roadDirection	Vector field representing the nominal traffic direction at a point on the road
road	Region representing the roads in the GTA map.
curb	Region representing the curbs in the GTA map.

Functions

createPlatoonAt	Create a platoon starting from the given car.

Classes

Bus	Convenience subclass for buses.
Car	Scenic class for cars.
Compact	Convenience subclass for compact cars.
EgoCar	Convenience subclass with defaults for ego cars.

Member Details

roadDirection

Vector field representing the nominal traffic direction at a point on the road

road

Region representing the roads in the GTA map.

curb

Region representing the curbs in the GTA map.

class Car <specifiers>

Bases: *Object2D*

Scenic class for cars.

Properties

- **position** – The default position is uniformly random over the *road*.
- **heading** – The default heading is aligned with *roadDirection*, plus an offset given by `roadDeviation`.
- **roadDeviation** (*float*) – Relative heading with respect to the road direction at the *Car*'s position. Used by the default value for `heading`.
- **model** (*CarModel*) – Model of the car.
- **color** (`Color` or RGB tuple) – Color of the car.

class EgoCar <specifiers>

Bases: *Car*

Convenience subclass with defaults for ego cars.

class Bus <specifiers>

Bases: *Car*

Convenience subclass for buses.

class Compact <specifiers>

Bases: *Car*

Convenience subclass for compact cars.

createPlatoonAt(*car*, *numCars*, *model=None*, *dist=Range(2.0, 8.0)*, *shift=Range(-0.5, 0.5)*, *wiggle=0*)

Create a platoon starting from the given car.

scenic.simulators.lgsvl

Interface to the LGSVL driving simulator.

This interface has been tested with LGSVL version 2020.06. It supports dynamic scenarios involving vehicles and pedestrians.

The interface implements the *scenic.domains.driving* abstract domain, so any object types, behaviors, utility functions, etc. from that domain may be used freely.

actions	Actions for agents in the LGSVL model.
behaviors	Behaviors for dynamic agents in LGSVL.
model	Scenic world model for the LGSVL Simulator.
simulator	Dynamic simulator interface for LGSVL.
utils	Common LGSVL interface.

scenic.simulators.lgsvl.actions

Actions for agents in the LGSVL model.

Summary of Module Members

Classes

CancelWaypointsAction
FollowWaypointsAction
SetDestinationAction
TrackWaypointsAction

Member Details

scenic.simulators.lgsvl.behaviors

Behaviors for dynamic agents in LGSVL.

scenic.simulators.lgsvl.model

Scenic world model for the LGSVL Simulator.

Summary of Module Members

Functions

LGSVLSimulator	

Classes

ApolloCar	
Bus	
Car	alias of EgoCar
EgoCar	
LGSVLObject	
NPCCar	
Pedestrian	
Vehicle	
Waypoint	

Member Details

scenic.simulators.lgsvl.simulator

Dynamic simulator interface for LGSVL.

Summary of Module Members

Classes

LGSVLSimulation	Subclass of *Simulation* for LGSVL.
LGSVLSimulator	A connection to an instance of LGSVL.

Member Details

class LGSVLSimulator(*sceneID*, *address='localhost'*, *port=8181*, *alwaysReload=False*)

Bases: *Simulator*

A connection to an instance of LGSVL.

See the SVL documentation for details on how to set the parameters below.

Uses a default timestep of 0.1 seconds.

Parameters

- **sceneID** (*str*) – Identifier for the map ("scene") to load in SVL.
- **address** (*str*) – Address where SVL is running.
- **port** (*int*) – Port on which to connect to SVL.
- **alwaysReload** (*bool*) – Whether to force reloading the map upon connecting, even if the simulator already has the desired map loaded.

class LGSVLSimulation(*scene*, *client*, *, *timestep*, **kwargs*)

Bases: *Simulation*

Subclass of *Simulation* for LGSVL.

initApolloFor(*obj*, *lgsvlObj*)

Initialize Apollo for an ego vehicle.

Uses LG's interface which injects packets into Dreamview.

scenic.simulators.lgsvl.utils

Common LGSVL interface.

Summary of Module Members

Functions

gpsToScenicPosition	Convert GPS positions to Scenic positions.
lgsvlToScenicAngularSpeed	
lgsvlToScenicElevation	Convert LGSVL positions to Scenic elevations.
lgsvlToScenicPosition	Convert LGSVL positions to Scenic positions.
lgsvlToScenicRotation	Convert LGSVL rotations to Scenic headings.
scenicToLGSVLPosition	
scenicToLGSVLRotation	

Member Details

lgsvlToScenicPosition(*pos*)

Convert LGSVL positions to Scenic positions.

gpsToScenicPosition(*northing*, *easting*)

Convert GPS positions to Scenic positions.

lgsvlToScenicElevation(*pos*)

Convert LGSVL positions to Scenic elevations.

lgsvlToScenicRotation(*rot*)

Convert LGSVL rotations to Scenic headings.

Drops all but the Y component.

scenic.simulators.newtonian

Simple Newtonian physics simulator.

This simulator allows dynamic scenarios to be tested without installing an external simulator. It is currently very simplistic (e.g. not modeling collisions).

The simulator provides two world models: a generic one, and a more specialized model supporting traffic scenarios using the *scenic.domains.driving* abstract domain.

driving_model	Scenic world model for traffic scenarios in the Newtonian simulator.
model	Scenic world model for the Newtonian simulator.
simulator	Newtonian simulator implementation.

scenic.simulators.newtonian.driving_model

Scenic world model for traffic scenarios in the Newtonian simulator.

This model implements the basic *Car* class from the *scenic.domains.driving* domain. Vehicles support the basic actions and behaviors from the driving domain.

A path to a map file for the scenario should be provided as the `map` global parameter; see the driving domain's documentation for details.

Summary of Module Members

Classes

Car	
Debris	Abstract class for debris scattered randomly in the workspace.
NewtonianActor	
Pedestrian	
Vehicle	

Member Details

class Debris <specifiers>

Bases: *Object2D*

Abstract class for debris scattered randomly in the workspace.

scenic.simulators.newtonian.model

Scenic world model for the Newtonian simulator.

This is a completely generic model that does not assume the scenario takes place in a road network (unlike *scenic.simulators.newtonian.driving_model*).

scenic.simulators.newtonian.simulator

Newtonian simulator implementation.

Summary of Module Members

Classes

NewtonianSimulation	Implementation of *Simulation* for the Newtonian simulator.
NewtonianSimulator	Implementation of *Simulator* for the Newtonian simulator.

Member Details

class NewtonianSimulator(*network=None*, *render=False*)

Bases: *DrivingSimulator*

Implementation of *Simulator* for the Newtonian simulator.

Parameters

- **network** (*Network*) – road network to display in the background, if any.
- **render** (*bool*) – whether to render the simulation in a window.

Changed in version 3.0: The **timestep** argument is removed: it can be specified when calling *simulate* instead. The default timestep for the Newtonian simulator when not otherwise specified is still 0.1 seconds.

class NewtonianSimulation(*scene*, *network*, *render*, *timestep*, ***kwargs*)

Bases: *DrivingSimulation*

Implementation of *Simulation* for the Newtonian simulator.

scenic.simulators.utils

Various utilities useful across multiple simulators.

colors	A basic color type.

scenic.simulators.utils.colors

A basic color type.

This used for example to represent car colors in the abstract driving domain, as well as in the interfaces to GTA and Webots.

Summary of Module Members

Classes

Color	A color as an RGB tuple.
ColorMutator	Mutator that adds Gaussian HSL noise to the `color` property.
NoisyColorDistribution	A distribution given by HSL noise around a base color.

Member Details

class Color(*r*, *g*, *b*)

Bases: *Color*

A color as an RGB tuple.

static uniformColor()

Return a uniformly random color.

static defaultCarColor()

Default color distribution for cars.

The distribution starts with a base distribution over 9 discrete colors, then adds Gaussian HSL noise. The base distribution uses color popularity statistics from a 2012 DuPont survey.

class NoisyColorDistribution(*baseColor*, *hueNoise*, *satNoise*, *lightNoise*)

Bases: *Distribution*

A distribution given by HSL noise around a base color.

Parameters

- **baseColor** (*RGB tuple*) – base color
- **hueNoise** (*float*) – noise to add to base hue
- **satNoise** (*float*) – noise to add to base saturation
- **lightNoise** (*float*) – noise to add to base lightness

class ColorMutator

Bases: *Mutator*

Mutator that adds Gaussian HSL noise to the `color` property.

scenic.simulators.webots

Scenic world models for the Webots robotics simulator.

This module contains common code for working with Webots, e.g. parsing WBT files, as well as a generic dynamic simulator interface and world model for Webots. More detailed world models for particular types of scenarios are in submodules.

actions	Actions for dynamic agents in Webots simulations.
guideways	World model for road intersection scenarios in Webots.
model	Generic Scenic world model for the Webots simulator.
road	World model and associated code for traffic scenarios in Webots.
simulator	Interface to Webots for dynamic simulations.
utils	Various utilities for working with Webots scenarios.
WBTLexer	
WBTParser	
WBTVisitor	
world_parser	Parser for WBT files using ANTLR.

scenic.simulators.webots.actions

Actions for dynamic agents in Webots simulations.

Summary of Module Members

Classes

ApplyForceAction	Apply a given force to the object.
OffsetAction	Move an object by the given offset relative to its current heading.
WriteFileAction	Pickle the given data and write the result to a file.

Member Details

class OffsetAction(*offset*)

Bases: *Action*

Move an object by the given offset relative to its current heading.

class ApplyForceAction(*force*, *relative=False*)

Bases: *Action*

Apply a given force to the object.

class WriteFileAction(*path*, *data*)

Bases: *Action*

Pickle the given data and write the result to a file.

For use in communication with external controllers or other code.

scenic.simulators.webots.guideways

World model for road intersection scenarios in Webots.

This is a more specialized version of the *scenic.simulators.webots.road* model which also includes guideway information from the Intelligent Intersections Toolkit.

interface
intersection
model

scenic.simulators.webots.guideways.interface

Summary of Module Members

Functions

localize
projectionAt
toWebots

Classes

Bordered
ConflictZone
Crosswalk
Guideway
Intersection
IntersectionWorkspace

Member Details

scenic.simulators.webots.guideways.intersection

Summary of Module Members

Functions

setLocalIntersection

Member Details

scenic.simulators.webots.guideways.model

Summary of Module Members

Classes

Car
Marker

Member Details

scenic.simulators.webots.model

Generic Scenic world model for the Webots simulator.

This model provides a general type of object *WebotsObject* corresponding to a node in the Webots scene tree, as well as a few more specialized objects.

Scenarios using this model cannot be launched directly from the command line using the *--simulate* option. Instead, Webots should be started first, with a `.wbt` file that includes nodes for all the objects in the scenario (see the *WebotsObject* documentation for how to specify which objects correspond to which nodes). A supervisor node can then invoke Scenic to compile the scenario and run dynamic simulations: see *scenic.simulators.webots.simulator* for details.

Summary of Module Members

Functions

is2DMode

Classes

Ground	Special kind of object representing a (possibly irregular) ground surface.
Hill	*Terrain* shaped like a Gaussian.
Terrain	Abstract class for objects added together to make a *Ground*.
WebotsObject	Abstract class for Webots objects.

Member Details

class WebotsObject <specifiers>

Bases: *Object*

Abstract class for Webots objects.

There several ways to specify which Webots node this object corresponds to:

- Set the `webotsName` property to the DEF name of the Webots node, which must already exist in the world loaded into Webots.
- Set the `webotsType` property to a prefix like 'ROCK': the interface will then search for nodes called 'ROCK_0', 'ROCK_1', etc. Again the nodes must already exist in the world loaded into Webots.
- Set the `webotsAdhoc` property to a dictionary of parameters. This will cause Scenic to dynamically create an Object in Webots, according to the parameters in the dictionary. **This is currently the only way to create objects in Webots that do not correspond to an existing node**. The parameters that can be contained in the dictionary are:
  - `physics`: Whether or not physics should be enabled for this object. Default value is *True*.

Properties

- **elevation** (*float or None; dynamic*) – default `None` (see above).
- **requireVisible** (*bool*) – Default value `False` (overriding the default from *Object*).
- **webotsAdhoc** (*None | dict*) – None implies this object is not Adhoc. A dictionary implies this is an object that Scenic should create in Webots.. If a dictionary, provides parameters for how to instantiate the adhoc object. See *scenic.simulators.webots.model* for more details.
- **webotsName** (*str*) – 'DEF' name of the Webots node to use for this object.
- **webotsType** (*str*) – If `webotsName` is not set, the first available node with 'DEF' name consisting of this string followed by '_0', '_1', etc. will be used for this object.
- **webotsObject** – Is set at runtime to a handle to the Webots node for the object, for use with the Supervisor API. Primarily for internal use.
- **controller** (*str or None*) – name of the Webots controller to use for this object, if any (instead of a Scenic behavior).
- **resetController** (*bool*) – Whether to restart the controller for each simulation (default `True`).
- **positionOffset** (*Vector*) – Offset to add when computing the object's position in Webots; for objects whose Webots `translation` field is not aligned with the center of the object.
- **rotationOffset** (*tuple[float, float, float]*) – Offset to add when computing the object's orientation in Webots; for objects whose front is not aligned with the Webots North axis.
- **density** (*float*) – Density of this object in kg/m^3. The corresponding Webots object must have the `density` field.

class Ground <specifiers>

Bases: *WebotsObject*

Special kind of object representing a (possibly irregular) ground surface.

Implemented using an ElevationGrid node in Webots.

Attributes

- **allowCollisions** (*bool*) – default value `True` (overriding default from *Object*).
- **webotsName** (*str*) – default value 'Ground'

class Terrain <specifiers>

Bases: *Object*

Abstract class for objects added together to make a *Ground*.

This is not a *WebotsObject* since it doesn't actually correspond to a Webots node. Only the overall *Ground* has a node.

class Hill <specifiers>

Bases: *Terrain*

Terrain shaped like a Gaussian.

Attributes

- **height** (*float*) – height of the hill (default 1).
- **spread** (*float*) – standard deviation as a fraction of the hill's size (default 3).

scenic.simulators.webots.road

World model and associated code for traffic scenarios in Webots.

This model handles Webots world files generated from Open Street Map data using the Webots OSM importer.

car_models	Car models built into Webots.
interface	Python library supporting the main Scenic module.
model	Scenic world model for traffic scenarios in Webots.
world	Stub to allow changing the Webots world without changing the model.

scenic.simulators.webots.road.car_models

Car models built into Webots.

Summary of Module Members

Classes

CarModel

Member Details

class CarModel(*name: str*, *width: float*, *length: float*, *height: float*)

Parameters

- **name** (*str*) –
- **width** (*float*) –
- **length** (*float*) –
- **height** (*float*) –

scenic.simulators.webots.road.interface

Python library supporting the main Scenic module.

Summary of Module Members

Functions

polygonWithPoints	
regionWithPolygons	
scenicToWebotsPosition	Convert a Scenic position to a Webots position.
scenicToWebotsRotation	Convert a Scenic heading to a Webots rotation vector.
webotsToScenicPosition	Convert a Webots position to a Scenic position.
webotsToScenicRotation	Convert a Webots rotation vector to a Scenic heading.

Classes

Crossroad	OSM crossroads
OSMObject	Objects with OSM id tags
PedestrianCrossing	PedestrianCrossing nodes
Road	OSM roads
WebotsWorkspace	

Member Details

class OSMObject(*attrs*)

Objects with OSM id tags

class Road(*attrs*, *driveOnLeft=False*)

Bases: *OSMObject*

OSM roads

class Crossroad(*attrs*)

Bases: *OSMObject*

OSM crossroads

class PedestrianCrossing(*attrs*)

PedestrianCrossing nodes

webotsToScenicPosition(*pos*)

Convert a Webots position to a Scenic position. Drops the Webots Y coordinate.

Deprecated since version 2.1.0: Use `WebotsCoordinateSystem` instead.

scenicToWebotsPosition(*pos*, *y=0*, *coordinateSystem='ENU'*)

Convert a Scenic position to a Webots position.

Deprecated since version 2.1.0: Use `WebotsCoordinateSystem` instead.

webotsToScenicRotation(*rot*, *tolerance2D=None*)

Convert a Webots rotation vector to a Scenic heading. Assumes the object lies in the Webots X-Z plane, with a rotation axis close to the Y axis. If `tolerance2D` is given, returns `None` if the orientation of the object is not sufficiently close to being 2D.

Deprecated since version 2.1.0: Use `WebotsCoordinateSystem` instead.

scenicToWebotsRotation(*heading*)

Convert a Scenic heading to a Webots rotation vector.

Deprecated since version 2.1.0: Use `WebotsCoordinateSystem` instead.

scenic.simulators.webots.road.model

Scenic world model for traffic scenarios in Webots.

Summary of Module Members

Classes

BmwX5
Bus
Car
CitroenCZero
LincolnMKZ
Motorcycle
OilBarrel
Pedestrian
RangeRoverSportSVR
SmallCar
SolidBox
ToyotaPrius
Tractor
TrafficCone
Truck
WebotsObject
WorkBarrier

Member Details

scenic.simulators.webots.road.world

Stub to allow changing the Webots world without changing the model.

Summary of Module Members

Module Attributes

worldPath	Path to the WBT file to load the Webots world from

Functions

setLocalWorld	Select a WBT file relative to the given module.

Member Details

worldPath = '../tests/simulators/webots/road/simple.wbt'

Path to the WBT file to load the Webots world from

setLocalWorld(*module*, *relpath*)

Select a WBT file relative to the given module.

This function is intended to be used with `__file__` as the *module*.

scenic.simulators.webots.simulator

Interface to Webots for dynamic simulations.

This interface is intended to be instantiated from inside the controller script of a Webots Robot node with the `supervisor` field set to true. Such a script can create a *WebotsSimulator* (passing in a reference to the supervisor node) and then call its *simulate* method as usual to run a simulation. For an example, see `examples/webots/generic/controllers/scenic_supervisor.py`.

Scenarios written for this interface should use our generic Webots world model *scenic.simulators.webots.model* or a model derived from it. Objects which are instances of *WebotsObject* will be matched to Webots nodes; see the model documentation for details.

Summary of Module Members

Functions

getFieldSafe	Get field from webots object.
isPhysicsEnabled	Whether or not physics is enabled for this *WebotsObject*

Classes

WebotsSimulation	*Simulation* object for Webots.
WebotsSimulator	*Simulator* object for Webots.

Member Details

class WebotsSimulator(*supervisor*)

Bases: *Simulator*

Simulator object for Webots.

Parameters
supervisor – Supervisor node handle from the Webots Python API.

class WebotsSimulation(*scene*, *supervisor*, *coordinateSystem=<scenic.simulators.webots.utils.WebotsCoordinateSystem object>*, ***, *timestep*, ***kwargs*)

Bases: *Simulation*

Simulation object for Webots.

Attributes
supervisor – Webots supervisor node used for the simulation. This is exposed for the use of scenarios which need to call Webots APIs directly; e.g. `simulation().supervisor.setLabel(...)`.

getFieldSafe(*webotsObject*, *fieldName*)

Get field from webots object. Return null if no such field exists.

Needed to workaround this issue (https://github.com/cyberbotics/webots/issues/5646)

Parameters

- **webotsObject** – webots object
- **fieldName** – name of the field to look for

Returns
Field|None – Field object if the field with the given name exists. None otherwise.

isPhysicsEnabled(*webotsObject*)

Whether or not physics is enabled for this *WebotsObject*

scenic.simulators.webots.utils

Various utilities for working with Webots scenarios.

Summary of Module Members

Module Attributes

ENU	The ENU coordinate system (the Webots default).
NUE	The NUE coordinate system.
EUN	The EUN coordinate system.

Classes

WebotsCoordinateSystem	A Webots coordinate system into which Scenic positions can be converted.

Member Details

class WebotsCoordinateSystem(*system='ENU'*)

A Webots coordinate system into which Scenic positions can be converted.

See the Webots documentation of WorldInfo.coordinateSystem for a discussion of the possible coordinate systems. Since Webots R2022a, the default coordinate axis convention is ENU (X-Y-Z=East-North-Up), which is the same as Scenic's.

positionToScenic(*pos*)

Convert a Webots position to a Scenic position.

positionFromScenic(*pos*)

Convert a Scenic position to a Webots position.

ENU = <scenic.simulators.webots.utils.WebotsCoordinateSystem object>

The ENU coordinate system (the Webots default).

NUE = <scenic.simulators.webots.utils.WebotsCoordinateSystem object>

The NUE coordinate system.

EUN = <scenic.simulators.webots.utils.WebotsCoordinateSystem object>

The EUN coordinate system.

scenic.simulators.webots.WBTLexer

Summary of Module Members

Functions

serializedATN

Classes

WBTLexer

Member Details

scenic.simulators.webots.WBTParser

Summary of Module Members

Functions

serializedATN

Classes

WBTParser

Member Details

scenic.simulators.webots.WBTVisitor

Summary of Module Members

Classes

WBTVisitor

Member Details

scenic.simulators.webots.world_parser

Parser for WBT files using ANTLR.

The ANTLR parser itself, consisting of the *WBTLexer.py*, *WBTParser.py*, and *WBTVisitor.py* files, is autogenerated from *WBT.g4*.

Summary of Module Members

Functions

findNodeTypesIn	Find all nodes of the given types in a world
parse	Parse a world from a WBT file

Classes

ErrorReporter	ANTLR listener for reporting parse errors
Evaluator	Constructs an object representing the given value from the parse tree
Node	A generic VRML node

Member Details

class Node(*nodeType*, *attrs*)

A generic VRML node

class ErrorReporter

Bases: `ErrorListener`

ANTLR listener for reporting parse errors

class Evaluator(*nodeClasses*)

Bases: `WBTVisitor`

Constructs an object representing the given value from the parse tree

parse(*path*)

Parse a world from a WBT file

findNodeTypesIn(*types*, *world*, *nodeClasses={}*)

Find all nodes of the given types in a world

scenic.simulators.xplane

Scenic world model for the X-Plane flight simulator.

See the VerifAI distribution for examples of how to use Scenic with X-Plane.

model	Scenic world model for the X-Plane simulator.

scenic.simulators.xplane.model

Scenic world model for the X-Plane simulator.

At the moment this is extremely simple, since the current interface does not allow changing the type of aircraft, adding other objects, etc.

Summary of Module Members

Classes

Plane	Placeholder object for the plane.

Member Details

class Plane <specifiers>

Bases: *Object*

Placeholder object for the plane.

scenic.syntax

The Scenic compiler and associated support code.

ast	
compiler	
parser	
pygment	Pygments lexer and style for Scenic.
relations	Extracting relations (for later pruning) from the syntax of requirements.
translator	Translator turning Scenic programs into Scenario objects.
veneer	Python implementations of Scenic language constructs.

scenic.syntax.ast

Summary of Module Members

Classes

AST	Scenic AST base class
Abort	
Above	
Additive	
AheadOf	
AltitudeFromOp	
Always	
AngleFromOp	
ApparentHeadingOp	
ApparentlyFacingSpecifier	
AtSpecifier	
Back	Represents position of `back of` operator
BackLeft	Represents position of `back left of` operator
BackRight	Represents position of `back right of` operator
BehaviorDef	
Behind	
Below	
BeyondSpecifier	
Bottom	Represents position of `bottom of` operator
BottomBackLeft	Represents position of `bottom back left of` operator
BottomBackRight	Represents position of `bottom back right of` operator
BottomFrontLeft	Represents position of `bottom front left of` operator
BottomFrontRight	Represents position of `bottom front right of` operator
CanSeeOp	
ContainedInSpecifier	

continues on next page

Table 2 – continued from previous page

DegOp	
DirectionOfSpecifier	
DistanceFromOp	
DistancePastOp	
Do	
DoChoose	
DoFor	
DoShuffle	
DoUntil	
Dynamic	
Ego	*ego* tracked assign target
Eventually	
FacingAwayFromSpecifier	
FacingDirectlyAwayFromSpecifier	
FacingDirectlyTowardSpecifier	
FacingSpecifier	
FacingTowardSpecifier	
FieldAtOp	
Final	
FollowOp	
FollowingSpecifier	
Front	Represents position of `front of` operator
FrontLeft	Represents position of `front left of` operator
FrontRight	Represents position of `front right of` operator
ImpliesOp	
InSpecifier	
InitialScenario	

continues on next page

Table 2 – continued from previous page

InterruptWhenHandler	
Invariant	
Left	Represents position of `left of` operator
LeftOf	
Model	
MonitorDef	
Mutate	
New	
Next	
NotVisibleFromOp	
NotVisibleOp	
NotVisibleSpecifier	
OffsetAlongOp	
OffsetAlongSpecifier	
OffsetBySpecifier	
OnSpecifier	
Override	
Param	*param* statements
PositionOfOp	
Precondition	
PropertyDef	
Record	
RecordFinal	
RecordInitial	
RelativeHeadingOp	
RelativePositionOp	

continues on next page

Table 2 – continued from previous page

RelativeToOp	
Require	
RequireMonitor	
Right	Represents position of `right of` operator
RightOf	
ScenarioDef	
Seconds	
Simulator	
Steps	
Take	
Terminate	
TerminateAfter	
TerminateSimulation	
TerminateSimulationWhen	
TerminateWhen	
Top	Represents position of `top of` operator
TopBackLeft	Represents position of `top back left of` operator
TopBackRight	Represents position of `top back right of` operator
TopFrontLeft	Represents position of `top front left of` operator
TopFrontRight	Represents position of `top front right of` operator
TrackedAssign	
TryInterrupt	Scenic AST node that represents try-interrupt statements
UntilOp	
VectorOp	
VisibleFromOp	
VisibleOp	
VisibleSpecifier	
Wait	
WithSpecifier	

continues on next page

Table 2 – continued from previous page

Workspace	workspace tracked assign target
parameter	represents a parameter that is defined with *param* statements

Member Details

class AST(**args*, ***kwargs*)

Bases: AST

Scenic AST base class

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class TryInterrupt(*body*, *interrupt_when_handlers*, *except_handlers*, *orelse*, *finalbody*, **args*, ***kwargs*)

Bases: *AST*

Scenic AST node that represents try-interrupt statements

Parameters

- **body** (*List[stmt]*) –
- **interrupt_when_handlers** (*List[InterruptWhenHandler]*) –
- **except_handlers** (*List[ExceptHandler]*) –
- **orelse** (*List[stmt]*) –
- **finalbody** (*List[AST]*) –
- **args** (*any*) –
- **kwargs** (*any*) –

class Ego(**args*, ***kwargs*)

Bases: *AST*

ego tracked assign target

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class Workspace(**args*, ***kwargs*)

Bases: *AST*

workspace tracked assign target

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class Param(*elts*, **args*, ***kwargs*)

Bases: *AST*

param statements

Parameters

- **elts** (*List[parameter]*) –
- **args** (*any*) –
- **kwargs** (*any*) –

class parameter(*identifier*, *value*, **args*, ***kwargs*)

Bases: *AST*

represents a parameter that is defined with *param* statements

Parameters

- **identifier** (*str*) –
- **value** (AST) –
- **args** (*any*) –
- **kwargs** (*any*) –

class Front(**args*, ***kwargs*)

Bases: *AST*

Represents position of `front of` operator

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class Back(**args*, ***kwargs*)

Bases: *AST*

Represents position of `back of` operator

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class Left(**args*, ***kwargs*)

Bases: *AST*

Represents position of `left of` operator

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class Right(**args*, ***kwargs*)

Bases: *AST*

Represents position of `right of` operator

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class Top(**args*, ***kwargs*)

Bases: *AST*

Represents position of `top of` operator

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class Bottom(**args*, ***kwargs*)

Bases: *AST*

Represents position of `bottom of` operator

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class FrontLeft(**args*, ***kwargs*)

Bases: *AST*

Represents position of `front left of` operator

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class FrontRight(**args*, ***kwargs*)

Bases: *AST*

Represents position of `front right of` operator

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class BackLeft(**args*, ***kwargs*)

Bases: *AST*

Represents position of `back left of` operator

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class BackRight(**args*, ***kwargs*)

Bases: *AST*

Represents position of `back right of` operator

Parameters

- **args** (*any*) –

- **kwargs** (*any*) –

class TopFrontLeft(**args*, ***kwargs*)

Bases: *AST*

Represents position of `top front left of` operator

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class TopFrontRight(**args*, ***kwargs*)

Bases: *AST*

Represents position of `top front right of` operator

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class TopBackLeft(**args*, ***kwargs*)

Bases: *AST*

Represents position of `top back left of` operator

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class TopBackRight(**args*, ***kwargs*)

Bases: *AST*

Represents position of `top back right of` operator

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class BottomFrontLeft(**args*, ***kwargs*)

Bases: *AST*

Represents position of `bottom front left of` operator

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class BottomFrontRight(**args*, ***kwargs*)

Bases: *AST*

Represents position of `bottom front right of` operator

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class BottomBackLeft(*args, **kwargs)

Bases: AST

Represents position of `bottom back left of` operator

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

class BottomBackRight(*args, **kwargs)

Bases: AST

Represents position of `bottom back right of` operator

Parameters

- **args** (*any*) –
- **kwargs** (*any*) –

scenic.syntax.compiler

Summary of Module Members

Functions

compileScenicAST	Compiles Scenic AST to Python AST
unquote	

Classes

AttributeFinder	Utility class for finding all referenced attributes of a given name.
Context	An enumeration.
LocalFinder	Utility class for finding all local variables of a code block.
PropositionTransformer	
ScenicToPythonTransformer	
Transformer	Subclass of `ast.NodeTransformer` with a method for raising syntax errors.

Member Details

compileScenicAST(*scenicAST*, ***, *filename='<unknown>'*, *inBehavior=False*, *inMonitor=False*, *inCompose=False*, *inSetup=False*, *inInterruptBlock=False*)

Compiles Scenic AST to Python AST

Parameters

- **scenicAST** (AST) –
- **filename** (*str*) –
- **inBehavior** (*bool*) –
- **inMonitor** (*bool*) –
- **inCompose** (*bool*) –
- **inSetup** (*bool*) –
- **inInterruptBlock** (*bool*) –

Return type
Tuple[*Union*[AST, *List*[AST]], *List*[AST]]

class AttributeFinder(*target*)

Bases: NodeVisitor

Utility class for finding all referenced attributes of a given name.

class LocalFinder

Bases: NodeVisitor

Utility class for finding all local variables of a code block.

class Transformer(*filename*)

Bases: NodeTransformer

Subclass of ast.NodeTransformer with a method for raising syntax errors.

class Context(*value*)

Bases: IntFlag

An enumeration.

scenic.syntax.parser

Summary of Module Members

Functions

parse_file	Parse a file.
parse_string	Parse a string.

Classes

Parser	
ScenicParser	
Target	An enumeration.

Member Details

parse_file(*path*, *py_version=None*, *token_stream_factory=None*, *verbose=False*)

Parse a file.

Parameters

- **path** (*str*) –
- **py_version** (*Optional[tuple]*) –
- **token_stream_factory** (*Optional[Callable[[Callable[[], str]], Iterator[TokenInfo]]]*) –
- **verbose** (*bool*) –

Return type
Module

parse_string(*source*, *mode*, *py_version=None*, *token_stream_factory=None*, *verbose=False*, *filename='<unknown>'*)

Parse a string.

Parameters

- **source** (*str*) –
- **mode** (*Union[Literal['eval'], ~typing.Literal['exec']]*) –
- **py_version** (*Optional[tuple]*) –
- **token_stream_factory** (*Optional[Callable[[Callable[[], str]], Iterator[TokenInfo]]]*) –
- **verbose** (*bool*) –
- **filename** (*str*) –

Return type
Any

class Target(*value*)

Bases: Enum

An enumeration.

scenic.syntax.pygment

Pygments lexer and style for Scenic.

These work with the Pygments syntax highlighter. The module actually defines several lexers used for the Scenic documentation; the main *ScenicLexer* and its associated style *ScenicStyle* are exported by `pyproject.toml` as plugins to Pygments. This means that if you have the `scenic` package installed, the Pygments command-line tool and Python API will automatically recognize Scenic files. For example, to highlight a Scenic program as a self-contained HTML or LaTeX file:

```
$ pygmentize -f html -Ofull,style=scenic prog.scenic > out.html
$ pygmentize -f latex -Ofull,style=scenic prog.scenic > out.tex
```

If highlighting multiple pieces of code, remove the `full` option to avoid having the requisite CSS/preamble material duplicated in all your outputs; you can run **`pygmentize -S scenic -f html`** (or **`latex`**) to generate that material separately.

Summary of Module Members

Classes

BetterPythonLexer	Python lexer with better highlighting of function calls, parameters, etc.
PegenLexer	Lexer for Pegen grammars.
PythonSnippetLexer	Variant PythonLexer for code snippets rather than complete programs.
ScenicGrammarLexer	Lexer for the grammar notation used in the Scenic docs.
ScenicLexer	Lexer for Scenic code.
ScenicPropertyLexer	Silly lexer to color property names consistently with the real lexer.
ScenicRequirementLexer	Further variant lexer for requirements at the top level.
ScenicSnippetLexer	Variant ScenicLexer for code snippets rather than complete programs.
ScenicSpecifierLexer	Further variant lexer for specifiers at the top level.
ScenicStyle	A style providing specialized highlighting for the Scenic language.

Member Details

class BetterPythonLexer(**args*, ***kwds*)

Bases: `PythonLexer`

Python lexer with better highlighting of function calls, parameters, etc.

OK, 'better' is a matter of opinion; but it provides more informative tokens. These tokens will not cause errors under any Pygments style, but require the style to be aware of them in order to actually get better highlighting: use the *ScenicStyle* below for best results.

Adapted from the PythonLexer and the MagicPython grammar by MagicStack Inc., available at https://github.com/MagicStack/MagicPython.

class ScenicLexer(**args*, ***kwds*)

Bases: *BetterPythonLexer*

Lexer for Scenic code.

class ScenicSnippetLexer(**args*, ***kwds*)

Bases: *ScenicLexer*

Variant ScenicLexer for code snippets rather than complete programs.

Specifically, this lexer formats syntactic variables of the form "{name}" as "name" italicized.

class PythonSnippetLexer(**args*, ***kwds*)

Bases: *BetterPythonLexer*

Variant PythonLexer for code snippets rather than complete programs.

Specifically, this lexer formats syntactic variables of the form "{name}" as "name" italicized.

class ScenicSpecifierLexer(**args*, ***kwds*)

Bases: *ScenicSnippetLexer*

Further variant lexer for specifiers at the top level.

class ScenicRequirementLexer(**args*, ***kwds*)

Bases: *ScenicSnippetLexer*

Further variant lexer for requirements at the top level.

class ScenicPropertyLexer(**args*, ***kwds*)

Bases: RegexLexer

Silly lexer to color property names consistently with the real lexer.

class ScenicGrammarLexer(**args*, ***kwds*)

Bases: RegexLexer

Lexer for the grammar notation used in the Scenic docs.

class ScenicStyle

Bases: Style

A style providing specialized highlighting for the Scenic language.

The color scheme is a loose hybrid of that used in the Scenic papers and the 'Mariana' color scheme from Sublime Text. The chosen colors all have a contrast ratio of at least 4.5:1 against the background color, per the W3C's Web Content Accessibility Guidelines.

class PegenLexer(**args*, ***kwds*)

Bases: *BetterPythonLexer*

Lexer for Pegen grammars.

scenic.syntax.relations

Extracting relations (for later pruning) from the syntax of requirements.

Summary of Module Members

Functions

inferDistanceRelations	Infer bounds on distances from a requirement.
inferRelationsFrom	Infer relations between objects implied by a requirement.
inferRelativeHeadingRelations	Infer bounds on relative headings from a requirement.

Classes

BoundRelation	Abstract relation bounding something about another object.
DistanceRelation	Relation bounding another object's distance from this one.
RelativeHeadingRelation	Relation bounding another object's relative heading with respect to this one.
RequirementMatcher	

Member Details

inferRelationsFrom(*reqNode*, *namespace*, *ego*, *line*)

Infer relations between objects implied by a requirement.

inferRelativeHeadingRelations(*matcher*, *reqNode*, *ego*, *line*)

Infer bounds on relative headings from a requirement.

inferDistanceRelations(*matcher*, *reqNode*, *ego*, *line*)

Infer bounds on distances from a requirement.

class BoundRelation(*target*, *lower*, *upper*)

Abstract relation bounding something about another object.

class RelativeHeadingRelation(*target*, *lower*, *upper*)

Bases: *BoundRelation*

Relation bounding another object's relative heading with respect to this one.

class DistanceRelation(*target*, *lower*, *upper*)

Bases: *BoundRelation*

Relation bounding another object's distance from this one.

scenic.syntax.translator

Translator turning Scenic programs into Scenario objects.

The top-level interface to Scenic is provided by two functions:

- *scenarioFromString* – compile a string of Scenic code;
- *scenarioFromFile* – compile a Scenic file.

These output a *Scenario* object, from which scenes can be generated. See the documentation for *Scenario* for details.

When imported, this module hooks the Python import system in order to implement the *import* statement. This is only for the compiler's own use: it is not allowed to import a Scenic module from Python, and attempting to do so will fail with a ModuleNotFoundError.

Scenic is compiled in two main steps: translating the code into Python, and executing the resulting Python module to generate a Scenario object encoding the objects, distributions, etc. in the scenario. For details, see the function *compileStream* below.

Summary of Module Members

Functions

astToSource	
compileStream	Compile a stream of Scenic code and execute it in a namespace.
compileTranslatedTree	
constructScenarioFrom	Build a Scenario object from an executed Scenic module.
dump	
executeCodeIn	Execute the final translated Python code in the given namespace.
gatherBehaviorNamespacesFrom	Gather any global namespaces which could be referred to by behaviors.
purgeModulesUnsafeToCache	Uncache loaded modules which should not be kept after compilation.
scenarioFromFile	Compile a Scenic file into a *Scenario*.
scenarioFromString	Compile a string of Scenic code into a *Scenario*.
scenic_path_hook	
storeScenarioStateIn	Post-process an executed Scenic module, extracting state from the veneer.
topLevelNamespace	Creates an environment like that of a Python script being run directly.

Classes

CompileOptions	Internal class for capturing options used when compiling a scenario.
ScenicFileFinder	
ScenicLoader	

Member Details

class CompileOptions(*mode2D=False*, *modelOverride=None*, *paramOverrides=<factory>*, *scenario=None*)

Internal class for capturing options used when compiling a scenario.

Parameters

- **mode2D** (*bool*) –
- **modelOverride** (*Optional[str]*) –
- **paramOverrides** (*dict*) –
- **scenario** (*Optional[str]*) –

mode2D: bool = False

Whether or not the scenario uses 2D compatibility mode.

modelOverride: Optional[str] = None

Overridden world model, if any.

paramOverrides: dict

Overridden global parameters.

scenario: Optional[str] = None

Selected modular scenario, if any.

property hash

Deterministic hash saved in serialized scenes to catch option mismatches.

scenarioFromString(*string*, *params={}*, *model=None*, *scenario=None*, ***, *filename='<string>'*, *mode2D=False*, ***kwargs*)

Compile a string of Scenic code into a *Scenario*.

The optional **filename** is used for error messages. Other arguments are as in *scenarioFromFile*.

scenarioFromFile(*path*, *params={}*, *model=None*, *scenario=None*, ***, *mode2D=False*, ***kwargs*)

Compile a Scenic file into a *Scenario*.

Parameters

- **path** (*str*) – Path to a Scenic file.
- **params** (*dict*) – Global parameters to override, as a dictionary mapping parameter names to their desired values.
- **model** (*str*) – Scenic module to use as world model.
- **scenario** (*str*) – If there are multiple modular scenarios in the file, which one to compile; if not specified, a scenario called 'Main' is used if it exists.

- **mode2D** (*bool*) – Whether to compile this scenario in `2D compatibility mode`.

Returns
A *Scenario* object representing the Scenic scenario.

Note for Scenic developers: this function accepts additional keyword arguments which are intended for internal use and debugging only. See *_scenarioFromStream* for details.

topLevelNamespace(*path=None*)

Creates an environment like that of a Python script being run directly.

Specifically, __name__ is '__main__', __file__ is the path used to invoke the script (not necessarily its absolute path), and the parent directory is added to the path so that 'import blobbo' will import blobbo from that directory if it exists there.

purgeModulesUnsafeToCache(*oldModules*)

Uncache loaded modules which should not be kept after compilation.

Keeping Scenic modules in `sys.modules` after compilation will cause subsequent attempts at compiling the same module to reuse the compiled scenario: this is usually not what is desired, since compilation can depend on external state (in particular overridden global parameters, used e.g. to specify the map for driving domain scenarios).

Parameters
oldModules – List of names of modules loaded before compilation. These will be skipped.

compileStream(*stream*, *namespace*, *compileOptions*, *filename*)

Compile a stream of Scenic code and execute it in a namespace.

The compilation procedure consists of the following main steps:

1. Parse the Scenic code into a Scenic AST using the parser generated by `pegen` from `scenic.gram`.
2. Compile the Scenic AST into a Python AST with the desired semantics. This is done by the compiler, *scenic.syntax.compiler*.
3. Compile and execute the Python AST.
4. Extract the global state (e.g. objects). This is done by the *storeScenarioStateIn* function.

executeCodeIn(*code*, *namespace*)

Execute the final translated Python code in the given namespace.

storeScenarioStateIn(*namespace*, *requirementSyntax*, *astHash*, *options*)

Post-process an executed Scenic module, extracting state from the veneer.

gatherBehaviorNamespacesFrom(*behaviors*)

Gather any global namespaces which could be referred to by behaviors.

We'll need to rebind any sampled values in them at runtime.

constructScenarioFrom(*namespace*, *scenarioName=None*)

Build a Scenario object from an executed Scenic module.

_scenarioFromStream(*stream*, *compileOptions*, *filename*, ***, *scenario=None*, *path=None*, *_cacheImports=False*)

Compile a stream of Scenic code into a *Scenario*.

This method is not meant to be called directly by users of Scenic. Use the top-level functions *scenarioFromFile* and *scenarioFromString* instead.

These functions also accept the following keyword arguments, which are intended for internal use and debugging only. They should be considered unstable and are subject to modification or removal at any time.

Parameters

_cacheImports (*bool*) – Whether to cache any imported Scenic modules. The default behavior is to not do this, so that subsequent attempts to import such modules will cause them to be recompiled. If it is safe to cache Scenic modules across multiple compilations, set this argument to True. Then importing a Scenic module will have the same behavior as importing a Python module. See *purgeModulesUnsafeToCache* for a more detailed discussion of the internals behind this.

scenic.syntax.veneer

Python implementations of Scenic language constructs.

This module is automatically imported by all Scenic programs. In addition to defining the built-in functions, operators, specifiers, etc., it also stores global state such as the list of all created Scenic objects.

Summary of Module Members

Functions

Above	The above *X* by *Y* polymorphic specifier.
Ahead	The ahead of *X* by *Y* polymorphic specifier.
AltitudeFrom	The altitude from <*vector*> to <*vector*> operator.
AltitudeTo	The angle to <*vector*> operator (using the position of ego as the reference).
Always	
AngleFrom	The angle from <*vector*> to <*vector*> operator.
AngleTo	The angle to <*vector*> operator (using the position of ego as the reference).
ApparentHeading	The apparent heading of <*oriented point*> [from <*vector*>] operator.
ApparentlyFacing	The apparently facing <*heading*> [from <*vector*>] specifier.
At	The at <*vector*> specifier.
AtomicProposition	
Back	The back of <*object*> operator.
BackLeft	The back left of <*object*> operator.
BackRight	The back right of <*object*> operator.
Behind	The behind *X* by *Y* polymorphic specifier.
Below	The below *X* by *Y* polymorphic specifier.
Beyond	The beyond *X* by *Y* from *Z* polymorphic specifier.
Bottom	The bottom of <*object*> operator.
BottomBackLeft	The bottom back left of <*object*> operator.
BottomBackRight	The bottom back right of <*object*> operator.
BottomFrontLeft	The bottom front left of <*object*> operator.
BottomFrontRight	The bottom front right of <*object*> operator.
CanSee	The *X* can see *Y* polymorphic operator.
ContainedIn	The contained in <*region*> specifier.

continues on next page

Table 3 – continued from previous page

DistanceFrom	The distance from *X* to *Y* polymorphic operator.
DistancePast	The distance past <*vector*> of <*oriented point*> operator.
Eventually	
Facing	The facing *X* polymorphic specifier.
FacingAwayFrom	The facing away from <*vector*> specifier.
FacingDirectlyAwayFrom	The facing directly away from <*vector*> specifier.
FacingDirectlyToward	The facing directly toward <*vector*> specifier.
FacingToward	The facing toward <*vector*> specifier.
FieldAt	The <*vector field*> at <*vector*> operator.
Follow	The follow <*field*> from <*vector*> for <*number*> operator.
Following	The following *F* from *X* for *D* specifier.
Front	The front of <*object*> operator.
FrontLeft	The front left of <*object*> operator.
FrontRight	The front right of <*object*> operator.
Implies	
In	The in <*region*> specifier.
Left	The left of <*object*> operator.
LeftSpec	The left of *X* by *Y* polymorphic specifier.
Next	
NotVisible	The not visible <*region*> operator.
NotVisibleFrom	The not visible from <*point*> specifier.
NotVisibleFromOp	The <*region*> not visible from <*point*> operator.
NotVisibleSpec	The not visible specifier (equivalent to not visible from *ego*).
OffsetAlong	The *X* offset along *H* by *Y* polymorphic operator.
OffsetAlongSpec	The offset along *X* by *Y* polymorphic specifier.
OffsetBy	The offset by <*vector*> specifier.
On	The on *X* specifier.
PropositionAnd	
PropositionNot	
PropositionOr	
RelativeHeading	The relative heading of <*heading*> [from <*heading*>] operator.
RelativePosition	The relative position of <*vector*> [from <*vector*>] operator.
RelativeTo	The *X* relative to *Y* polymorphic operator.
Right	The right of <*object*> operator.
RightSpec	The right of *X* by *Y* polymorphic specifier.
Top	The top of <*object*> operator.
TopBackLeft	The top back left of <*object*> operator.
TopBackRight	The top back right of <*object*> operator.

continues on next page

Table 3 – continued from previous page

TopFrontLeft	The `top front left of <object>` operator.
TopFrontRight	The `top front right of <object>` operator.
Until	
Visible	The `visible <region>` operator.
VisibleFrom	The `visible from <point>` specifier.
VisibleFromOp	The `<region> visible from <point>` operator.
VisibleSpec	The `visible` specifier (equivalent to `visible from ego`).
With	The `with <property> <value>` specifier.
activate	Activate the veneer when beginning to compile a Scenic module.
alwaysProvidesOrientation	Whether a Region or distribution over Regions always provides an orientation.
beginSimulation	
callWithStarArgs	
deactivate	Deactivate the veneer after compiling a Scenic module.
directionalSpecHelper	
ego	Function implementing loads and stores to the 'ego' pseudo-variable.
endScenario	
endSimulation	
executeInBehavior	
executeInGuard	
executeInRequirement	
executeInScenario	
filter	
finishScenarioSetup	
float	
globalParameters	
in_initial_scenario	
instantiateSimulator	
int	
isActive	Are we in the middle of compiling a Scenic module?

continues on next page

Table 3 – continued from previous page

len	
localPath	Convert a path relative to the calling Scenic file into an absolute path.
makeRequirement	
model	
mutate	Function implementing the mutate statement.
new	
override	
param	Function implementing the param statement.
prepareScenario	
projectVectorHelper	
range	
record	
record_final	
record_initial	
registerDynamicScenarioClass	
registerExternalParameter	Register a parameter whose value is given by an external sampler.
registerInstance	Add a Scenic instance to the global list of created objects.
registerObject	Add a Scenic object to the global list of created objects.
require	Function implementing the require statement.
require_always	Function implementing the 'require always' statement.
require_eventually	Function implementing the 'require eventually' statement.
require_monitor	
resample	The built-in resample function.
round	
simulation	Get the currently-running *Simulation*.
simulationInProgress	
simulator	
startScenario	
str	

continues on next page

Table 3 – continued from previous page

terminate_after	
terminate_simulation_when	Function implementing the 'terminate simulation when' statement.
terminate_when	Function implementing the 'terminate when' statement.
verbosePrint	Built-in function printing a message only in verbose mode.
workspace	Function implementing loads and stores to the 'workspace' pseudo-variable.
wrapStarredValue	

Classes

Modifier	
ParameterTableProxy	

Member Details

ego(*obj=None*)

Function implementing loads and stores to the ‘ego’ pseudo-variable.

The translator calls this with no arguments for loads, and with the source value for stores.

workspace(*workspace=None*)

Function implementing loads and stores to the ‘workspace’ pseudo-variable.

See *ego*.

require(*reqID*, *req*, *line*, *name*, *prob=1*)

Function implementing the require statement.

resample(*dist*)

The built-in resample function.

param(*params*)

Function implementing the param statement.

mutate(**objects*, *scale=1*)

Function implementing the mutate statement.

verbosePrint(**objects*, *level=1*, *indent=True*, *sep=' '*, *end='\n'*, *file=sys.stdout*, *flush=False*)

Built-in function printing a message only in verbose mode.

Scenic’s verbosity may be set using the `-v` command-line option. The simplest way to use this function is with code like `verbosePrint('hello world!')` or `verbosePrint('details here', level=3)`; the other keyword arguments are probably only useful when replacing more complex uses of the Python `print` function.

Parameters

- **objects** – Object(s) to print (`str` will be called to make them strings).

- **level** (*int*) – Minimum verbosity level at which to print. Default is 1.
- **indent** (*bool*) – Whether to indent the message to align with messages generated by Scenic (default true).
- **sep** – As in `print`.
- **end** – As in `print`.
- **file** – As in `print`.
- **flush** – As in `print`.

localPath(*relpath*)

Convert a path relative to the calling Scenic file into an absolute path.

For example, *localPath*(`'resource.dat'`) evaluates to the absolute path of a file called `resource.dat` located in the same directory as the Scenic file where this expression appears. Note that the path is returned as a `pathlib.Path` object.

simulation()

Get the currently-running *Simulation*.

May only be called from code that runs at simulation time, e.g. inside dynamic behaviors and *compose* blocks of scenarios.

terminate_when(*reqID*, *req*, *line*, *name*)

Function implementing the 'terminate when' statement.

terminate_simulation_when(*reqID*, *req*, *line*, *name*)

Function implementing the 'terminate simulation when' statement.

Visible(*region*)

The `visible` <***region***> operator.

NotVisible(*region*)

The `not visible` <***region***> operator.

Front(*X*)

The `front of` <***object***> operator.

Back(*X*)

The `back of` <***object***> operator.

Left(*X*)

The `left of` <***object***> operator.

Right(*X*)

The `right of` <***object***> operator.

FrontLeft(*X*)

The `front left of` <***object***> operator.

FrontRight(*X*)

The `front right of` <***object***> operator.

BackLeft(*X*)

The `back left of` <***object***> operator.

BackRight(*X*)

The `back right of` <***object***> operator.

Top(*X*)

The `top of` <*object*> operator.

Bottom(*X*)

The `bottom of` <*object*> operator.

TopFrontLeft(*X*)

The `top front left of` <*object*> operator.

TopFrontRight(*X*)

The `top front right of` <*object*> operator.

TopBackLeft(*X*)

The `top back left of` <*object*> operator.

TopBackRight(*X*)

The `top back right of` <*object*> operator.

BottomFrontLeft(*X*)

The `bottom front left of` <*object*> operator.

BottomFrontRight(*X*)

The `bottom front right of` <*object*> operator.

BottomBackLeft(*X*)

The `bottom back left of` <*object*> operator.

BottomBackRight(*X*)

The `bottom back right of` <*object*> operator.

RelativeHeading(*X*, *Y=None*)

The `relative heading of` <*heading*> [`from` <*heading*>] operator.

If the `from` <*heading*> is omitted, the heading of ego is used.

ApparentHeading(*X*, *Y=None*)

The `apparent heading of` <*oriented point*> [`from` <*vector*>] operator.

If the `from` <*vector*> is omitted, the position of ego is used.

RelativePosition(*X*, *Y=None*)

The `relative position of` <*vector*> [`from` <*vector*>] operator.

If the `from` <*vector*> is omitted, the position of ego is used.

DistanceFrom(*X*, *Y=None*)

The `distance from` *X* `to` *Y* polymorphic operator.

Allowed forms:

```
distance from <vector> [to <vector>]
distance from <region> [to <vector>]
distance from <vector> to <region>
```

If the `to` <*vector*> is omitted, the position of ego is used.

DistancePast(*X*, *Y=None*)

The `distance past` <*vector*> `of` <*oriented point*> operator.

If the `of {oriented point}` is omitted, the ego object is used.

Follow(*F*, *X*, *D*)

The `follow` <*field*> `from` <*vector*> `for` <*number*> operator.

AngleTo(*X*)

The `angle to` <*vector*> operator (using the position of ego as the reference).

AngleFrom(*X=None*, *Y=None*)

The `angle from` <*vector*> `to` <*vector*> operator.

AltitudeTo(*X*)

The `angle to` <*vector*> operator (using the position of ego as the reference).

AltitudeFrom(*X=None*, *Y=None*)

The `altitude from` <*vector*> `to` <*vector*> operator.

FieldAt(*X*, *Y*)

The <*vector field*> `at` <*vector*> operator.

RelativeTo(*X*, *Y*)

The *X* `relative to` *Y* polymorphic operator.

Allowed forms:

```
<value> relative to <value> # with at least one a field, the other a field or␣
↪heading
<vector> relative to <oriented point> # and vice versa
<vector> relative to <vector>
<heading> relative to <heading>
<orientation> relative to <orientation>
```

Return type
Union[Vector, float, Orientation]

OffsetAlong(*X*, *H*, *Y*)

The *X* `offset along` *H* `by` *Y* polymorphic operator.

Allowed forms:

```
<vector> offset along <heading> by <vector>
<vector> offset along <field> by <vector>
```

CanSee(*X*, *Y*)

The *X* `can see` *Y* polymorphic operator.

Allowed forms:

```
<point> can see <vector>
<point> can see <point>
```

VisibleFromOp(*region*, *base*)

The <*region*> `visible from` <*point*> operator.

NotVisibleFromOp(*region*, *base*)

The <*region*> `not visible from` <*point*> operator.

class Vector(*x*, *y*, *z=0*)

Bases: *Samplable*, Sequence

A 3D vector, whose coordinates can be distributions.

sphericalCoordinates()

Returns this vector in spherical coordinates (rho, theta, phi)

rotatedBy(*angleOrOrientation*)

Return a vector equal to this one rotated counterclockwise by angle/orientation.

Return type
Vector

angleWith(*other*)

Compute the signed angle between self and other.

The angle is positive if other is counterclockwise of self (considering the smallest possible rotation to align them).

Return type
float

class Orientation(*rotation*)

An orientation in 3D space.

classmethod fromQuaternion(*quaternion*)

Create an *Orientation* from a quaternion (of the form (x,y,z,w))

Return type
Orientation

classmethod fromEuler(*yaw*, *pitch*, *roll*)

Create an *Orientation* from yaw, pitch, and roll angles (in radians).

Return type
Orientation

property yaw: float

Yaw in the global coordinate system.

property pitch: float

Pitch in the global coordinate system.

property roll: float

Roll in the global coordinate system.

property eulerAngles: Tuple[float, float, float]

Global intrinsic Euler angles yaw, pitch, roll.

localAnglesFor(*orientation*)

Get local Euler angles for an orientation w.r.t. this orientation.

That is, considering `self` as the parent orientation, find the Euler angles expressing the given orientation.

Return type
Tuple[float, float, float]

globalToLocalAngles(*yaw*, *pitch*, *roll*)

Convert global Euler angles to local angles w.r.t. this orientation.

Equivalent to *localAnglesFor* but takes Euler angles as input.

Return type
Tuple[float, float, float]

class VectorField(*name*, *value*, *minSteps=4*, *defaultStepSize=5*)

A vector field, providing an orientation at every point.

Parameters

- **name** (*str*) – name for debugging.
- **value** – function computing the heading at the given *Vector*.
- **minSteps** (*int*) – Minimum number of steps for *followFrom*; default 4.
- **defaultStepSize** (*float*) – Default step size for *followFrom*; default 5. This is an upper bound: more steps will be taken as needed to ensure that no single step is longer than this value, but if the distance to travel is small then the steps may be smaller.

followFrom(*pos*, *dist*, *steps=None*, *stepSize=None*)

Follow the field from a point for a given distance.

Uses the forward Euler approximation, covering the given distance with equal-size steps. The number of steps can be given manually, or computed automatically from a desired step size.

Parameters

- **pos** (*Vector*) – point to start from.
- **dist** (*float*) – distance to travel.
- **steps** (*int*) – number of steps to take, or None to compute the number of steps based on the distance (default None).
- **stepSize** (*float*) – length used to compute how many steps to take, or None to use the field's default step size.

static forUnionOf(*regions*, *tolerance=0*)

Creates a *PiecewiseVectorField* from the union of the given regions.

If none of the regions have an orientation, returns None instead.

class PolygonalVectorField(*name*, *cells*, *headingFunction=None*, *defaultHeading=None*)

Bases: *VectorField*

A piecewise-constant vector field defined over polygonal cells.

Parameters

- **name** (*str*) – name for debugging.
- **cells** – a sequence of cells, with each cell being a pair consisting of a Shapely geometry and a heading. If the heading is None, we call the given **headingFunction** for points in the cell instead.
- **headingFunction** – function computing the heading for points in cells without specified headings, if any (default None).
- **defaultHeading** – heading for points not contained in any cell (default None, meaning reject such points).

class Shape(*dimensions*, *scale*)

Bases: ABC

An abstract base class for Scenic shapes.

Represents a physical shape in Scenic. Does not encode position or orientation, which are handled by the *Region* class. Does contain dimension information, which is used as a default value by any *Object* with this shape and can be overwritten.

If dimensions and scale are both specified the dimensions are first set by dimensions, and then scaled by scale.

Parameters

- **dimensions** – The raw (before scaling) dimensions of the shape.
- **scale** – Scales all the dimensions of the shape by a multiplicative factor.

property containsCenter

Whether or not this object contains its central point

class MeshShape(*mesh*, *dimensions=None*, *scale=1*, *initial_rotation=None*)

Bases: *Shape*

A Shape subclass defined by a trimesh.base.Trimesh object.

The mesh passed must be a trimesh.base.Trimesh object that represents a well defined volume (i.e. the is_volume property must be true), meaning the mesh must be watertight, have consistent winding and have outward facing normals.

Parameters

- **mesh** – A mesh object.
- **dimensions** – The raw (before scaling) dimensions of the shape. If dimensions and scale are both specified the dimensions are first set by dimensions, and then scaled by scale.
- **scale** – Scales all the dimensions of the shape by a multiplicative factor. If dimensions and scale are both specified the dimensions are first set by dimensions, and then scaled by scale.
- **initial_rotation** – A 3-tuple containing the yaw, pitch, and roll respectively to apply when loading the mesh. Note the initial_rotation must be fixed.

classmethod fromFile(*path*, *filetype=None*, *compressed=None*, *binary=False*, ***kwargs*)

Load a mesh shape from a file, attempting to infer filetype and compression.

For example: "foo.obj.bz2" is assumed to be a compressed .obj file. "foo.obj" is assumed to be an uncompressed .obj file. "foo" is an unknown filetype, so unless a filetype is provided an exception will be raised.

Parameters

- **path** (*str*) – Path to the file to import.
- **filetype** (*str*) – Filetype of file to be imported. This will be inferred if not provided. The filetype must be one compatible with trimesh.load.
- **compressed** (*bool*) – Whether or not this file is compressed (with bz2). This will be inferred if not provided.
- **binary** (*bool*) – Whether or not to open the file as a binary file.
- **kwargs** – Additional arguments to the MeshShape initializer.

class BoxShape(*dimensions=(1, 1, 1)*, *scale=1*, *initial_rotation=None*)

Bases: *MeshShape*

A box shape with all dimensions 1 by default.

class CylinderShape(*dimensions=(1, 1, 1)*, *scale=1*, *initial_rotation=None*, *sections=24*)

Bases: *MeshShape*

A cylinder shape with all dimensions 1 by default.

class ConeShape(*dimensions=(1, 1, 1)*, *scale=1*, *initial_rotation=None*)

Bases: *MeshShape*

A cone shape with all dimensions 1 by default.

class SpheroidShape(*dimensions=(1, 1, 1)*, *scale=1*, *initial_rotation=None*)

Bases: *MeshShape*

A spheroid shape with all dimensions 1 by default.

class MeshVolumeRegion(**args*, ***kwargs*)

Bases: *MeshRegion*

A region representing the volume of a mesh.

The mesh passed must be a `trimesh.base.Trimesh` object that represents a well defined volume (i.e. the `is_volume` property must be true), meaning the mesh must be watertight, have consistent winding and have outward facing normals.

The mesh is first placed so the origin is at the center of the bounding box (unless `centerMesh` is `False`). The mesh is scaled to `dimensions`, translated so the center of the bounding box of the mesh is at `positon`, and then rotated to `rotation`.

Meshes are centered by default (since `centerMesh` is true by default). If you disable this operation, do note that scaling and rotation transformations may not behave as expected, since they are performed around the origin.

Parameters

- **mesh** – The base mesh for this region.
- **name** – An optional name to help with debugging.
- **dimensions** – An optional 3-tuple, with the values representing width, length, height respectively. The mesh will be scaled such that the bounding box for the mesh has these dimensions.
- **position** – An optional position, which determines where the center of the region will be.
- **rotation** – An optional Orientation object which determines the rotation of the object in space.
- **orientation** – An optional vector field describing the preferred orientation at every point in the region.
- **tolerance** – Tolerance for internal computations.
- **centerMesh** – Whether or not to center the mesh after copying and before transformations.
- **onDirection** – The direction to use if an object being placed on this region doesn't specify one.
- **engine** – Which engine to use for mesh operations. Either "blender" or "scad".

intersects(*other*, *triedReversed=False*)

Check if this region intersects another.

This function handles intersect calculations for *MeshVolumeRegion* with: * *MeshVolumeRegion* * *MeshSurfaceRegion* * *PolygonalFootprintRegion*

containsPoint(*point*)

Check if this region's volume contains a point.

containsObject(*obj*)

Check if this region's volume contains an *Object*.

intersect(*other*, *triedReversed=False*)

Get a *Region* representing the intersection of this region with another.

This function handles intersection computation for *MeshVolumeRegion* with: * *MeshVolumeRegion* * *PolygonalFootprintRegion* * *PolygonalRegion* * *PathRegion* * *PolylineRegion*

union(*other*, *triedReversed=False*)

Get a *Region* representing the union of this region with another.

This function handles union computation for *MeshVolumeRegion* with:

- *MeshVolumeRegion*

difference(*other*, *debug=False*)

Get a *Region* representing the difference of this region with another.

This function handles union computation for *MeshVolumeRegion* with: * *MeshVolumeRegion* * *PolygonalFootprintRegion*

distanceTo(*point*)

Get the minimum distance from this region to the specified point.

getSurfaceRegion()

Return a region equivalent to this one, except as a MeshSurfaceRegion

getVolumeRegion()

Returns this object, as it is already a MeshVolumeRegion

class MeshSurfaceRegion(**args*, ***kwargs*)

Bases: *MeshRegion*

A region representing the surface of a mesh.

The mesh is first placed so the origin is at the center of the bounding box (unless `centerMesh` is `False`). The mesh is scaled to `dimensions`, translated so the center of the bounding box of the mesh is at `positon`, and then rotated to `rotation`.

Meshes are centered by default (since `centerMesh` is true by default). If you disable this operation, do note that scaling and rotation transformations may not behave as expected, since they are performed around the origin.

If an orientation is not passed to this mesh, a default orientation is provided which provides an orientation that aligns an object's z axis with the normal vector of the face containing that point, and has a yaw aligned with a yaw of 0 in the global coordinate system.

Parameters

- **mesh** – The base mesh for this region.
- **name** – An optional name to help with debugging.

- **dimensions** – An optional 3-tuple, with the values representing width, length, height respectively. The mesh will be scaled such that the bounding box for the mesh has these dimensions.
- **position** – An optional position, which determines where the center of the region will be.
- **rotation** – An optional Orientation object which determines the rotation of the object in space.
- **orientation** – An optional vector field describing the preferred orientation at every point in the region.
- **tolerance** – Tolerance for internal computations.
- **centerMesh** – Whether or not to center the mesh after copying and before transformations.
- **onDirection** – The direction to use if an object being placed on this region doesn't specify one.

intersects(*other*, *triedReversed=False*)

Check if this region's surface intersects another.

This function handles intersection computation for *MeshSurfaceRegion* with: * *MeshSurfaceRegion* * *PolygonalFootprintRegion*

containsPoint(*point*)

Check if this region's surface contains a point.

distanceTo(*point*)

Get the minimum distance from this object to the specified point.

getFlatOrientation(*pos*)

Get a flat orientation at a point in the region.

Given a point on the surface of the mesh, returns an orientation that aligns an instance's z axis with the normal vector of the face containing that point. Since there are infinitely many such orientations, the orientation returned has yaw aligned with a global yaw of 0.

If `pos` is not within `self.tolerance` of the surface of the mesh, a `RejectionException` is raised.

getVolumeRegion()

Return a region equivalent to this one, except as a MeshVolumeRegion

getSurfaceRegion()

Returns this object, as it is already a MeshSurfaceRegion

class BoxRegion(**args*, ***kwargs*)

Bases: *MeshVolumeRegion*

Region in the shape of a rectangular cuboid, i.e. a box.

By default the unit box centered at the origin and aligned with the axes is used.

Parameters are the same as *MeshVolumeRegion*, with the exception of the `mesh` parameter which is excluded.

class SpheroidRegion(**args*, ***kwargs*)

Bases: *MeshVolumeRegion*

Region in the shape of a spheroid.

By default the unit sphere centered at the origin and aligned with the axes is used.

Parameters are the same as *MeshVolumeRegion*, with the exception of the `mesh` parameter which is excluded.

class PathRegion(*points=None*, *polylines=None*, *tolerance=1e-08*, *name=None*)

Bases: *Region*

A region composed of multiple polylines in 3D space.

One of points or polylines should be provided.

Parameters

- **points** – A list of points defining a single polyline.
- **polylines** – A list of list of points, defining multiple polylines.
- **tolerance** – Tolerance used internally.

class Region(*name*, **dependencies*, *orientation=None*)

Bases: *Samplable*, ABC

An abstract base class for Scenic Regions

abstract uniformPointInner()

Do the actual random sampling. Implemented by subclasses.

abstract containsPoint(*point*)

Check if the *Region* contains a point. Implemented by subclasses.

Return type
bool

abstract containsObject(*obj*)

Check if the *Region* contains an *Object*

Return type
bool

abstract containsRegionInner(*reg*, *tolerance*)

Check if the *Region* contains a *Region*

Return type
bool

abstract distanceTo(*point*)

Distance to this region from a given point.

Return type
float

abstract projectVector(*point*, *onDirection*)

Returns point projected onto this region along onDirection.

abstract property AABB

Axis-aligned bounding box for this *Region*.

intersects(*other*)

Check if this *Region* intersects another.

Return type
bool

intersect(*other*, *triedReversed=False*)

Get a *Region* representing the intersection of this one with another.

If both regions have a preferred orientation, the one of `self` is inherited by the intersection.

Return type
Region

union(*other*, *triedReversed=False*)

Get a *Region* representing the union of this one with another.

Not supported by all region types.

Return type
Region

difference(*other*)

Get a *Region* representing the difference of this one and another.

Not supported by all region types.

Return type
Region

static uniformPointIn(*region*)

Get a uniform *Distribution* over points in a *Region*.

orient(*vec*)

Orient the given vector along the region's orientation, if any.

class PointSetRegion(*name*, *points*, *kdTree=None*, *orientation=None*, *tolerance=1e-06*)

Bases: *Region*

Region consisting of a set of discrete points.

No *Object* can be contained in a *PointSetRegion*, since the latter is discrete. (This may not be true for subclasses, e.g. *GridRegion*.)

Parameters

- **name** (*str*) – name for debugging
- **points** (*arraylike*) – set of points comprising the region
- **kdTree** (scipy.spatial.KDTree, optional) – k-D tree for the points (one will be computed if none is provided)
- **orientation** (*VectorField*; optional) – preferred orientation for the region
- **tolerance** (*float; optional*) – distance tolerance for checking whether a point lies in the region

class RectangularRegion(*position*, *heading*, *width*, *length*, *name=None*)

Bases: *PolygonalRegion*

A rectangular region with a possibly-random position, heading, and size.

Parameters

- **position** (*Vector*) – center of the rectangle.
- **heading** (*float*) – the heading of the length axis of the rectangle.
- **width** (*float*) – width of the rectangle.
- **length** (*float*) – length of the rectangle.
- **name** (*str; optional*) – name for debugging.

class CircularRegion(*center*, *radius*, *resolution=32*, *name=None*)

Bases: *PolygonalRegion*

A circular region with a possibly-random center and radius.

Parameters

- **center** (*Vector*) – center of the disc.
- **radius** (*float*) – radius of the disc.
- **resolution** (*int; optional*) – number of vertices to use when approximating this region as a polygon.
- **name** (*str; optional*) – name for debugging.

class SectorRegion(*center*, *radius*, *heading*, *angle*, *resolution=32*, *name=None*)

Bases: *PolygonalRegion*

A sector of a *CircularRegion*.

This region consists of a sector of a disc, i.e. the part of a disc subtended by a given arc.

Parameters

- **center** (*Vector*) – center of the corresponding disc.
- **radius** (*float*) – radius of the disc.
- **heading** (*float*) – heading of the centerline of the sector.
- **angle** (*float*) – angle subtended by the sector.
- **resolution** (*int; optional*) – number of vertices to use when approximating this region as a polygon.
- **name** (*str; optional*) – name for debugging.

class PolygonalRegion(*points=None*, *polygon=None*, *z=0*, *orientation=None*, *name=None*, *additionalDeps=[]*)

Bases: *Region*

Region given by one or more polygons (possibly with holes) at a fixed z coordinate.

The region may be specified by giving either a sequence of points defining the boundary of the polygon, or a collection of `shapely` polygons (a `Polygon` or `MultiPolygon`).

Parameters

- **points** – sequence of points making up the boundary of the polygon (or `None` if using the **polygon** argument instead).
- **polygon** – `shapely` polygon or collection of polygons (or `None` if using the **points** argument instead).
- **z** – The z coordinate the polygon is located at.
- **orientation** (*VectorField*; optional) – preferred orientation to use.
- **name** (*str; optional*) – name for debugging.

property boundary: *PolylineRegion*

Get the boundary of this region as a *PolylineRegion*.

class PolylineRegion(*points=None*, *polyline=None*, *orientation=True*, *name=None*)

Bases: *Region*

Region given by one or more polylines (chain of line segments).

The region may be specified by giving either a sequence of points or `shapely` polylines (a `LineString` or `MultiLineString`).

Parameters

- **points** – sequence of points making up the polyline (or `None` if using the **polyline** argument instead).
- **polyline** – `shapely` polyline or collection of polylines (or `None` if using the **points** argument instead).
- **orientation** (*optional*) – preferred orientation to use, or `True` to use an orientation aligned with the direction of the polyline (the default).
- **name** (*str; optional*) – name for debugging.

property start

Get an *OrientedPoint* at the start of the polyline.

The OP's orientation will be aligned with the orientation of the region, if there is one (the default orientation pointing along the polyline).

property end

Get an *OrientedPoint* at the end of the polyline.

The OP's orientation will be aligned with the orientation of the region, if there is one (the default orientation pointing along the polyline).

signedDistanceTo(*point*)

Compute the signed distance of the PolylineRegion to a point.

The distance is positive if the point is left of the nearest segment, and negative otherwise.

Return type
float

pointAlongBy(*distance*, *normalized=False*)

Find the point a given distance along the polyline from its start.

If **normalized** is true, then distance should be between 0 and 1, and is interpreted as a fraction of the length of the polyline. So for example `pointAlongBy(0.5, normalized=True)` returns the polyline's midpoint.

Return type
Vector

class Workspace(*region=<AllRegion everywhere>*)

Bases: *Region*

A workspace describing the fixed world of a scenario.

Parameters
region (Region) – The region defining the extent of the workspace (default *everywhere*).

show3D(*viewer*)

Render a schematic of the workspace (in 3D) for debugging

show2D(*plt*)

Render a schematic of the workspace (in 2D) for debugging

zoomAround(*plt*, *objects*, *expansion=1*)

Zoom the schematic around the specified objects

scenicToSchematicCoords(*coords*)

Convert Scenic coordinates to those used for schematic rendering.

class Mutator

An object controlling how the *mutate* statement affects an *Object*.

A *Mutator* can be assigned to the `mutator` property of an *Object* to control the effect of the *mutate* statement. When mutation is enabled for such an object using that statement, the mutator's *appliedTo* method is called to compute a mutated version. The *appliedTo* method can also decide whether to apply mutators inherited from superclasses.

appliedTo(*obj*)

Return a mutated copy of the given object. Implemented by subclasses.

The mutator may inspect the `mutationScale` attribute of the given object to scale its effect according to the scale given in `mutate O by S`.

Returns

A pair consisting of the mutated copy of the object (which is most easily created using *_copyWith*) together with a Boolean indicating whether the mutator inherited from the superclass (if any) should also be applied.

class Range(*low*, *high*)

Bases: *Distribution*

Uniform distribution over a range

class DiscreteRange(*low*, *high*, *weights=None*, *emptyMessage='empty DiscreteRange'*)

Bases: *Distribution*

Distribution over a range of integers.

class Options(*opts*)

Bases: *MultiplexerDistribution*

Distribution over a finite list of options.

Specified by a dict giving probabilities; otherwise uniform over a given iterable.

Uniform(**opts*)

Uniform distribution over a finite list of options.

Implemented as an instance of *Options* when the set of options is known statically, and an instance of *UniformDistribution* otherwise.

Discrete

alias of *Options*

class Normal(*mean*, *stddev*)

Bases: *Distribution*

Normal distribution

class TruncatedNormal(*mean*, *stddev*, *low*, *high*)

Bases: *Normal*

Truncated normal distribution.

class VerifaiParameter(*domain*)

Bases: *ExternalParameter*

An external parameter sampled using one of VerifAI's samplers.

static withPrior(*dist*, *buckets=None*)

Creates a *VerifaiParameter* using the given distribution as a prior.

Since the VerifAI cross-entropy sampler currently only supports piecewise-constant distributions, if the prior is not of that form it may be approximated. For most built-in distributions, the approximation is exact: for a particular distribution, check its *bucket* method.

class VerifaiRange(*low*, *high*, *buckets=None*, *weights=None*)

Bases: *VerifaiParameter*

A *Range* (real interval) sampled by VerifAI.

_defaultValueType

alias of `float`

class VerifaiDiscreteRange(*low*, *high*, *weights=None*)

Bases: *VerifaiParameter*

A *DiscreteRange* (integer interval) sampled by VerifAI.

_defaultValueType

alias of `float`

class VerifaiOptions(*opts*)

Bases: *Options*

An *Options* (discrete set) sampled by VerifAI.

class Point <specifiers>

Bases: *Constructible*

The Scenic base class `Point`.

The default mutator for *Point* adds Gaussian noise to `position` with a standard deviation given by the `positionStdDev` property.

Properties

- **position** (*Vector*; dynamic) – Position of the point. Default value is the origin (0,0,0).
- **width** (*float*) – Default value 0 (only provided for compatibility with operators that expect an *Object*).
- **length** (*float*) – Default value 0.
- **height** (*float*) – Default value 0.
- **baseOffset** (*Vector*) – Only provided for compatibility with the *on region* specifier. Default value is (0,0,0).
- **contactTolerance** (*float*) – Only provided for compatibility with the specifiers that expect an *Object*. Default value 0.

- **onDirection** (*Vector*) – The direction used to determine where to place this *Point* on a region, when using the modifying *on* specifier. See the *on region* page for more details. Default value is None, indicating the direction will be inferred from the region this object is being placed on.
- **visibleDistance** (*float*) – Distance used to determine the visible range of this object. Default value 50.
- **viewRayDensity** (*float*) – By default determines the number of rays used during visibility checks. This value is the density of rays per degree of visible range in one dimension. The total number of rays sent will be this value squared per square degree of this object's view angles. This value determines the default value for `viewRayCount`, so if `viewRayCount` is overwritten this value is ignored. Default value 5.
- **viewRayCount** (*None | tuple[float, float]*) – The total number of horizontal and vertical view angles to be sent, or None if this value should be computed automatically. Default value `None`.
- **viewRayDistanceScaling** (*bool*) – Whether or not the number of rays should scale with the distance to the object. Ignored if `viewRayCount` is passed. Default value `False`.
- **mutationScale** (*float*) – Overall scale of mutations, as set by the *mutate* statement. Default value 0 (mutations disabled).
- **positionStdDev** (*tuple[float, float, float]*) – Standard deviation of Gaussian noise for each dimension (x,y,z) to be added to this object's position when mutation is enabled with scale 1. Default value (1,1,0), mutating only the x,y values of the point.

property visibleRegion

The visible region of this object.

The visible region of a *Point* is a sphere centered at its `position` with radius `visibleDistance`.

canSee(*other*, *occludingObjects=()*, *debug=False*)

Whether or not this *Point* can see `other`.

Parameters

- **other** – A *Point*, *OrientedPoint*, or *Object* to check for visibility.
- **occludingObjects** – A list of objects that can occlude visibility.

Return type
bool

class OrientedPoint <specifiers>

Bases: *Point*

The Scenic class `OrientedPoint`.

The default mutator for *OrientedPoint* adds Gaussian noise to `yaw` while leaving `pitch` and `roll` unchanged, using the three standard deviations (for yaw/pitch/roll respectively) given by the `orientationStdDev` property. It then also applies the mutator for *Point*.

The default mutator for *OrientedPoint* adds Gaussian noise to `yaw`, `pitch` and `roll` according to `orientationStdDev`. By default the standard deviations for `pitch` and `roll` are zero so that, by default, only `yaw` is mutated.

Properties

- **yaw** (*float; dynamic*) – Yaw of the *OrientedPoint* in radians in the local coordinate system provided by `parentOrientation`. Default value 0.

- **pitch** (*float; dynamic*) – Pitch of the *OrientedPoint* in radians in the local coordinate system provided by parentOrientation. Default value 0.
- **roll** (*float; dynamic*) – Roll of the *OrientedPoint* in radians in the local coordinate system provided by parentOrientation. Default value 0.
- **parentOrientation** (*Orientation*) – The local coordinate system that the *OrientedPoint*'s yaw, pitch, and roll are interpreted in. Default value is the global coordinate system, where an object is flat in the XY plane, facing North.
- **orientation** (*Orientation*; dynamic; final) – The orientation of the *OrientedPoint* relative to the global coordinate system. Derived from the yaw, pitch, roll, and parentOrientation of this *OrientedPoint* and non-overridable.
- **heading** (*float; dynamic; final*) – Yaw value of this *OrientedPoint* in the global coordinate system. Derived from orientation and non-overridable.
- **viewAngles** (*tuple[float,float]*) – Horizontal and vertical view angles of this *OrientedPoint* in radians. Horizontal view angle can be up to 2 and vertical view angle can be up to . Values greater than these will be truncated. Default value is (2,)
- **orientationStdDev** (*tuple[float,float,float]*) – Standard deviation of Gaussian noise to add to this object's Euler angles (yaw, pitch, roll) when mutation is enabled with scale 1. Default value (5°, 0, 0), mutating only the yaw of this *OrientedPoint*.

property visibleRegion

The visible region of this object.

The visible region of an *OrientedPoint* restricts that of *Point* (a sphere with radius visibleDistance) based on the value of viewAngles. In general, it is a capped rectangular pyramid subtending an angle of viewAngles[0] horizontally and viewAngles[1] vertically, as long as those angles are less than /2; larger angles yield various kinds of wrap-around regions. See *ViewRegion* for details.

canSee(*other*, *occludingObjects=()*, *debug=False*)

Whether or not this *OrientedPoint* can see other.

Parameters

- **other** – A *Point*, *OrientedPoint*, or *Object* to check for visibility.
- **occludingObjects** – A list of objects that can occlude visibility.

Return type
bool

distancePast(*vec*)

Distance past a given point, assuming we've been moving in a straight line.

class Object <specifiers>

Bases: *OrientedPoint*

The Scenic class Object.

This is the default base class for Scenic classes.

Properties

- **width** (*float*) – Width of the object, i.e. extent along its X axis. Default value of 1 inherited from the object's shape.
- **length** (*float*) – Length of the object, i.e. extent along its Y axis. Default value of 1 inherited from the object's shape.

- **height** (*float*) – Height of the object, i.e. extent along its Z axis. Default value of 1 inherited from the object's `shape`.
- **shape** (*Shape*) – The shape of the object, which must be an instance of *Shape*. The default shape is a box, with default unit dimensions.
- **allowCollisions** (*bool*) – Whether the object is allowed to intersect other objects. Default value `False`.
- **regionContainedIn** (*Region* or `None`) – A *Region* the object is required to be contained in. If `None`, the object need only be contained in the scenario's workspace.
- **baseOffset** (*Vector*) – An offset from the `position` of the Object to the base of the object, used by the *on region* specifier. Default value `(0, 0, -self.height/2)`, placing the base of the Object at the bottom center of the Object's bounding box.
- **contactTolerance** (*float*) – The maximum distance this object can be away from a surface to be considered on the surface. Objects are placed at half this distance away from a point when the *on region* specifier or a directional specifier like *(left | right) of Object [by scalar]* is used. Default value 1e-4.
- **sideComponentThresholds** (*DimensionLimits*) – Used to determine the various sides of an object (when using the default implementation). The three interior 2-tuples represent the maximum and minimum bounds for each dimension's (x,y,z) surface. See *defaultSideSurface* for details. Default value `((-0.5, 0.5), (-0.5, 0.5), (-0.5, 0.5))`.
- **cameraOffset** (*Vector*) – Position of the camera for the *can see* operator, relative to the object's `position`. Default `(0, 0, 0)`.
- **requireVisible** (*bool*) – Whether the object is required to be visible from the `ego` object. Default value `False`.
- **occluding** (*bool*) – Whether or not this object can occlude other objects. Default value `True`.
- **showVisibleRegion** (*bool*) – Whether or not to display the visible region in the Scenic internal visualizer.
- **color** (tuple[float, float, float, float] or tuple[float, float, float] or `None`) – An optional color (with optional alpha) property that is used by the internal visualizer, or possibly simulators. All values should be between 0 and 1. Default value `None`
- **velocity** (*Vector*; *dynamic*) – Velocity in dynamic simulations. Default value is the velocity determined by `speed` and `orientation`.
- **speed** (*float; dynamic*) – Speed in dynamic simulations. Default value 0.
- **angularVelocity** (*Vector*; *dynamic*)
- **angularSpeed** (*float; dynamic*) – Angular speed in dynamic simulations. Default value 0.
- **behavior** – Behavior for dynamic agents, if any (see *Dynamic Scenarios*). Default value `None`.
- **lastActions** – Tuple of actions taken by this agent in the last time step (or `None` if the object is not an agent or this is the first time step).

startDynamicSimulation()

Hook called when the object is created in a dynamic simulation.

Does nothing by default; provided for objects to do simulator-specific initialization as needed.

Changed in version 3.0: This method is called on objects created in the middle of dynamic simulations, not only objects present in the initial scene.

containsPoint(*point*)

Whether or not the space this object occupies contains a point

distanceTo(*point*)

The minimal distance from the space this object occupies to a given point

intersects(*other*)

Whether or not this object intersects another object

property visibleRegion

The visible region of this object.

The visible region of an *Object* is the same as that of an *OrientedPoint* (see *OrientedPoint.visibleRegion*) except that it is offset by the value of cameraOffset (which is the zero vector by default).

canSee(*other*, *occludingObjects=()*, *debug=False*)

Whether or not this *Object* can see other.

Parameters

- **other** – A *Point*, *OrientedPoint*, or *Object* to check for visibility.
- **occludingObjects** – A list of objects that can occlude visibility.

Return type
bool

property corners

A tuple containing the corners of this object's bounding box

property occupiedSpace

A region representing the space this object occupies

property _isConvex

Whether this object's shape is convex

property boundingBox

A region representing this object's bounding box

property inradius

A lower bound on the inradius of this object

property surface

A region containing the entire surface of this object

property onSurface

The surface used by the on specifier.

This region is used to sample position when another object is placed on this object. By default the top surface of this object (*topSurface*), but can be overwritten by subclasses.

property topSurface

A region containing the top surface of this object

For how this surface is computed, see *defaultSideSurface*.

property rightSurface

A region containing the right surface of this object

For how this surface is computed, see *defaultSideSurface*.

property leftSurface

A region containing the left surface of this object

For how this surface is computed, see *defaultSideSurface*.

property frontSurface

A region containing the front surface of this object

For how this surface is computed, see *defaultSideSurface*.

property backSurface

A region containing the back surface of this object

For how this surface is computed, see *defaultSideSurface*.

property bottomSurface

A region containing the bottom surface of this object

For how this surface is computed, see *defaultSideSurface*.

property _isPlanarBox

Whether this object is a box aligned with the XY plane.

With(*prop*, *val*)

The `with` <***property***> <***value***> specifier.

Specifies the given property, with no dependencies.

At(*pos*)

The `at` <***vector***> specifier.

Specifies `position`, with no dependencies.

In(*region*)

The `in` <***region***> specifier.

Specifies `position`, and optionally, `parentOrientation` if the given region has a preferred orientation, with no dependencies.

ContainedIn(*region*)

The `contained in` <***region***> specifier.

Specifies `position`, `regionContainedIn`, and optionally, `parentOrientation` if the given region has a preferred orientation, with no dependencies.

On(*thing*)

The `on` ***X*** specifier.

Specifies `position`, and optionally, `parentOrientation` if the given region has a preferred orientation. Depends on `onDirection`, `baseOffset`, and `contactTolerance`.

Note that while `on` can be used with *Region*, *Object* and *Vector*, it cannot be used with a distribution containing anything other than *Region*.

May be used to modify an already-specified `position` property.

Allowed forms:

on <region> on <object> on <vector>

Beyond(*pos*, *offset*, *fromPt=None*)

The `beyond` ***X*** `by` ***Y*** `from` ***Z*** polymorphic specifier.

Specifies `position`, and optionally `parentOrientation`, with no dependencies.

Allowed forms:

```
beyond <vector> by <number> [from <vector>]
beyond <vector> by <vector> [from <vector>]
```

If the `from` <***vector***> is omitted, the position of ego is used.

VisibleFrom(*base*)

The `visible from` <***point***> specifier.

Specifies `_observingEntity` and `position`, with no dependencies.

NotVisibleFrom(*base*)

The `not visible from` <***point***> specifier.

Specifies `_nonObservingEntity` and `position`, depending on `regionContainedIn`.

See *VisibleFrom*.

VisibleSpec()

The `visible` specifier (equivalent to `visible from` *ego*).

Specifies `_observingEntity` and `position`, with no dependencies.

NotVisibleSpec()

The `not visible` specifier (equivalent to `not visible from` *ego*).

Specifies `_nonObservingEntity` and `position`, depending on `regionContainedIn`.

OffsetBy(*offset*)

The `offset by` <***vector***> specifier.

Specifies `position`, and optionally `parentOrientation`, with no dependencies.

OffsetAlongSpec(*direction*, *offset*)

The `offset along` ***X*** `by` ***Y*** polymorphic specifier.

Specifies `position`, and optionally `parentOrientation`, with no dependencies.

Allowed forms:

```
offset along <heading> by <vector>
offset along <field> by <vector>
```

Facing(*heading*)

The `facing` ***X*** polymorphic specifier.

Specifies `yaw`, `pitch`, and `roll`, depending on `parentOrientation`, and depending on the form:

```
facing <number>     # no further dependencies;
facing <field>      # depends on 'position'
```

ApparentlyFacing(*heading*, *fromPt=None*)

The `apparently facing` <***heading***> `[from` <***vector***>`]` specifier.

Specifies `yaw`, depending on `position` and `parentOrientation`.

If the from <vector> is omitted, the position of ego is used.

FacingToward(*pos*)

The facing toward <vector> specifier.

Specifies yaw, depending on position and parentOrientation.

FacingDirectlyToward(*pos*)

The facing directly toward <vector> specifier.

Specifies yaw and pitch, depends on position and parentOrientation.

FacingAwayFrom(*pos*)

The facing away from <vector> specifier.

Specifies yaw, depending on position and parentOrientation.

FacingDirectlyAwayFrom(*pos*)

The facing directly away from <vector> specifier.

Specifies yaw and pitch, depending on position and parentOrientation.

LeftSpec(*pos*, *dist=None*)

The left of *X* by *Y* polymorphic specifier.

Specifies position, and optionally, parentOrientation, depending on width.

Allowed forms:

```
left of <oriented point> [by <scalar/vector>]
left of <vector> [by <scalar/vector>]
```

If the by <scalar/vector> is omitted, the object's contact tolerance is used.

RightSpec(*pos*, *dist=None*)

The right of *X* by *Y* polymorphic specifier.

Specifies position, and optionally parentOrientation, depending on width.

Allowed forms:

```
right of <oriented point> [by <scalar/vector>]
right of <vector> [by <scalar/vector>]
```

If the by <scalar/vector> is omitted, zero is used.

Ahead(*pos*, *dist=None*)

The ahead of *X* by *Y* polymorphic specifier.

Specifies position, and optionally parentOrientation, depending on length.

Allowed forms:

```
ahead of <oriented point> [by <scalar/vector>]
ahead of <vector> [by <scalar/vector>]
```

If the by <scalar/vector> is omitted, the object's contact tolerance is used.

Behind(*pos*, *dist=None*)

The `behind X by Y` polymorphic specifier.

Specifies `position`, and optionally `parentOrientation`, depending on `length`.

Allowed forms:

```
behind <oriented point> [by <scalar/vector>]
behind <vector> [by <scalar/vector>]
```

If the `by <scalar/vector>` is omitted, the object's contact tolerance is used.

Above(*pos*, *dist=None*)

The `above X by Y` polymorphic specifier.

Specifies `position`, and optionally `parentOrientation`, depending on `height`.

Allowed forms:

```
above <oriented point> [by <scalar/vector>]
above <vector> [by <scalar/vector>]
```

If the `by <scalar/vector>` is omitted, the object's contact tolerance is used.

Below(*pos*, *dist=None*)

The `below X by Y` polymorphic specifier.

Specifies :prop`position`, and optionally `parentOrientation`, depending on `height`.

Allowed forms:

```
below <oriented point> [by <scalar/vector>]
below <vector> [by <scalar/vector>]
```

If the `by <scalar/vector>` is omitted, the object's contact tolerance is used.

Following(*field*, *dist*, *fromPt=None*)

The `following F from X for D` specifier.

Specifies `position`, and optionally `parentOrientation`, with no dependencies.

Allowed forms:

```
following <field> [from <vector>] for <number>
```

If the `from <vector>` is omitted, the position of ego is used.

exception GuardViolation(*behavior*, *lineno*)

Bases: `Exception`

Abstract exception raised when a guard of a behavior is violated.

This will never be raised directly; either of the subclasses *PreconditionViolation* or *InvariantViolation* will be used, as appropriate.

exception PreconditionViolation(*behavior*, *lineno*)

Bases: *GuardViolation*

Exception raised when a precondition is violated

Raised when a precondition is violated when invoking a behavior or when a precondition encounters a *RejectionException*, so that rejections count as precondition violations.

exception InvariantViolation(*behavior*, *lineno*)

Bases: *GuardViolation*

Exception raised when an invariant is violated

Raised when an invariant is violated when invoking/resuming a behavior or when an invariant encounters a *RejectionException*, so that rejections count as invariant violations.

exception RejectionException

Bases: Exception

Exception used to signal that the sample currently being generated must be rejected.

_scenic_default

alias of *PropertyDefault*

class Behavior(**args*, ***kwargs*)

Bases: *Invocable*, *Samplable*

Dynamic behaviors of agents.

Behavior statements are translated into definitions of subclasses of this class.

class Monitor(**args*, ***kwargs*)

Bases: *Behavior*

Monitors for dynamic simulations.

Monitor statements are translated into definitions of subclasses of this class.

class BlockConclusion(*value*)

Bases: Enum

An enumeration.

class Modifier(*name*, *value*, *terminator*)

Bases: NamedTuple

Parameters

- **name** (*str*) –
- **value** (*Any*) –
- **terminator** (*Optional[str]*) –

name: str

Alias for field number 0

value: Any

Alias for field number 1

terminator: Optional[str]

Alias for field number 2

_asdict()

Return a new dict which maps field names to their values.

classmethod _make(*iterable*)

Make a new Modifier object from a sequence or iterable

_replace(***kwds*)

Return a new Modifier object replacing specified fields with new values

class DynamicScenario(**args*, ***kwargs*)

Bases: *Invocable*

Internal class for scenarios which can execute during dynamic simulations.

Provides additional information complementing *Scenario*, which originally only supported static scenarios. The two classes should probably eventually be merged.

classmethod _requiresArguments()

Whether this scenario cannot be instantiated without arguments.

_bindTo(*scene*)

Bind this scenario to a sampled scene when starting a new simulation.

_prepare(*delayPreconditionCheck=False*)

Prepare the scenario for execution, executing its setup block.

_start()

Start the scenario, starting its compose block, behaviors, and monitors.

_step()

Execute the (already-started) scenario for one time step.

Returns

None if the scenario will continue executing; otherwise a string describing why it has terminated.

_stop(*reason*, *quiet=False*)

Stop the scenario's execution, for the given reason.

_addRequirement(*ty*, *reqID*, *req*, *line*, *name*, *prob*)

Save a requirement defined at compile-time for later processing.

_addDynamicRequirement(*ty*, *req*, *line*, *name*)

Add a requirement defined during a dynamic simulation.

_addMonitor(*monitor*)

Add a monitor during a dynamic simulation.

Summary of Module Members

Functions

buildParser

Member Details

The `scenic` module itself provides the top-level API for using Scenic: see *Using Scenic Programmatically*.

1.13 Scenic Libraries

One of the strengths of Scenic is its ability to reuse functions, classes, and behaviors across many scenarios, simplifying the process of writing complex scenarios. This page describes the libraries built into Scenic to facilitate scenario writing by end users.

1.13.1 Simulator Interfaces

Many of the simulator interfaces provide utility functions which are useful when writing scenarios for particular simulators. See the documentation for each simulator on the *Supported Simulators* page, as well as the corresponding module under `scenic.simulators`.

1.13.2 Abstract Domains

To enable cross-platform scenarios which are not specific to one simulator, Scenic defines *abstract domains* which provide APIs for particular application domains like driving scenarios. An abstract domain defines a protocol which can be implemented by various simulator interfaces so that scenarios written for that domain can be executed in those simulators. For example, a scenario written for our *driving domain* can be run in both LGSVL and CARLA.

A domain provides a Scenic world model which defines Scenic classes for the various types of objects that occur in its scenarios. The model also provides a simulator-agnostic way to access the geometry of the simulated world, by defining regions, vector fields, and other objects as appropriate (for example, the driving domain provides a `Network` class abstracting a road network). For domains which support dynamic scenarios, the model will also define a set of simulator-agnostic actions for dynamic agents to use.

Driving Domain

The driving domain, `scenic.domains.driving`, is designed to support scenarios taking place on or near roads. It defines generic classes for cars and pedestrians, and provides a representation of a road network that can be loaded from standard map formats (e.g. OpenDRIVE). The domain supports dynamic scenarios, providing actions for agents which can drive and walk as well as implementations of common behaviors like lane following and collision avoidance. See the documentation of the `scenic.domains.driving` module for further details.

1.14 Supported Simulators

Scenic is designed to be easily interfaced to any simulator (see *Interfacing to New Simulators*). On this page we list interfaces that we and others have developed; if you have a new interface, let us know and we'll list it here!

Supported Simulators:

- *Built-in Newtonian Simulator*
- *CARLA*

- *Grand Theft Auto V*
- *LGSVL*
- *Webots*
- *X-Plane*

1.14.1 Built-in Newtonian Simulator

To enable debugging of dynamic scenarios without having to install an external simulator, Scenic includes a simple Newtonian physics simulator. The simulator supports scenarios written using the cross-platform *Driving Domain*, and can render top-down views showing the positions of objects relative to the road network. See the documentation of the `scenic.simulators.newtonian` module for details.

1.14.2 CARLA

Our interface to the CARLA simulator enables using Scenic to describe autonomous driving scenarios. The interface supports dynamic scenarios written using the CARLA world model (`scenic.simulators.carla.model`) as well as scenarios using the cross-platform *Driving Domain*. To use the interface, please follow these instructions:

1. Install the latest version of CARLA (we've tested versions 0.9.9 through 0.9.14) from the CARLA Release Page. Note that CARLA currently only supports Linux and Windows.
2. Install Scenic in your Python virtual environment as instructed in *Getting Started with Scenic*.
3. Within the same virtual environment, install CARLA's Python API. How to do this depends on the CARLA version and whether you built it from source:

   0.9.12+

   Run the following command, replacing `X.Y.Z` with the version of CARLA you installed:

   ```
   python -m pip install carla==X.Y.Z
   ```

   Older Versions

   For older versions of CARLA, you'll need to install its Python API from the provided `.egg` file. If your system has the **`easy_install`** command, you can run:

   ```
   easy_install /PATH_TO_CARLA_FOLDER/PythonAPI/carla/dist/carla-0.9.9-py3.7-
   ↪linux-x86_64.egg
   ```

   The exact name of the `.egg` file may vary depending on the version of CARLA you installed; make sure to use the file for Python 3, not 2. You may get an error message saying `Could not find suitable distribution`, which you can ignore.

   The **`easy_install`** command is deprecated and may not exist if you have a newer version of Python. In that case, you can try setting your `PYTHONPATH` environment variable to include the egg with a command like:

   ```
   export PYTHONPATH=/PATH_TO_CARLA_FOLDER/PythonAPI/carla/dist/carla-0.9.9-
   ↪py3.7-linux-x86_64.egg
   ```

   Built from Source

   If you built CARLA from source, the process is more involved: see the detailed instructions here.

You can check that the `carla` package was correctly installed by running **`python -c 'import carla'`**: if it prints `No module named 'carla'`, the installation didn't work. We suggest upgrading to a newer version of CARLA so that you can use **`pip`** to install the Python API.

To start CARLA, run the command **`./CarlaUE4.sh`** in your CARLA folder. Once CARLA is running, you can run dynamic Scenic scenarios following the instructions in *the dynamics tutorial*.

1.14.3 Grand Theft Auto V

The interface to Grand Theft Auto V, used in our PLDI paper, allows Scenic to position cars within the game as well as to control the time of day and weather conditions. Many examples using the interface (including all scenarios from the paper) can be found in `examples/gta`. See the paper and *`scenic.simulators.gta`* for documentation.

Importing scenes into GTA V and capturing rendered images requires a GTA V plugin, which you can find here.

1.14.4 LGSVL

We have developed an interface to the LGSVL simulator for autonomous driving, used in our ITSC 2020 paper. The interface supports dynamic scenarios written using the LGSVL world model (*`scenic.simulators.lgsvl.model`*) as well as scenarios using the cross-platform *Driving Domain*.

To use the interface, first install the simulator from the LGSVL Simulator website. Then, within the Python virtual environment where you installed Scenic, install LGSVL's Python API package from source.

An example of how to run a dynamic Scenic scenario in LGSVL is given in *Dynamic Scenarios*.

1.14.5 Webots

We have several interfaces to the Webots robotics simulator, for different use cases. Our main interface provides a generic world model that can be used with any Webots world and supports dynamic scenarios. See the `examples/webots` folder for example Scenic scenarios and Webots worlds using this interface, and *`scenic.simulators.webots`* for documentation.

Scenic also includes more specialized world models for use with Webots:

- A general model for traffic scenarios, used in our VerifAI paper. Examples using this model can be found in the VerifAI repository; see also the documentation of *`scenic.simulators.webots.road`*.

Note: The last model above, and the example `.wbt` files for it, was written for the R2018 version of Webots. Relatively minor changes would be required to make it work with the newer open source versions of Webots. We may get around to porting them eventually; we'd also gladly accept a pull request!

1.14.6 X-Plane

Our interface to the X-Plane flight simulator enables using Scenic to describe aircraft taxiing scenarios. This interface is part of the VerifAI toolkit; documentation and examples can be found in the VerifAI repository.

1.15 Interfacing to New Simulators

To interface Scenic to a new simulator, there are two steps: using the Scenic API to compile scenarios, generate scenes, and orchestrate dynamic simulations, and writing a Scenic library defining the virtual world provided by the simulator.

1.15.1 Using the Scenic API

Scenic's Python API is covered in more detail in our *Using Scenic Programmatically* page; we summarize the main steps here.

Compiling a Scenic scenario is easy: just call the `scenic.scenarioFromFile` function with the path to a Scenic file (there's also a variant `scenic.scenarioFromString` which works on strings). This returns a `Scenario` object representing the scenario; to sample a scene from it, call its `generate` method. Scenes are represented by `Scene` objects, from which you can extract the objects and their properties as well as the values of the global parameters (see the `Scene` documentation for details).

Supporting dynamic scenarios requires additionally implementing a subclass of `Simulator` which communicates periodically with your simulator to implement the actions taken by dynamic agents and read back the state of the simulation. See the `scenic.simulators.carla.simulator` and `scenic.simulators.lgsvl.simulator` modules for examples.

1.15.2 Defining a World Model

To make writing scenarios for your simulator easier, you should write a Scenic library specifying all the relevant information about the simulated world. This world model could include:

- Scenic classes (subclasses of *Object*) corresponding to types of objects in the simulator;
- instances of *Region* corresponding to locations of interest (e.g. one for each road);
- a workspace specifying legal locations for objects (and optionally providing methods for schematically rendering scenes);
- a set of actions which can be taken by dynamic agents during simulations;
- any other information or utility functions that might be useful in scenarios.

Then any Scenic programs for your simulator can import this world model and make use of the information within.

Each of the simulators natively supported by Scenic has a corresponding `model.scenic` file containing its world model. See the *Supported Simulators* page for links to the module under `scenic.simulators` for each simulator, where the world model can be found. For an example, see the `scenic.simulators.lgsvl` model, which specializes the simulator-agnostic model provided by the *Driving Domain* (in `scenic.domains.driving.model`).

1.16 Publications Using Scenic

1.16.1 Main Papers

The main paper on Scenic 2.x is:

> *Scenic: A Language for Scenario Specification and Data Generation*.
> Fremont, Kim, Dreossi, Ghosh, Yue, Sangiovanni-Vincentelli, and Seshia.
> *Machine Learning*, 2022. [available here]
> (see also the full version with appendices)

Our journal paper extends the earlier conference paper on Scenic 1.0:

> *Scenic: A Language for Scenario Specification and Scene Generation*.
> Fremont, Dreossi, Ghosh, Yue, Sangiovanni-Vincentelli, and Seshia.
> PLDI 2019. [full version]

An expanded version of this paper appears as Chapters 5 and 8 of this thesis:

> *Algorithmic Improvisation*. [thesis]
> Daniel J. Fremont.
> Ph.D. dissertation, 2019 (University of California, Berkeley; Group in Logic and the Methodology of Science).

Scenic is also integrated into the VerifAI toolkit, which is described in another paper:

> *VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems*.
> Dreossi*, Fremont*, Ghosh*, Kim, Ravanbakhsh, Vazquez-Chanlatte, and Seshia.
> CAV 2019.

* Equal contribution.

1.16.2 Case Studies

We have also used Scenic in several industrial case studies:

> *Formal Analysis and Redesign of a Neural Network-Based Aircraft Taxiing System with VerifAI*.
> Fremont, Chiu, Margineantu, Osipychev, and Seshia.
> CAV 2020.

> *Formal Scenario-Based Testing of Autonomous Vehicles: From Simulation to the Real World*.
> Fremont, Kim, Pant, Seshia, Acharya, Bruso, Wells, Lemke, Lu, and Mehta.
> ITSC 2020.
> [See also this white paper and associated blog post]

1.16.3 Other Papers Building on Scenic

> *A Programmatic and Semantic Approach to Explaining and Debugging Neural Network Based Object Detectors*.
> Kim, Gopinath, Pasareanu, and Seshia.
> CVPR 2020.

1.17 Credits

If you use Scenic, we request that you cite our 2022 journal paper and/or our original PLDI 2019 paper.

Scenic is primarily maintained by Daniel J. Fremont.

The Scenic project was started at UC Berkeley in Sanjit Seshia's research group.

The language was initially developed by Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia.

Edward Kim assisted in developing the library for dynamic driving scenarios and putting together this documentation.

Eric Vin, Matthew Rhea, and Ellen Kalvan developed Scenic's support for 3D geometry. Shun Kashiwa developed the auto-generated parser for Scenic 3.0 and its support for temporal requirements.

The Scenic tool and example scenarios have benefitted from additional code contributions from:

- Johnathan Chiu
- Greg Crow
- Francis Indaheng
- Ellen Kalvan
- Martin Jansa (LG Electronics, Inc.)
- Kevin Li
- Guillermo López
- Shalin Mehta
- Joel Moriana
- Gaurav Rao
- Matthew Rhea
- Ameesh Shah
- Jay Shenoy
- Eric Vin
- Kesav Viswanadha
- Wilson Wu

Finally, many other people provided helpful advice and discussions, including:

- Ankush Desai
- Alastair Donaldson
- Andrew Gordon
- Steve Lemke
- Jonathan Ragan-Kelley
- Sriram Rajamani
- German Ros
- Marcell Vazquez-Chanlatte

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- glossary

CHAPTER

THREE

LICENSE

Scenic is distributed under the 3-Clause BSD License.

BIBLIOGRAPHY

[F22] Fremont et al., *Scenic: A Language for Scenario Specification and Data Generation*, Machine Learning, 2022. [Online]

[F19] Fremont et al., *Scenic: A Language for Scenario Specification and Scene Generation*, PLDI 2019.

[B10] Bauer et al., *Comparing LTL Semantics for Runtime Verification*. Journal of Logic and Computation, 2010. [Online]

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y

Z