Scenic

Daniel J. Fremont, Edward Kim, Tommaso Dreossi, Shromona Gh

Sep 06, 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, 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 2.x is not completely backwards-compatible with 1.x, which was used in our papers prior to late 2020. See *What's New in Scenic* for a list of syntax changes and new features. If your existing code no longer works, install the latest 1.x release 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 3.7** or newer. You can install Scenic from PyPI by simply running:

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

Alternatively, if you want to run some of our example scenarios, modify Scenic, or make use of features that have not yet been released on PyPI, you can download or clone the Scenic repository. Activate the virtual environment in which you would like to install Scenic, go into the root folder of the Scenic repository, and then run:

```
$ python -m pip install -e .
```

Either installation method will install all of the dependencies which are required to run Scenic. If you will be developing Scenic, you will want to use a variant of this command: see *Developing Scenic*.

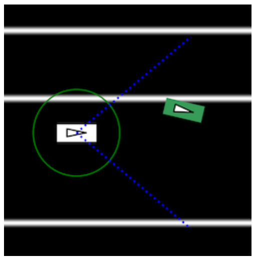
Note: If you are using Windows, or encounter any errors during installation or trying the examples below, please see our *Notes on Installing Scenic* for suggestions.

1.1.2 Trying Some Examples

The Scenic repository contains many example scenarios, found in the `examples` directory. They are organized by the simulator they are written for, e.g. GTA (Grand Theft Auto V) or Webots; there are also cross-platform scenarios written for Scenic's abstract application domains, e.g. the *driving domain*. 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/gta/badlyParkedCar2.scenic
```

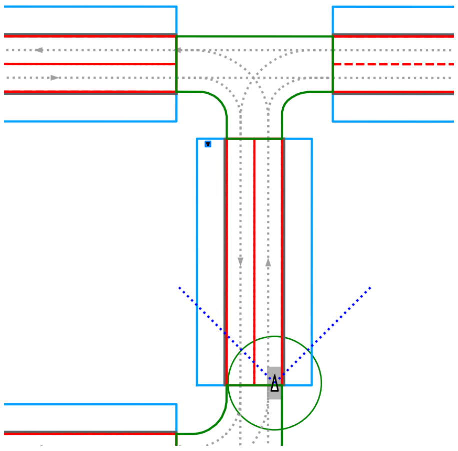
This will compile the Scenic program and sample from it, displaying a schematic of the resulting scene. Since this is the badly-parked car example from our GTA case study, you should get something like this:

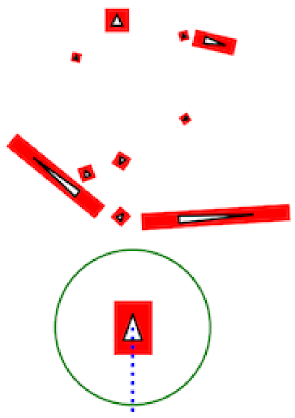


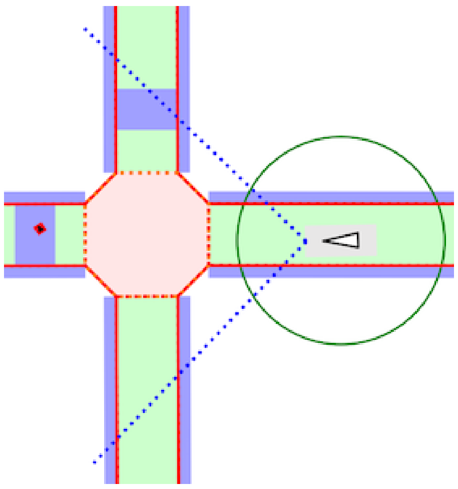
Here the circled rectangle is the ego car; its view cone extends to the right, where we see another car parked rather poorly at the side of the road (the white lines are curbs). If you close the window, Scenic will sample another scene from the same scenario and display it. This will repeat until you kill the generator (`Control-c` in Linux; right-clicking on the Dock icon and selecting Quit on OS X).

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

```
$ scenic examples/driving/pedestrian.scenic
$ scenic examples/webots/mars/narrowGoal.scenic
$ scenic 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 the *Scenic Tutorial*.
- 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 during installation you get an error saying that your machine does not have a compatible version, this means that you do not have Python 3.7 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.

Note: If you are using Poetry to manage your virtual environments, and you install the new version of Python somewhere that is not on your PATH (so that running `python --version` doesn't give you the correct version), you'll need to run `poetry env use /full/path/to/python` before running `poetry install`.

“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.

Note: If you used Poetry to install Scenic, the **`poetry install`** command may have created a virtual environment for you. You can then run **`poetry shell`** to create a terminal inside the environment and run Scenic as usual.

Scene Schematics Don’t Appear

If no window appears when you ask Scenic to generate and display a scene (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 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. If you'd like to use Poetry, it can be installed on WSL in the same way as on Linux: see the instructions here.

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. Note that after applying this fix the command **`poetry shell`** may not work and one may need to use **`source $(poetry env info --path)/bin/activate`** instead.

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 Scenic Tutorial

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. Throughout, we use examples from our case study using Scenic to generate traffic scenes in GTA V to test and train autonomous cars [F22], [F19].

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

1.3.1 Classes, Objects, and Geometry

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 = Car
3 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*. 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 (we'll see an example momentarily).

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:

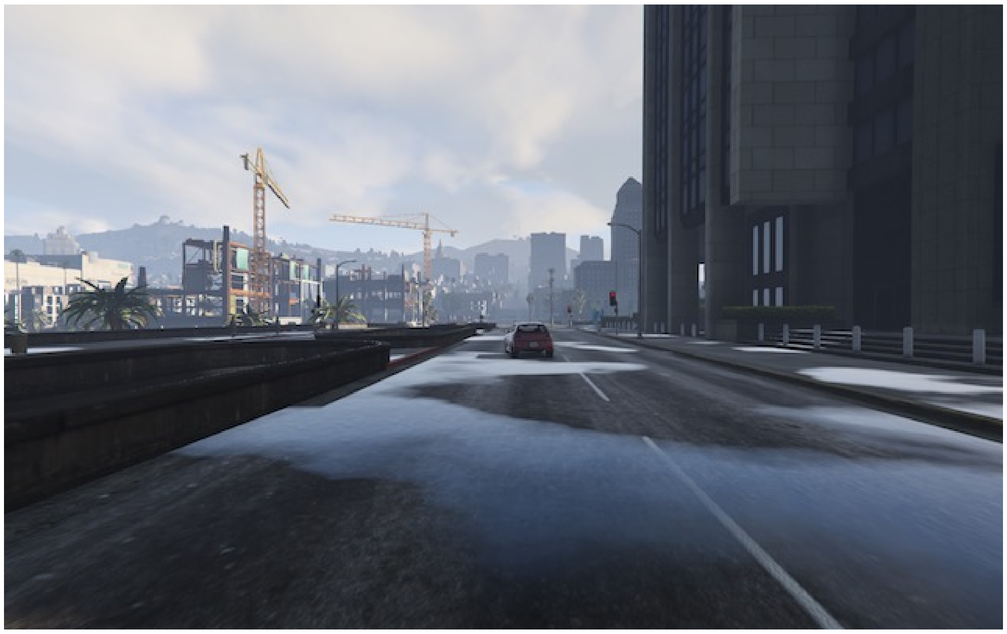


Fig. 1: A scene sampled from the simple car scenario, rendered in GTA V.

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 in Scenic, any unspecified properties take on the *default values* inherited from the object's class. Slightly simplified, the definition of the class `Car` begins:

```
class Car:
    position: Point on road
    heading: roadDirection at self.position
    width: self.model.width
    height: self.model.height
    model: CarModel.defaultModel()      # a distribution over several car models
```

Here `road` is a *region*, one of Scenic's primitive types, 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 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 `Point on road` (we will explain this syntax shortly), 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.

We can of course override the class-provided defaults and define the position of an object more specifically. For example,

```
Car offset by Range(-10, 10) @ Range(20, 40)
```

creates a car that is 20–40 meters ahead of the camera (the `ego`), and up to 10 meters to the left or right, while still using the default heading (namely, being aligned with the road). Here `Range(X, Y)` creates a uniform distribution on the interval between `X` and `Y`, and `X @ Y` creates a vector from *xy* coordinates as in Smalltalk [GR83]. If you prefer, you can give a list or tuple of *xy* coordinates instead, e.g.:

```
Car offset by (Range(-10, 10), Range(20, 40))
```

One exception to the above rules for object creation is that if the name of a class is followed immediately by punctuation, then an object is not created. This allows us to refer to a Scenic class without creating an instance of that class in the environment, which is useful for statements like `isinstance(obj, Car)`, `[Taxi, Truck]`, `Car.staticMethod`, etc.

1.3.2 Local Coordinate Systems

Scenic provides a number of constructs for working with local coordinate systems, which are often helpful when building a scene incrementally out of component parts. Above, we saw how `offset by` could be used to position an object in the coordinate system of the `ego`, for instance placing a car a certain distance away from the camera[1].

It is equally easy in Scenic to use local coordinate systems around other objects or even arbitrary points. For example, suppose we want to make the scenario above more realistic by not requiring the car to be *exactly* aligned with the road, but to be within say 5°. We could write

```
Car offset by Range(-10, 10) @ Range(20, 40),
    facing Range(-5, 5) deg
```

but this is not quite what we want, since this sets the orientation of the car in *global* coordinates. Thus the car will end up facing within 5° of North, rather than within 5° of the road direction. Instead, we can use Scenic's general operator `X relative to Y`, which can interpret vectors and headings as being in a variety of local coordinate systems:

```
Car offset by Range(-10, 10) @ Range(20, 40),
    facing Range(-5, 5) deg relative to roadDirection
```

If instead we want the heading to be relative to that of the ego car, so that the two cars are (roughly) aligned, we can simply write `Range(-5, 5) deg relative to ego`.

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. This is a feature of Scenic's system of *specifiers*, which we explain next.

[1] In fact, *ego* is a variable and can be reassigned, so 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.

1.3.3 Readable, Flexible Specifiers

The syntax `offset by` *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 an object:

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 `heading`; 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
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
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 factors the definition of objects into potentially-interacting but syntactically-independent parts:

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

Here `left of` *X* `by` *D* and `with model` *M* are *specifiers* which do not have an order, but which *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.

1.3.4 Specifying Multiple Properties Together

Recall that we defined the default `position` for a `Car` to be a `Point on road`: this is an example of another specifier, `on region`, which specifies `position` to be a uniformly random point in the given region. This specifier illustrates another feature of Scenic, namely that specifiers can specify multiple properties simultaneously. Consider the following scenario, which creates a parked car given a region `curb` (also defined in the *scenic.simulators.gta.model* library):

```
spot = OrientedPoint on visible curb
Car left of spot by 0.25
```

The function `visible region` returns the part of the region that is visible from the ego object. The specifier `on visible curb` will then set `position` to be a uniformly random visible point on the curb. We create `spot` as an *OrientedPoint*, which is a built-in class that defines a local coordinate system by having both a `position` and a `heading`. The `on region` specifier can also specify `heading` if the region has a preferred orientation (a vector field) associated with it: in our example, `curb` is oriented by `roadDirection`. So `spot` is, in fact, a uniformly random visible point on the curb, oriented along the road. That orientation then causes the `Car` to be placed 0.25 m left of `spot` in `spot`'s local coordinate system, i.e. 0.25 m away from the curb, as desired.

In fact, Scenic makes it easy to elaborate this scenario without needing to alter the code above. Most simply, we could specify a particular model or non-default distribution over models by just adding `with model M` to the definition of the `Car`. More interestingly, we could produce a scenario for *badly*-parked cars by adding two lines:

```
spot = OrientedPoint on visible curb
badAngle = Uniform(1, -1) * Range(10, 20) deg
Car left of spot by 0.25,
    facing badAngle relative to roadDirection
```

This will yield cars parked 10-20° off from the direction of the curb, as seen in the image below. This example illustrates how specifiers greatly enhance Scenic's flexibility and modularity.

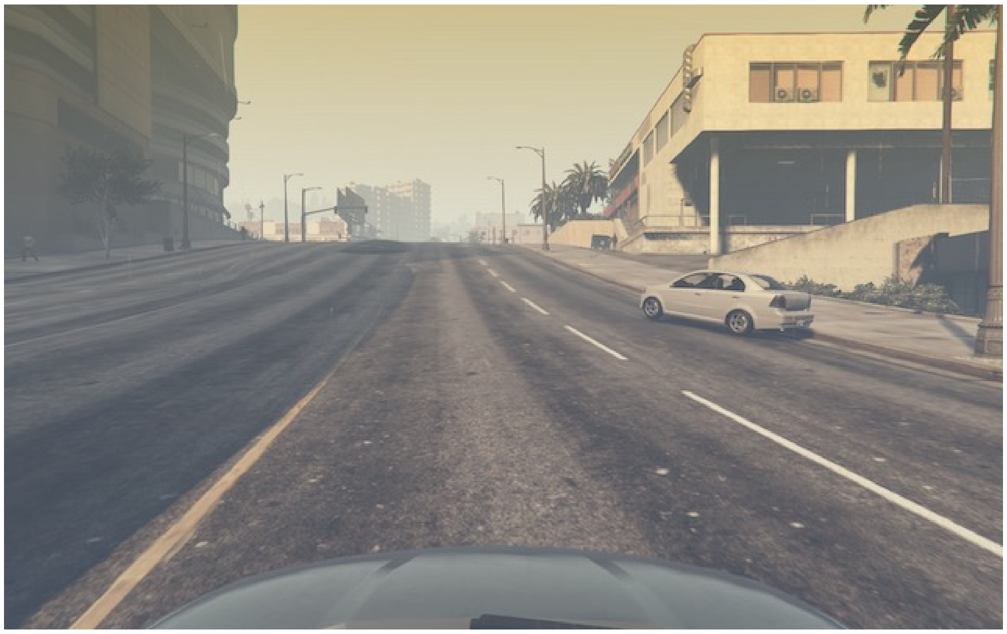


Fig. 2: A scene sampled from the badly-parked car scenario, rendered in GTA V.

1.3.5 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*:

- 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).
- Objects must be visible from the ego object (so that they affect the rendered image; this requirement can also be disabled, for example for dynamic scenarios).

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 = Car on road
car2 = 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[2]. 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:

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

[2] 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`).

1.3.6 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 `heading` 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 `heading` (see *scenic.core.object_types.Mutator*).

1.3.7 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 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.

We will write a scenario representing a rubble field of rocks and piples 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 (*scenic.simulators.webots.mars.model*) which defines the (empty) workspace and several types of objects: the **Rover** itself, the **Goal** (represented by a flag), and debris classes **Rock**, **BigRock**, and **Pipe**. **Rock** and **BigRock** have fixed sizes, and the rover can 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).

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

Then we create the rover at a fixed position and the goal at a random position on the other side of the workspace:

```
ego = Rover at 0 @ -2
goal = Goal at Range(-2, 2) @ Range(2, 2.5)
```

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.

```
bottleneck = OrientedPoint 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
BigRock at bottleneck
```

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, with their ends far enough apart for the robot to be able to pass between:

```
halfGapWidth = (1.2 * ego.width) / 2
leftEnd = OrientedPoint left of bottleneck by halfGapWidth,
                        facing Range(60, 120) deg relative to bottleneck
rightEnd = OrientedPoint right of bottleneck by halfGapWidth,
                         facing Range(-120, -60) deg relative to bottleneck
```

(continues on next page)

(continued from previous page)

```
Pipe ahead of leftEnd, with length Range(1, 2)
Pipe ahead of rightEnd, with length Range(1, 2)
```

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).

```
BigRock beyond bottleneck by Range(-0.5, 0.5) @ Range(0.5, 1)
BigRock beyond bottleneck by Range(-0.5, 0.5) @ Range(0.5, 1)
Pipe
Rock
Rock
Rock
```

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

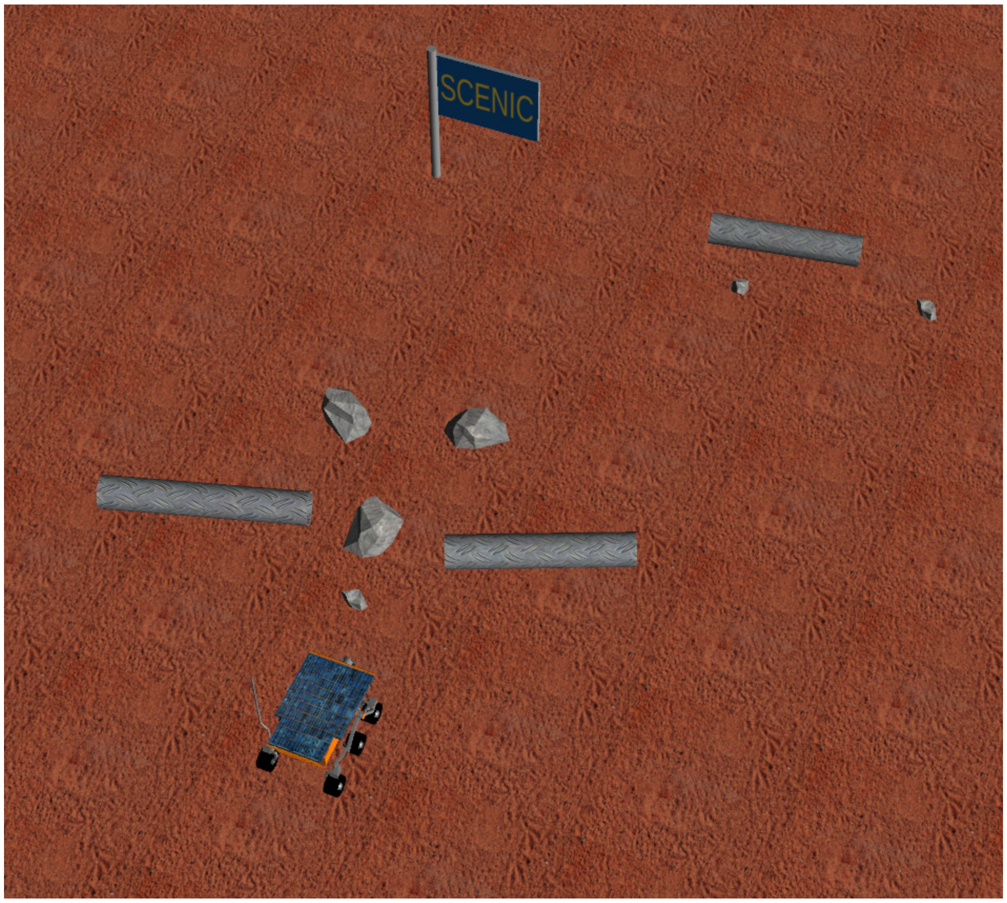
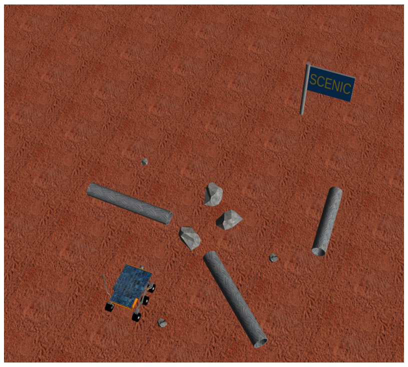
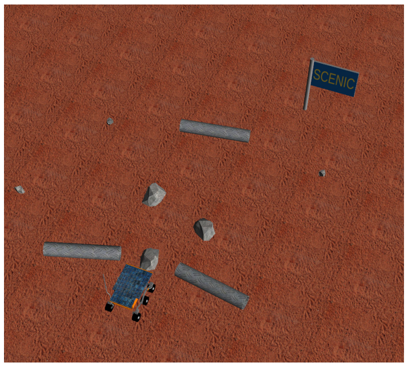
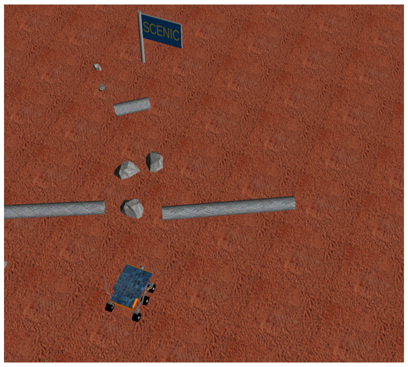


Fig. 3: A scene sampled from the Mars rover scenario, rendered in Webots.







1.3.8 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.

Our page on *Dynamic Scenarios* describes how to define scenarios with dynamic agents that move or take other actions over time.

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

1.4 Dynamic Scenarios

The *Scenic Tutorial* 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.

1.4.1 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
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)
```

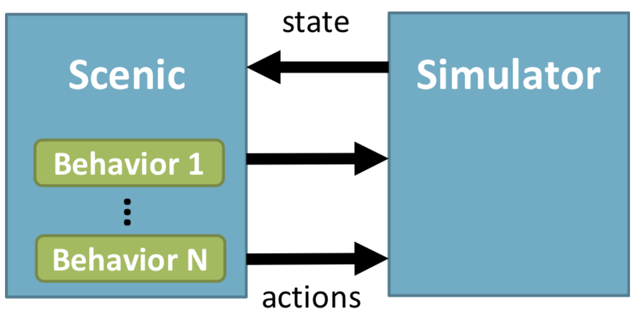
We intend this behavior to run for the entire scenario, so we use an infinite loop. In each step of the loop, we compute appropriate throttle and steering controls, then use the `take` statement to take the corresponding actions. When that statement is executed, Scenic pauses the behavior until the next 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 = Car
car2 = Car visible, with behavior WaitUntilClose
car3 = 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).

1.4.2 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.

1.4.3 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.

1.4.4 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))
```

The *require always* condition statement enforces that the given condition must hold at every time step of the scenario; if it is ever violated during a simulation, we reject that simulation and sample a new one. Similarly, we can require that a condition hold at *some* time during the scenario using the *require eventually* statement:

```
require eventually ego in intersection
```

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:

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

The requirement ensures that no pedestrian comes close to *self* until the ego does; after that, we place no further restrictions.

To enforce more complex temporal properties like this one without modifying behaviors, 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 the property "car1 and car2 enter the intersection before car3":

```
1  monitor Car3EntersLast:
2      seen1, seen2 = False, False
3      while not (seen1 and seen2):
4          require car3 not in intersection
5          if car1 in intersection:
6              seen1 = True
7          if car2 in intersection:
8              seen2 = True
9          wait
```

We use the variables seen1 and seen2 to remember whether we have seen car1 and car2 respectively enter the intersection. The loop will iterate as long as at least one of the cars has not yet entered the intersection, so if car3 enters before either car1 or car2, the requirement on line 4 will fail and we will reject the simulation. Note the necessity of the *wait* statement on line 9: if we omitted it, the loop could run forever without any time actually passing in the simulation.

1.4.5 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:
            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.

1.4.6 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 = Point on road
ego = 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.

1.4.7 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
```

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 \
    --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 \
    --simulate \
    --model scenic.simulators.lgsvl.model \
    --time 200 \
    --param map tests/formats/opendrive/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[1] and restart it with different arguments.

1.4.8 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).

1.5 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.

1.5.1 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 = OrientedPoint on visible ego.laneGroup.curb
        parkedCar = Car left of spot by gap
```

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

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 = 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.

1.5.2 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:

```
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] 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.

1.5.3 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 = 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 = 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
        Pedestrian on visible road.sidewalkRegion, with behavior ...
```

1.5.4 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]

1.6 Syntax Guide

This page summarizes the syntax of Scenic (excluding syntax inherited from Python). For more details, click the links for individual language constructs to go to the corresponding section of the *Language Reference*.

1.6.1 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 orientations in space
Vector Fields	associating an orientation (i.e. a heading) to each point in space
Regions	representing sets of points in space

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

1.6.2 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
`Point (in | on) region`	uniformly-distributed *Point* in a region

1.6.3 Statements

Compound Statements

Syntax	Meaning
`class name[(superclass)]:`	Defines a Scenic class.
`behavior name(arguments):`	Defines a dynamic behavior.
`monitor name:`	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 (always | eventually) boolean`	Define a dynamic hard requirement.
`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.
`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.

1.6.4 Objects

The syntax `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)	position in global coordinates
viewDistance	50	distance for the 'can see' operator
mutationScale	0	overall scale of *mutations*
positionStdDev	1	mutation standard deviation for `position`

Properties added by *OrientedPoint*:

Property	Default	Meaning
heading[1]	0	heading in global coordinates
viewAngle	360 degrees	angle for the 'can see' operator
headingStdDev	5 degrees	mutation standard deviation for `heading`

Properties added by *Object*:

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

Property	Default	Meaning
width	1	width of bounding box (X axis)
length	1	length of bounding box (Y axis)
speed[1]	0	initial speed (later, instantaneous speed)
velocity[Page 29, 1]	from speed	initial velocity (later, instantaneous velocity)
angularSpeed[Page 29, 1]	0	angular speed (change in heading/time)
behavior	None	dynamic behavior, if any
allowCollisions	False	whether collisions are allowed
requireVisible	True	whether object must be visible from ego
regionContainedIn	workspace	Region the object must lie within
cameraOffset	(0, 0)	position of camera for 'can see'

1.6.5 Specifiers

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

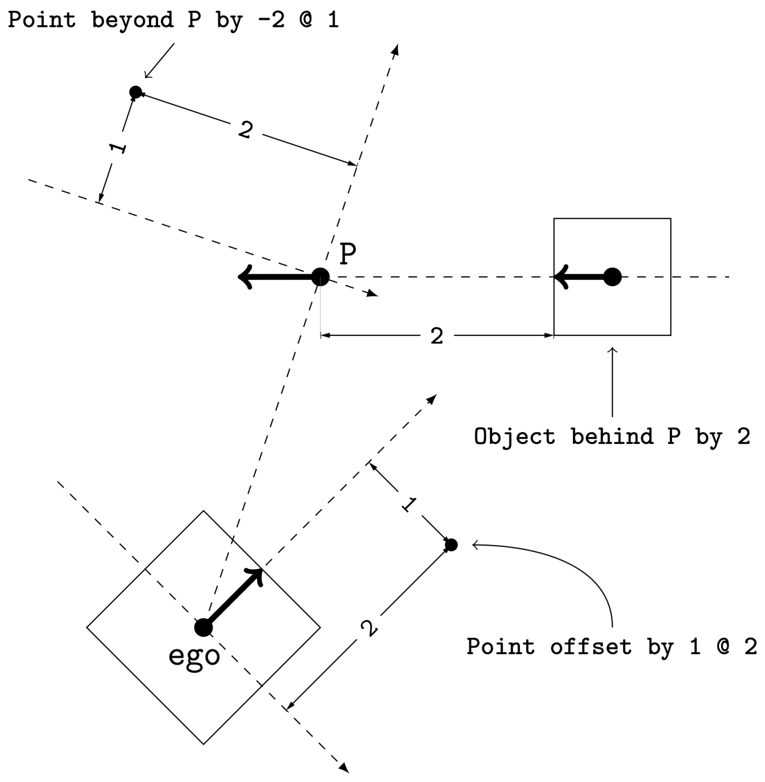


Fig. 4: Illustration of the beyond, behind, and offset by specifiers. Each *OrientedPoint* (e.g. P) is shown as a bold arrow.

Specifier for `position`	Meaning
at vector	Positions the object at the given global coordinates
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
(left | right) of vector *[by* scalar*]*	Positions the object further to the left/right by the given scalar distance
(ahead of | behind) vector *[by* scalar*]*	As above, except placing the object ahead of or behind the given position
beyond vector *by* vector *[from* vector*]*	Positions the object at coordinates given by the second vector, centered at the first vector and oriented along the line of sight from the third vector/ego
*visible [from (*Point *|* OrientedPoint*)]*	Positions the object uniformly at random in the visible region of the ego, or of the given Point/OrientedPoint if given
*not visible [from (*Point *|* OrientedPoint*)]*	Positions the object uniformly at random in the non-visible region of the ego, or of the given Point/OrientedPoint if given

Specifier for `position` and optionally `heading`	Meaning
(in | on) region	Positions the object uniformly at random in the given Region
*(left | right) of (*OrientedPoint *|* Object*) [by* scalar*]*	Positions the object to the left/right of the given OrientedPoint, depending on the object's width
*(ahead of | behind) (*OrientedPoint *|* Object*) [by* scalar*]*	As above, except positioning the object ahead of or behind the given OrientedPoint, thereby depending on length
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 `heading`	Meaning
facing heading	Orients the object along the given heading 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)
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)

1.6.6 Operators

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 to the given position from ego (or the `from` vector)

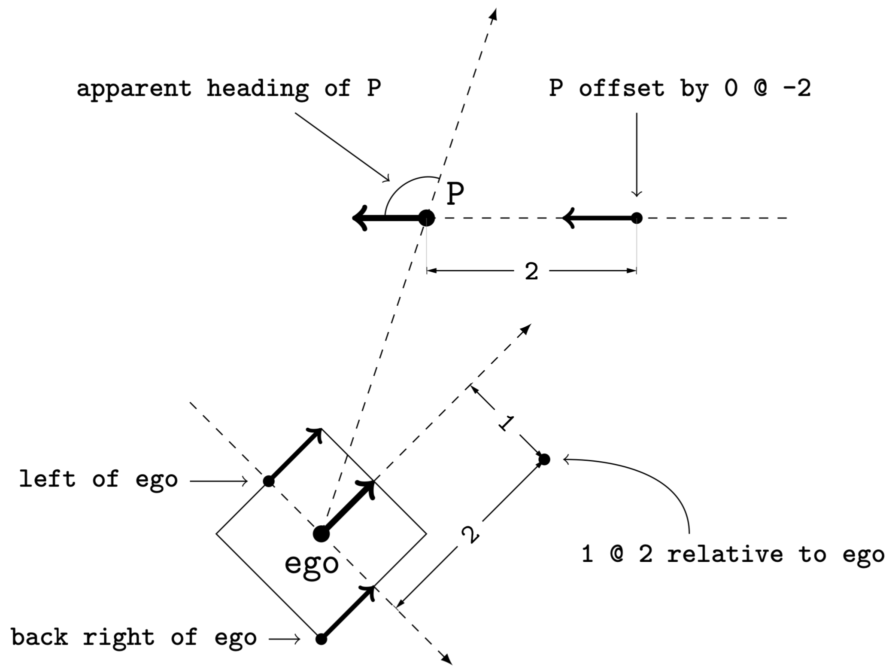


Fig. 5: Illustration of several operators. Each ***OrientedPoint*** (e.g. P) is shown as a bold arrow.

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

Heading Operators	Meaning
scalar *deg*	The given heading, interpreted as being in degrees
vectorField *at* vector	The heading specified by the vector field at the given position
direction *relative to* direction	The first direction, 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

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 edge of the bounding box of the Object, oriented along its heading
`(front | back) (left | right) of` `Object`	The corresponding corner of the Object's bounding box, also oriented along its heading

1.6.7 Built in Functions

Function	Description
Misc Python functions	Various Python functions including `min`, `max`, `sin`, `cos`, 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.

1.7 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*.

1.7.1 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 with the syntax `X @ Y` (inspired by Smalltalk); using a length-2 list or tuple (`[X, Y]` or `(X, Y)`) is also allowed. By convention, coordinates are in meters, although the semantics of Scenic does not depend on this. More significantly, the vector syntax is specialized for 2-dimensional space. The 2D assumption dramatically simplifies much of Scenic's syntax (particularly that dealing with orientations, as we will see below), while still being adequate for a variety of applications. Some world models, such as that for the *Driving Domain*, define an `elevation` property which defines a 3D location in combination with Scenic's 2D `position` property. A future extension of Scenic will natively support 3D space.

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)`.

Heading

Headings represent orientations in space. Conveniently, in 2D these can be expressed using a single angle (rather than Euler angles or a quaternion). 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.

Vector Field

Vector fields associate an orientation (i.e. a heading) 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`).

Region

Regions represent sets of points in space. Scenic provides a variety of ways to define regions: 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`, and the `(in | on) region` specifier to choose a position uniformly at random inside a region.

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 summarized below: if you are working on Scenic's internals, see the `scenic.core.regions` module for full details.

Abstract Regions

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

Abstract class for regions.

intersect(*other*)

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

intersects(*other*)

Check if this *Region* intersects another.

Return type
bool

union(*other*)

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

Not supported by all region types.

Return type
Region

Simple Shapes

Unlike the more complex regions, these 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.

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.

Polylines and Polygons

These subclasses represent fixed 1D and 2D regions defined by line segments and polygons.

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 heading 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 heading 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 PolygonalRegion`(*points=None*, *polygon=None*, *orientation=None*, *name=None*)

Region given by one or more polygons (possibly with holes).

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).
- **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*.

Point Sets and Grids

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 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

1.7.2 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 | on) region` specifier. 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.

1.7.3 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 be overridden in a subclass. 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>:
    <statement>+
```

Defines a Scenic monitor, which runs 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 `terminate` the simulation). The main purpose of monitors is to evaluate complex temporal properties that do not fit into the `require always` and `require eventually` 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*.

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.

Note: Scenic modules can only be imported at the top level, or in a top-level try-except block that does not create any objects (so that you can catch `ModuleNotFoundError` for example). Python modules can be imported dynamically inside functions as usual.

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 (always | eventually) *boolean*

Require a condition hold at each time step (`always`) or at some point during the simulation (`eventually`).

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 *Oriented-Point*) 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.

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.

1.7.4 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

```
<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.

Names of Scenic classes followed immediately by punctuation are not considered instance creations. This allows us to refer to a Scenic class without creating an instance of that class in the environment, which is useful for expressions like `isinstance(obj, Car)`, `[Taxi, Truck]`, `Car.staticMethod`, etc.

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.
- **visibleDistance** (*float*) – Distance for `can see` operator. Default value 50.
- **width** (*float*) – Default value zero (only provided for compatibility with operators that expect an *Object*).
- **length** (*float*) – Default value zero.
- **mutationScale** (*float*) – Overall scale of mutations, as set by the `mutate` statement. Default value zero (mutations disabled).
- **positionStdDev** (*float*) – Standard deviation of Gaussian noise to add to this object's `position` when mutation is enabled with scale 1. Default value 1.

property visibleRegion

The visible region of this object.

The visible region of a *Point* is a disc 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 `heading` and several associated properties.

class OrientedPoint(*<specifiers>*)

The Scenic class `OrientedPoint`.

The default mutator for *OrientedPoint* adds Gaussian noise to `heading` with a standard deviation given by the `headingStdDev` property, then applies the mutator for *Point*.

Properties

- **heading** (*float; dynamic*) – Heading of the *OrientedPoint*. Default value 0 (North).
- **viewAngle** (*float*) – View cone angle for `can see` operator. Default value 2.
- **headingStdDev** (*float*) – Standard deviation of Gaussian noise to add to this object's `heading` when mutation is enabled with scale 1. Default value 5°.

`property visibleRegion`

The visible region of this object.

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`.

Object

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

- `width` and `length` to define the bounding box 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, unless its `requireVisible` property is set to `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`.

`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 1.
- **length** (*float*) – Length of the object, i.e. extent along its Y axis. Default value 1.
- **allowCollisions** (*bool*) – Whether the object is allowed to intersect other objects. Default value `False`.
- **requireVisible** (*bool*) – Whether the object is required to be visible from the `ego` object. Default value `True`.
- **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.
- **cameraOffset** (*Vector*) – Position of the camera for the `can see` operator, relative to the object's `position`. Default `(0, 0)`.
- **speed** (*float; dynamic*) – Speed in dynamic simulations. Default value 0.
- **velocity** (*Vector*; *dynamic*) – Velocity in dynamic simulations. Default value is the velocity determined by `self.speed` and `self.heading`.

- **angularSpeed** (*float; dynamic*) – Angular speed in dynamic simulations. Default value 0.
- **behavior** – Behavior for dynamic agents, if any (see *Dynamic Scenarios*). Default value `None`.

`startDynamicSimulation()`

Hook called at the beginning of each dynamic simulation.

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

`property visibleRegion`

The visible region of this object.

The visible region of an *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).

1.7.5 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 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 *optionally* specifies the `heading` property (to be the same as *X*), meaning that it assigns a value to `heading` if no other specifier does so: if we write `Object left of X by Y, facing Z`, then the new object's `heading` property will be determined by `facing`, not `left of`. 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*

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

Position Specifiers

at *vector*

Positions the object at the given global coordinates.

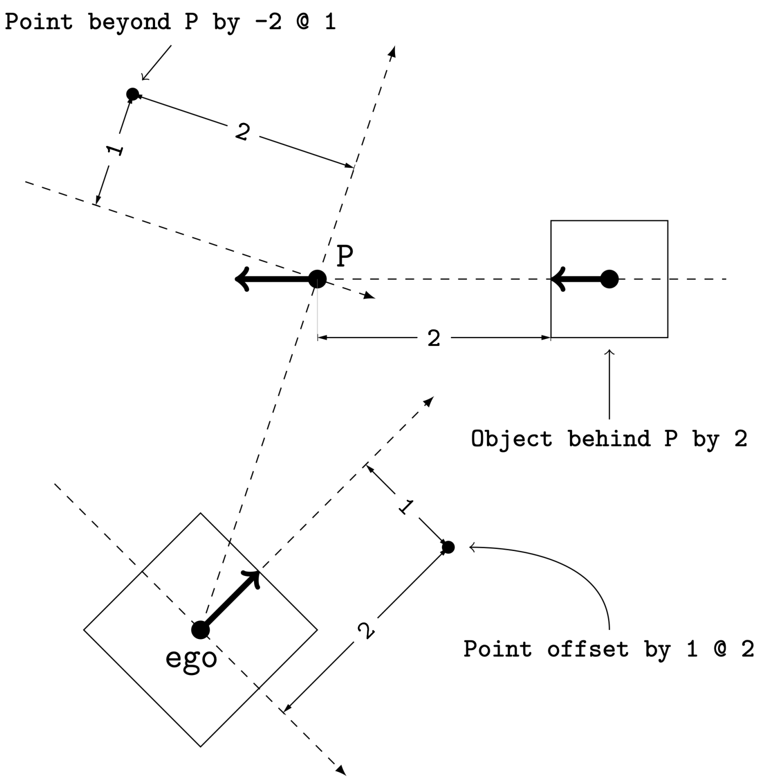


Fig. 6: Illustration of the beyond, behind, and offset by specifiers. Each *OrientedPoint* (e.g. P) is shown as a bold arrow.

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 (which, if a vector field, is evaluated at ego to obtain a heading).

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

Depends on `heading` and `width`. 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 object's right/left edge is at that position. If `by scalar` is used, the object is placed further to the left/right by the given distance.

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

As above, except placing the object ahead of or behind the given position (so that the midpoint of the object's back/front edge is at that position); thereby depending on `heading` and `length`.

beyond *vector* by *vector* [from *vector*]

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. a heading of 0 in the local coordinate system faces directly away from the first vector). If no third vector is provided, it is assumed to be the ego. For example, `beyond taxi by (0, 3)` means 3 meters directly behind the taxi as viewed by the camera.

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

Positions the object uniformly at random in the visible region of the ego, or of the given Point/OrientedPoint if given. More precisely, this specifier sets the `position` of the object being created (i.e. its center) to be a uniformly-random point in the visible region. (This specifier is therefore slightly stricter than a requirement that the ego `can see` the object: the specifier makes the *center* visible, while the `can see` condition will be satisfied if the center is not visible but some other part of the object is visible.)

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

Like *`visible [from (Point | OrientedPoint)]`* except it positions the object uniformly at random in the **non-visible** region of the ego. 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).

(in | on) *region*

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

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

Positions the object to the left/right of the given *OrientedPoint*, depending on the object's `width`. Also optionally specifies `heading` to be the same as that of the OrientedPoint. If the OrientedPoint is in fact an *Object*, the object being constructed is positioned to the left/right of its left/right edge (i.e. the `width` of both objects is taken into account).

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

As above, except positioning the object ahead of or behind the given OrientedPoint, thereby depending on `length`.

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

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`). Optionally specifies `heading` to be the heading 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.

Heading Specifiers

facing *heading*

Orients the object along the given heading 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 so that it faces 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 `from` vector). For example, `apparently facing 90 deg` orients the object so that the 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, both non-optionally and optionally, 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 non-optional specifier if one exists; otherwise its unique optional specifier if one exists; or finally by its default value if no specifiers apply at all (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 non-optionally by mutiple 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 specifier in S specifying it, using the following precedence order: non-optional specifier, optional specifier, then default value.
4. The dependency graph of the specifiers S is constructed. 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.

1.7.6 Operators Reference

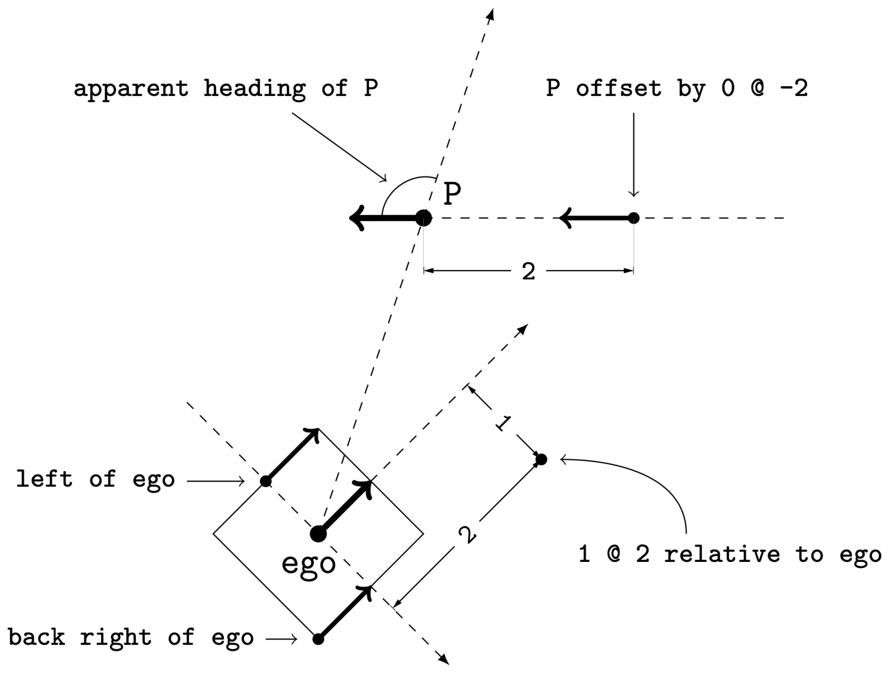


Fig. 7: Illustration of several operators. Each *OrientedPoint* (e.g. P) is shown as a bold arrow.

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 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

Boolean Operators

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

Whether or not a position or *Object* is visible from a *Point* or *OrientedPoint*. Visible regions are defined as follows: a *Point* can see out to a certain distance, and an *OrientedPoint* restricts this to the circular sector along its heading with a certain angle. A position is then visible if it lies in the visible region, and an *Object* is visible if its bounding box intersects the visible region.

Note: Technically, Scenic only checks that a corner of the object is visible, which could result in the side of a large object being visible but Scenic not counting it as so. Scenic's visibility model also does not take into account occlusion, although this would be straightforward to add.

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

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

Heading Operators

scalar deg

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

vectorField at *vector*

The heading specified by the vector field at the given position

direction relative to *direction*

The first direction, interpreted as an offset relative to the second direction. 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) relative to (100, 200)` is `(105, 205)`. Note that this polymorphic operator has a specialized version for instances of *OrientedPoint*, defined *below*: so for example `(-3, 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 a heading)

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).

OrientedPoint Operators

vector relative to *OrientedPoint*

The given vector, interpreted in the local coordinate system of the OrientedPoint. So for example `(1, 2) 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) of *Object*

The midpoint of the corresponding edge of the bounding box of the *Object*, oriented along its heading/

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

The corresponding corner of the Object's bounding box, also oriented along its heading.

1.7.7 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)
- `max`, `min`
- `str`

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)
```

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

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.

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`).

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.

1.7.8 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 is 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).

1.7.9 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 the latter also including additional checks to see if any of their `terminate when` conditions have been met or their `require always` conditions 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 `require always` conditions are violated; if so, reject the simulation.
   b. Check which `require eventually` conditions are satisfied; remember these for later.
   c. Check if the scenario's time limit (if `terminate after` has been used) has been reached; if so, go to step (f) below to stop the scenario.

[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.

   d. If the scenario is not currently running a sub-scenario (with *do*), check its invariants; if any are violated, reject the simulation.Page 58, 1

   e. 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 finishes executing, go to step (f) below to stop the scenario.

   f. If the scenario is stopping for one of the reasons above, first check if any of the *require eventually* conditions were never 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, 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.)

2. Save the values of all *record* statements, as well as *record initial* statements if it is time step 0.
3. Run each monitor 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*, 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.Page 58, 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*, 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. Update every dynamic property of every object to its current value in the simulator.
9. Increment the simulation clock (the `currentTime` attribute of *Simulation*).
10. If the simulation is stopping for one of the reasons above, first check if any of the *require eventually* conditions of any remaining scenarios were never satisfied: if so, reject the simulation. Otherwise, save the values of any *record final* statements.

1.8 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]
```

1.8.1 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`, so external Python code called from within Scenic can also be made deterministic (although `random` should not be used in place of Scenic's own sampling constructs in Scenic code). Note though that NumPy provides other RNGs whose seeds are not set by this option (see `numpy.random`).

--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`.

1.8.2 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.

1.8.3 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`*.

1.9 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.

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*, *cacheImports=False*)

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.
- **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.

Returns
A *Scenario* object representing the Scenic scenario.

scenarioFromString(*string*, *params={}*, *model=None*, *scenario=None*, *filename='<string>'*, *cacheImports=False*)

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 = Object with foo Range(0, 5)')
scene, numIterations = scenario.generate()
print(f'ego has foo = {scene.egoObject.foo}')
```

```
ego has foo = 2.083099362726706
```

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')
scene, _ = scenario.generate()
simulator = NewtonianSimulator()
```

(continues on next page)

(continued from previous page)

```
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.

See also:

If you get exceptions or unexpected behavior when using the API, Scenic provides various debugging features: see *Debugging*.

1.10 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.

1.10.1 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.

Note: If you use Poetry, you can instead run the command **`poetry install -E dev`** to create the virtual environment and install Scenic in it, then **`poetry shell`** to activate the environment.

To find documentation (and code) for specific parts of Scenic's implementation, see our page on *Scenic Internals*.

1.10.2 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`**.

[1] VerifAI's active samplers can be used directly from Scenic when VerifAI is installed. See *`scenic.core.external_params`*.

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.

1.10.3 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).

1.10.4 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.

1.11 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.

1.11.1 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.
pruning	Pruning parts of the sample space which violate requirements.
regions	Objects representing regions in space.
requirements	Support for hard and soft requirements.
scenarios	Scenario and scene objects.
serialization	Utilities to help serialize Scenic 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.
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.
canUnpackDistributions	Whether the function supports iterable unpacking of distributions.
dependencies	Dependencies which must be sampled before this value.
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.
needsSampling	Whether this value requires sampling.
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.
`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

`dependencies`(*thing*)

Dependencies which must be sampled before this value.

`needsSampling`(*thing*)

Whether this value requires sampling.

`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`(*msg*, *loc=None*)

Bases: *`RuntimeParseError`*

Exception indicating illegal conditional control flow depending on a random value.

This includes trying to iterate over a random value, take the length of a random sequence whose length can't be determined statically, 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(**args*, ***kwargs*)

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`

clone()

Construct an independent copy of this Distribution.

Optionally implemented by subclasses.

property isPrimitive

Whether this is a primitive Distribution.

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

Bases: `Distribution`, `Sequence`

Distributions over tuples (or namedtuples, or lists).

`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`(**args*, ***kwargs*)

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`(**args*, ***kwargs*)

Bases: `Distribution`

A placeholder for the iterable unpacking operator * applied to a distribution.

`class MethodDistribution`(**args*, ***kwargs*)

Bases: `Distribution`

Distribution resulting from passing distributions to a method of a fixed object

`distributionMethod`(*method*)

Decorator for wrapping a method so that it can take distributions as arguments.

`class AttributeDistribution`(**args*, ***kwargs*)

Bases: `Distribution`

Distribution resulting from accessing an attribute of a distribution

`static inferType`(*obj*, *attribute*)

Attempt to infer the type of the given attribute.

`class OperatorDistribution`(**args*, ***kwargs*)

Bases: `Distribution`

Distribution resulting from applying an operator to one or more distributions

`static inferType`(*obj*, *operator*, *operands*)

Attempt to infer the result type of the given operator application.

`class MultiplexerDistribution`(**args*, ***kwargs*)

Bases: `Distribution`

Distribution selecting among values based on another distribution.

`class Range`(**args*, ***kwargs*)

Bases: `Distribution`

Uniform distribution over a range

class Normal(**args*, ***kwargs*)

Bases: `Distribution`

Normal distribution

class TruncatedNormal(**args*, ***kwargs*)

Bases: `Normal`

Truncated normal distribution.

class DiscreteRange(**args*, ***kwargs*)

Bases: `Distribution`

Distribution over a range of integers.

class Options(**args*, ***kwargs*)

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

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

`functionForMonitor`	
`isAMonitorName`	
`makeTerminationAction`	
`monitorName`	
`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`	Raised when an invariant is violated when invoking/resuming a behavior.
`PreconditionViolation`	Raised when a precondition is violated when invoking a behavior.
`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.

`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*

Raised when a precondition is violated when invoking a behavior.

exception InvariantViolation(*behavior*, *lineno*)

Bases: *GuardViolation*

Raised when an invariant is violated when invoking/resuming a behavior.

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.
PythonParseError	Parse error occurring during Python parsing or compilation.
RuntimeParseError	A Scenic parse error generated during execution of the translated Python.
ScenicError	An error produced during Scenic compilation, scene generation, or simulation.
ScenicSyntaxError	An error produced by attempting to parse an invalid Scenic program.
TokenParseError	Parse error occurring during token translation.

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 TokenParseError(*tokenOrLine*, *filename*, *message*)

Bases: *ScenicSyntaxError*

Parse error occurring during token translation.

exception PythonParseError(*exc*)

Bases: *ScenicSyntaxError*

Parse error occurring during Python parsing or compilation.

exception ASTParseError(*node*, *message*, *filename*)

Bases: *ScenicSyntaxError*

Parse error occuring during modification of the Python AST.

exception RuntimeParseError(*msg*, *loc=None*)

Bases: *ScenicSyntaxError*

A Scenic parse error generated during execution of the translated Python.

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.

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 = Object at (VerifaiRange(5, 15), 0)
```

This is equivalent to the ordinary Scenic line `ego = 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 = 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 = 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**(**args*, ***kwargs*)

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

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

Bases: *VerifaiParameter*

A *Range* (real interval) sampled by VerifAI.

_defaultValueType

alias of `float`

class VerifaiDiscreteRange(**args*, ***kwargs*)

Bases: *VerifaiParameter*

A *DiscreteRange* (integer interval) sampled by VerifAI.

_defaultValueType

alias of `float`

class VerifaiOptions(**args*, ***kwargs*)

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.

class _RotatedRectangle

mixin providing collision detection for rectangular objects and regions

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

`makeDelayedFunctionCall`	Utility function for creating a lazily-evaluated function call.
`makeDelayedOperatorHandler`	
`needsLazyEvaluation`	Whether the given value requires lazy evaluation.
`requiredProperties`	Set of properties needed to evaluate the given value, if any.
`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*)

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`).

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 for testing purposes.

`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.

requiredProperties(*thing*)

Set of properties needed to evaluate the given value, if any.

needsLazyEvaluation(*thing*)

Whether the given value requires 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*).

Summary of Module Members

Functions

`disableDynamicProxyFor`
`enableDynamicProxyFor`
`setDynamicProxyFor`

Classes

Constructible	Abstract base class for Scenic objects.
HeadingMutator	Mutator adding Gaussian noise to `heading`.
Mutator	An object controlling how the *mutate* statement affects an *Object*.
Object	The Scenic class `Object`.
OrientedPoint	The Scenic class `OrientedPoint`.
Point	The Scenic base class `Point`.
PositionMutator	Mutator adding Gaussian noise to `position`.

Member Details

class Constructible(**args*, *_internal=False*, *_constProps=frozenset({})*, ***kwargs*)

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.

_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(*stddev*)

Bases: *Mutator*

Mutator adding Gaussian noise to `position`. Used by *Point*.

Attributes

stddev (*float*) – standard deviation of noise `class HeadingMutator(`*stddev*`)`

Bases: *`Mutator`*

Mutator adding Gaussian noise to `heading`. Used by *`OrientedPoint`*.

Attributes

stddev (*float*) – standard deviation of noise

`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.
- **visibleDistance** (*float*) – Distance for *`can see`* operator. Default value 50.
- **width** (*float*) – Default value zero (only provided for compatibility with operators that expect an *`Object`*).
- **length** (*float*) – Default value zero.
- **mutationScale** (*float*) – Overall scale of mutations, as set by the *`mutate`* statement. Default value zero (mutations disabled).
- **positionStdDev** (*float*) – Standard deviation of Gaussian noise to add to this object's `position` when mutation is enabled with scale 1. Default value 1.

`property visibleRegion`

The visible region of this object.

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

`class OrientedPoint(<`*specifiers*`>)`

Bases: *`Point`*

The Scenic class `OrientedPoint`.

The default mutator for *`OrientedPoint`* adds Gaussian noise to `heading` with a standard deviation given by the `headingStdDev` property, then applies the mutator for *`Point`*.

Properties

- **heading** (*float; dynamic*) – Heading of the *`OrientedPoint`*. Default value 0 (North).
- **viewAngle** (*float*) – View cone angle for `can see` operator. Default value 2.
- **headingStdDev** (*float*) – Standard deviation of Gaussian noise to add to this object's `heading` when mutation is enabled with scale 1. Default value 5°.

`property visibleRegion`

The visible region of this object.

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`.

`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 1.
- **length** (*float*) – Length of the object, i.e. extent along its Y axis. Default value 1.
- **allowCollisions** (*bool*) – Whether the object is allowed to intersect other objects. Default value `False`.
- **requireVisible** (*bool*) – Whether the object is required to be visible from the `ego` object. Default value `True`.
- **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.
- **cameraOffset** (`Vector`) – Position of the camera for the `can see` operator, relative to the object's `position`. Default `(0, 0)`.
- **speed** (*float; dynamic*) – Speed in dynamic simulations. Default value 0.
- **velocity** (`Vector`; *dynamic*) – Velocity in dynamic simulations. Default value is the velocity determined by `self.speed` and `self.heading`.
- **angularSpeed** (*float; dynamic*) – Angular speed in dynamic simulations. Default value 0.
- **behavior** – Behavior for dynamic agents, if any (see *Dynamic Scenarios*). Default value `None`.

`startDynamicSimulation()`

Hook called at the beginning of each dynamic simulation.

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

`property visibleRegion`

The visible region of this object.

The visible region of an `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.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.
isMethodCall	Match calls to a given method, taking into account distribution decorators.
matchInRegion	Match uniform samples from a *Region*, returning the Region if any.
matchPolygonalField	Match headings 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.

matchInRegion(*position*)

Match uniform samples from a *Region*, returning the Region if any.

matchPolygonalField(*heading*, *position*)

Match headings defined by a *PolygonalVectorField* at the given position.

Matches headings 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 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.

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.

Summary of Module Members

Module Attributes

everywhere	A *Region* containing all points.
nowhere	A *Region* containing no points.

Functions

orientationFor	
regionFromShapelyObject	Build a *Region* from Shapely geometry.
toPolygon	

Classes

AllRegion	Region consisting of all space.
CircularRegion	A circular region with a possibly-random center and radius.
DifferenceRegion	
EmptyRegion	Region containing no points.
GridRegion	A Region given by an obstacle grid.
IntersectionRegion	
PointInRegionDistribution	Uniform distribution over points in a Region.
PointSetRegion	Region consisting of a set of discrete points.
PolygonalRegion	Region given by one or more polygons (possibly with holes).
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	Abstract class for regions.
SectorRegion	A sector of a *CircularRegion*.

Member Details

regionFromShapelyObject(*obj*, *orientation=None*)

Build a *Region* from Shapely geometry.

class **PointInRegionDistribution**(**args*, ***kwargs*)

Bases: *VectorDistribution*

Uniform distribution over points in a Region.

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

Bases: *Samplable*

Abstract class for regions.

intersect(*other*)

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

intersects(*other*)

Check if this *Region* intersects another.

Return type
bool

difference(*other*)

Get a *Region* representing the difference of this one and another.

Return type
Region

union(*other*)

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

Not supported by all region types.

Return type
Region

static **uniformPointIn**(*region*)

Get a uniform *Distribution* over points in a *Region*.

uniformPointInner()

Do the actual random sampling. Implemented by subclasses.

containsPoint(*point*)

Check if the *Region* contains a point. Implemented by subclasses.

Return type
bool

containsObject(*obj*)

Check if the *Region* contains an *Object*.

The default implementation assumes the *Region* is convex; subclasses must override the method if this is not the case.

Return type
bool

distanceTo(*point*)

Distance to this region from a given point.

Not supported by all region types.

Return type
float

getAABB()

Axis-aligned bounding box for this *Region*. Implemented by some subclasses.

orient(*vec*)

Orient the given vector along the region's orientation, if any.

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.

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

Bases: `Region`

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: `Region`

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: `Region`

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 heading 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 heading 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 PolygonalRegion`(*points=None*, *polygon=None*, *orientation=None*, *name=None*)

Bases: *`Region`*

Region given by one or more polygons (possibly with holes).

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).
- **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 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 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

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

BoundRequirement	
CompiledRequirement	
DynamicRequirement	
PendingRequirement	
RequirementType	An enumeration.

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.

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.
- **egoObject** (`Object`) – The `ego` object.
- **params** (*dict*) – Dictionary mapping the name of each global parameter to its value.
- **workspace** (`Workspace`) – Workspace for the scenario.

`dumpAsScenicCode`(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Dump Scenic code reproducing this scene to the given stream.

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`).

`show`(*zoom=None*, *block=True*)

Render a 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.

`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).

scenic.core.serialization

Utilities to help serialize Scenic objects.

Summary of Module Members

Functions

dumpAsScenicCode	
scenicToJSON	Utility function to help serialize Scenic objects to JSON.

Member Details

scenicToJSON(*obj*)

Utility function to help serialize Scenic objects to JSON.

Suitable for passing as the `default` argument to `json.dump`.

scenic.core.simulators

Interface between Scenic and simulators.

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 nothing, for 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.
Simulation	A single simulation run, possibly in progress.
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

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 RejectSimulationException

Bases: `Exception`

Exception indicating a requirement was violated at runtime.

`class Simulator`

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`*.

`simulate`(*scene*, *maxSteps=None*, *maxIterations=100*, *verbosity=None*, *raiseGuardViolations=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.
- **`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).

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.

`createSimulation`(*scene*, *verbosity=0*)

Create a *`Simulation`* from a Scenic scene.

This should be overridden by subclasses to return instances of their own specialized subclass of *`Simulation`*.

`destroy`()

Clean up as needed when shutting down the simulator interface.

`class Simulation`(*scene*, *timestep=1*, *verbosity=0*)

A single simulation run, possibly in progress.

These objects are not manipulated manually, but are created and executed by a *`Simulator`*. Simulator interfaces should subclass this class, overriding abstract methods like *`createObjectInSimulator`*, *`step`*, and *`getProperties`* to call the appropriate simulator APIs.

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 attributes of this class are primarily for internal use.

`run`(*maxSteps*)

Run the simulation.

Throws a RejectSimulationException if a requirement is violated.

`createObject`(*obj*)

Dynamically create an object.

`createObjectInSimulator`(*obj*)

Create the given object in the simulator.

Implemented by subclasses, and called through `createObject`. Should raise SimulationCreationError if creating the object fails.

`scheduleForAgents`()

Return the order for the agents to run in the next time step.

`actionsAreCompatible`(*agent*, *actions*)

Check whether the given actions can be taken simultaneously by an agent.

The default is to have all actions compatible with each other and all agents. Subclasses should override this method as appropriate.

`executeActions`(*allActions*)

Execute the actions selected by the agents.

Note that `allActions` is an OrderedDict, as the order of actions may matter.

`step`()

Run the simulation for one step and return the next trajectory element.

`updateObjects`()

Update the positions and other properties of objects from the simulation.

`getProperties`(*obj*, *properties*)

Read the values of the given properties of the object from the simulation.

`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.

`destroy`()

Perform any cleanup necessary to reset the simulator after a simulation.

class DummySimulator(*timestep=1*)

Bases: *Simulator*

Simulator which does nothing, for debugging purposes.

class DummySimulation(*scene*, *timestep=1*, *verbosity=0*)

Bases: *Simulation*

Minimal *Simulation* subclass for *DummySimulator*.

class Action

An action which can be taken by an agent for one step of a simulation.

class EndSimulationAction(*line*)

Bases: *Action*

Special action indicating it is time to end the simulation.

Only for internal use.

class EndScenarioAction(*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's *compose* block finished executing.

simulationTerminationCondition = 'a simulation termination condition was met'

A user-specified termination condition was met.

terminatedByMonitor = 'a monitor terminated the simulation'

A monitor used *terminate* to end the simulation.

terminatedByBehavior = 'a behavior terminated the simulation'

A dynamic behavior used *terminate* 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

`PropertyDefault`	A default value, possibly with dependencies.
`Specifier`	Specifier providing a value for a property given dependencies.

Member Details

class Specifier(*prop*, *value*, *deps=None*, *optionals={}*)

Specifier providing a value for a property given dependencies.

Any optionally-specified properties are evaluated as attributes of the primary value.

applyTo(*obj*, *optionals*)

Apply specifier to an object, including the specified optional properties.

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
  - 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.
`coerceToBehavior`	
`coerceToFloat`	
`coerceToHeading`	
`coerceToVector`	
`evaluateRequiringEqualTypes`	Evaluate the func, assuming thingA and thingB have the same type.
`isA`	Does this evaluate to a member of the given Scenic type?
`toHeading`	Convert something to a heading, 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.
`unifyingType`	Most specific type unifying the given types.

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*)

Does this evaluate to a member of the given Scenic type?

unifyingType(*opts*)

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

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

`RuntimeParseError` – 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

`RuntimeParseError` – 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.

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

`RuntimeParseError` – 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.

scenic.core.utils

Assorted utility functions.

Summary of Module Members

Functions

`alarm`	
`argsToString`	
`cached`	Decorator for making a method with no arguments cache its result
`cached_property`	
`get_type_args`	Version of `typing.get_args` supporting Python 3.7.
`get_type_origin`	Version of `typing.get_origin` supporting Python 3.7.

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

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.

get_type_origin(*tp*)

Version of `typing.get_origin` supporting Python 3.7.

get_type_args(*tp*)

Version of `typing.get_args` supporting Python 3.7.

scenic.core.vectors

Scenic vectors and vector fields.

Summary of Module Members

Functions

`makeVectorOperatorHandler`	
scalarOperator	Decorator for vector operators that yield scalars.
vectorDistributionMethod	Decorator for methods that produce vectors.
vectorOperator	Decorator for vector operators that yield vectors.

Classes

`OrientedVector`	
PiecewiseVectorField	A vector field defined by patching together several regions.
PolygonalVectorField	A piecewise-constant vector field defined over polygonal cells.
Vector	A 2D vector, whose coordinates can be distributions.
VectorDistribution	A distribution over Vectors.
VectorField	A vector field, providing a heading at every point.
VectorMethodDistribution	Vector version of MethodDistribution.
VectorOperatorDistribution	Vector version of OperatorDistribution.

Member Details

class VectorDistribution(**args*, ***kwargs*)

Bases: *Distribution*

A distribution over Vectors.

_defaultValueType

alias of *Vector*

class VectorOperatorDistribution(**args*, ***kwargs*)

Bases: *VectorDistribution*

Vector version of OperatorDistribution.

class VectorMethodDistribution(**args*, ***kwargs*)

Bases: *VectorDistribution*

Vector version of MethodDistribution.

`scalarOperator`(*method*)

Decorator for vector operators that yield scalars.

`vectorOperator`(*method*)

Decorator for vector operators that yield vectors.

`vectorDistributionMethod`(*method*)

Decorator for methods that produce vectors. See distributionMethod.

`class Vector`(*x*, *y*)

Bases: *`Samplable`*, `Sequence`

A 2D vector, whose coordinates can be distributions.

`rotatedBy`(*angle*)

Return a vector equal to this one rotated counterclockwise by the given angle.

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 a heading 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.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*).

show(*plt*)

Render a schematic of the workspace 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:

[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.

```
$ 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
```

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`

Classes

`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

class FollowLaneBehavior(**args*, ***kwargs*)

Bases: *`Behavior`*

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.

class FollowTrajectoryBehavior(**args*, ***kwargs*)

Bases: *`Behavior`*

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.

class LaneChangeBehavior(**args*, ***kwargs*)

Bases: *`Behavior`*

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

`class TurnBehavior(*args, **kwargs)`

Bases: `Behavior`

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.

`class WalkForwardBehavior(*args, **kwargs)`

Bases: `Behavior`

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

`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

class Car(<*specifiers*>)

Bases: *`Vehicle`*

A car.

class DrivingObject(<*specifiers*>)

Bases: *`Object`*

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
Object

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.

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`.

`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) in front of the vehicle, (2) 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, Point, 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[float]

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[float]

`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*, *timestep=1*, *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.

Summary of Module Members

Classes

AutopilotBehavior	Behavior causing a vehicle to use CARLA's built-in autopilot.
CrossingBehavior	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.
`WalkBehavior`	
`WalkForwardBehavior`	

Member Details

class AutopilotBehavior(**args*, ***kwargs*)

Bases: *Behavior*

Behavior causing a vehicle to use CARLA's built-in autopilot.

class CrossingBehavior(**args*, ***kwargs*)

Bases: *Behavior*

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

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 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 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 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.

`class Vehicle(<specifiers>)`

Bases: `Vehicle`, `CarlaActor`, `Steers`

Abstract class for steerable vehicles.

`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).

`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.
workspace([workspace])	Function implementing loads and stores to the 'workspace' pseudo-variable.

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

`class Bus(<specifiers>)`

Bases: *Car*

Convenience subclass for buses.

`class Car(<specifiers>)`

Bases: *Object*

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 Compact(<specifiers>)`

Bases: *Car*

Convenience subclass for compact cars.

`class EgoCar(<specifiers>)`

Bases: *Car*

Convenience subclass with defaults for ego 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.

Summary of Module Members

Classes

`DriveTo`
`FollowWaypoints`
`WalkBehavior`

Member Details

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.

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*, *verbosity=0*)

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: *`Object`*

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*, *timestep=0.1*, *render=False*)

Bases: *DrivingSimulator*

Implementation of *Simulator* for the Newtonian simulator.

Parameters

- **network** (Network) – road network to display in the background, if any.
- **timestep** (*float*) – time step to use.
- **render** (*bool*) – whether to render the simulation in a window.

class NewtonianSimulation(*scene*, *network*, *timestep*, *verbosity=0*, *render=False*)

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

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.
mars	World model for a simple Mars rover example 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.mars

World model for a simple Mars rover example in Webots.

model	Scenic model for Mars rover scenarios in Webots.

scenic.simulators.webots.mars.model

Scenic model for Mars rover scenarios in Webots.

Summary of Module Members

Classes

BigRock	Large rock.
Debris	Abstract class for debris scattered randomly in the workspace.
Goal	Flag indicating the goal location.
Pipe	Pipe with variable length.
Rock	Small rock.
Rover	Mars rover.

Member Details

class BigRock(*<specifiers>*)

Bases: *Debris*

Large rock.

class Debris(*<specifiers>*)

Bases: *WebotsObject*

Abstract class for debris scattered randomly in the workspace.

class Goal(*<specifiers>*)

Bases: *WebotsObject*

Flag indicating the goal location.

class Pipe(*<specifiers>*)

Bases: *Debris*

Pipe with variable length.

class Rock(*<specifiers>*)

Bases: *Debris*

Small rock.

class Rover(*<specifiers>*)

Bases: *WebotsObject*

Mars rover.

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

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 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 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).

`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 WebotsObject`(*<specifiers>*)

Bases: `Object`

Abstract class for Webots objects.

There are two ways to specify which Webots node this object corresponds to (which must already exist in the world loaded into Webots): most simply, you can set the `webotsName` property to the DEF name of the Webots node. For convenience when working with many objects of the same type, you can instead set the `webotsType` property to a prefix like 'ROCK': the interface will then search for nodes called 'ROCK_0', 'ROCK_1', etc.

Also defines the `elevation` property as a standard way to access the "up" component of an object's position, since the Scenic built-in property `position` is only 2D. If `elevation` is set to `None`, it will be updated to the object's "up" coordinate in Webots when the simulation starts.

Properties

- **elevation** (*float or None; dynamic*) – default `None` (see above).
- **requireVisible** (*bool*) – Default value `False` (overriding the default from `Object`).
- **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** (*float*) – Offset to add when computing the object's rotation in Webots; for objects whose front is not aligned with the Webots North axis.

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*)

Parameters

- **name** (*str*) –
- **width** (*float*) –
- **length** (*float*) –

scenic.simulators.webots.road.interface

Python library supporting the main Scenic module.

Summary of Module Members

Functions

polygonWithPoints	
regionWithPolygons	

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

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

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*, *verbosity=0*, *coordinateSystem=<scenic.simulators.webots.utils.WebotsCoordinateSystem object>*)

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(...)`.

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.

Functions

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

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*, *elevation=0*)

Convert a Scenic position to a Webots position.

rotationToScenic(*rot*, *tolerance2D=None*)

Convert a Webots rotation vector to a Scenic heading.

Assumes the object lies in the Webots horizontal plane, with a rotation axis close to the up axis. If `tolerance2D` is given, returns `None` if the orientation of the object is not sufficiently close to being 2D.

rotationFromScenic(*heading*)

Convert a Scenic heading to a Webots rotation vector.

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.

webotsToScenicPosition(*pos*)

Convert a Webots position to a Scenic position.

Drops the Webots Y coordinate.

Deprecated since version 2.1.0: Use *WebotsCoordinateSystem.positionToScenic* instead, dropping the third coordinate.

scenicToWebotsPosition(*pos*, *y=0*, *coordinateSystem='ENU'*)

Convert a Scenic position to a Webots position.

Deprecated since version 2.1.0: Use *WebotsCoordinateSystem.positionFromScenic* 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.rotationToScenic* instead.

scenicToWebotsRotation(*heading*)

Convert a Scenic heading to a Webots rotation vector.

Deprecated since version 2.1.0: Use *WebotsCoordinateSystem.rotationFromScenic* instead.

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.

pygment	Pygments lexer and style for Scenic.
relations	Extracting relations (for later pruning) from the syntax of requirements.
scenic2to3	Converter from Scenic 2.x to 3.0.
translator	Translator turning Scenic programs into Scenario objects.
veneer	Python implementations of Scenic language constructs.

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.
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.
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 `ScenicSpecifierLexer`(**args*, ***kwds*)

Bases: *ScenicSnippetLexer*

Further variant lexer for specifiers 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.

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.scenic2to3

Converter from Scenic 2.x to 3.0.

This module is designed to be invoked from the command line. Run the following command to see the available options:

```
$ python -m scenic.syntax.scenic2to3 -h
```

Summary of Module Members

Classes

Scenic2to3

Member Details

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 so that Scenic modules can be imported using the `import` statement. This is primarily for the translator's own use, but you could import Scenic modules from Python to inspect them.[1] Because Scenic uses Python's import system, the latter's rules for finding modules apply, including the handling of packages.

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

[1] Note however that care must be taken when importing Scenic modules which will later be used when compiling multiple Scenic scenarios. Because Python caches modules, there is the possibility of one version of a Scenic module persisting even when it should be recompiled during the compilation of another module that imports it. Scenic handles the most common case, that of Scenic modules which refer to other Scenic modules at the top level; but it is not practical to catch all possible cases. In particular, importing a Python package which contains Scenic modules as submodules and then later compiling those modules more than once within the same Python process may lead to errors or unexpected behavior. See the **cacheImports** argument of `scenarioFromFile`.

Functions

compileStream	Compile a stream of Scenic code and execute it in a namespace.
compileTopLevelStream	
compileTranslatedTree	
constructScenarioFrom	Build a Scenario object from an executed Scenic module.
executeCodeIn	Execute the final translated Python code in the given namespace.
findConstructorsIn	Find all constructors (Scenic classes) defined in a namespace.
functionForStatement	
gatherBehaviorNamespacesFrom	Gather any global namespaces which could be referred to by behaviors.
nameForStatement	
parseTranslatedSource	
partitionByImports	Partition the tokens into blocks ending with import statements.
peek	
purgeModulesUnsafeToCache	Uncache loaded modules which should not be kept after compilation.
scenarioFromFile	Compile a Scenic file into a *Scenario*.
scenarioFromStream	Compile a stream of Scenic code 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.
translateParseTree	Modify the Python AST to produce the desired Scenic semantics.

Classes

`ASTSurgeon`	
`AttributeFinder`	Utility class for finding all referenced attributes of a given name.
`Constructor`	
`InfixOp`	
`LocalFinder`	Utility class for finding all local variables of a code block.
`ModifierInfo`	
`Peekable`	Utility class to allow iterator lookahead.
`ScenicFileFinder`	
`ScenicLoader`	
`ScenicModule`	
`TokenTranslator`	Translates a Scenic token stream into valid Python syntax.

Member Details

`scenarioFromString`(*string*, *params={}*, *model=None*, *scenario=None*, *filename='<string>'*, *cacheImports=False*)

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*, *cacheImports=False*)

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.
- **`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.

Returns

A `Scenario` object representing the Scenic scenario.

`scenarioFromStream`(*stream*, *params={}*, *model=None*, *scenario=None*, *filename='<stream>'*, *path=None*, *cacheImports=False*)

Compile a stream of Scenic code into a `Scenario`.

`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*, *params={}*, *model=None*, *filename='<stream>'*, *dumpScenic3=False*)

Compile a stream of Scenic code and execute it in a namespace.

The compilation procedure consists of the following main steps:

1. Tokenize the input using the Python tokenizer.
2. Partition the tokens into blocks separated by import statements. This is done by the `partitionByImports` function.
3. Translate Scenic constructions into valid Python syntax. This is done by the `TokenTranslator`.
4. Parse the resulting Python code into an AST using the Python parser.
5. Modify the AST to achieve the desired semantics for Scenic. This is done by the `translateParseTree` function.
6. Compile and execute the modified AST.
7. After executing all blocks, extract the global state (e.g. objects). This is done by the `storeScenarioStateIn` function.

`class Constructor`(*name*, *bases*)

Bases: `tuple`

`_asdict`()

Return a new dict which maps field names to their values.

`classmethod _make`(*iterable*)

Make a new Constructor object from a sequence or iterable

`_replace`(***kwds*)

Return a new Constructor object replacing specified fields with new values

`bases`

Alias for field number 1

`name`

Alias for field number 0

class ModifierInfo(*name*, *terminators*, *contexts*)

Bases: `NamedTuple`

Parameters

- **name** (*str*) –
- **terminators** (*Tuple[str]*) –
- **contexts** (*Optional[Tuple[str]]*) –

name: str

Alias for field number 0

terminators: Tuple[str]

Alias for field number 1

contexts: Optional[Tuple[str]]

Alias for field number 2

_asdict()

Return a new dict which maps field names to their values.

classmethod _make(*iterable*)

Make a new ModifierInfo object from a sequence or iterable

_replace(***kwds*)

Return a new ModifierInfo object replacing specified fields with new values

class InfixOp(*syntax*, *implementation*, *arity*, *token*, *node*, *contexts*)

Bases: `NamedTuple`

Parameters

- **syntax** (*str*) –
- **implementation** (*Optional[str]*) –
- **arity** (*int*) –
- **token** (*Tuple[int, str]*) –
- **node** (*AST*) –
- **contexts** (*Optional[Tuple[str]]*) –

syntax: str

Alias for field number 0

implementation: Optional[str]

Alias for field number 1

arity: int

Alias for field number 2

token: Tuple[int, str]

Alias for field number 3

node: AST

Alias for field number 4

contexts: Optional[Tuple[str]]

Alias for field number 5

_asdict()

Return a new dict which maps field names to their values.

classmethod _make(*iterable*)

Make a new InfixOp object from a sequence or iterable

_replace(***kwds*)

Return a new InfixOp object replacing specified fields with new values

partitionByImports(*tokens*)

Partition the tokens into blocks ending with import statements.

We avoid splitting top-level try-except statements, to allow the pattern of trying to import an optional module and catching an ImportError. If someone tries to define objects inside such a statement, woe unto them.

findConstructorsIn(*namespace*)

Find all constructors (Scenic classes) defined in a namespace.

class Peekable(*gen*)

Utility class to allow iterator lookahead.

class TokenTranslator(*constructors=()*, *filename='<unknown>'*)

Translates a Scenic token stream into valid Python syntax.

This is a stateful process because constructor (Scenic class) definitions change the way subsequent code is parsed.

translate(*tokens*, *dumpScenic3=False*)

Do the actual translation of the token stream.

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.

translateParseTree(*tree*, *constructors*, *filename*)

Modify the Python AST to produce the desired Scenic semantics.

executeCodeIn(*code*, *namespace*)

Execute the final translated Python code in the given namespace.

storeScenarioStateIn(*namespace*, *requirementSyntax*)

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.

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

Ahead	The ahead of *X* by *Y* polymorphic specifier.
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.
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.
Beyond	The beyond *X* by *Y* from *Z* polymorphic specifier.
CanSee	The *X* can see *Y* polymorphic operator.
DistanceFrom	The distance from *X* to *Y* polymorphic operator.
DistancePast	The distance past <**vector**> of <**oriented point**> operator.
Facing	The facing *X* polymorphic specifier.
FacingToward	The facing toward <**vector**> specifier.
FieldAt	The <**VectorField**> 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.
In	The in/on <**region**> specifier.
Left	The left of <**object**> operator.
LeftSpec	The left of *X* by *Y* polymorphic specifier.
NotVisible	The not visible <**region**> operator.
NotVisibleFrom	The not visible from <**Point**> specifier.
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.
RelativeHeading	The relative heading of <**heading**> [from <**heading**>] operator.

continues on next page

Table 2 – continued from previous page

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.
Visible	The visible <*region*> operator.
VisibleFrom	The visible from <*Point*> specifier.
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.
ego	Function implementing loads and stores to the 'ego' pseudo-variable.
endScenario	
endSimulation	
executeInBehavior	
executeInGuard	
executeInRequirement	
executeInScenario	
filter	
finishScenarioSetup	
globalParameters	
in_initial_scenario	
instantiateSimulator	
isActive	Are we in the middle of compiling a Scenic module?
leftSpecHelper	
localPath	Convert a path relative to the calling Scenic file into an absolute path.
makeRequirement	

continues on next page

Table 2 – continued from previous page

model	
mutate	Function implementing the mutate statement.
override	
param	Function implementing the param statement.
prepareScenario	
record	
record_final	
record_initial	
registerDynamicScenarioClass	
registerExternalParameter	Register a parameter whose value is given by an external sampler.
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.
resample	The built-in resample function.
simulation	Get the currently-running *Simulation*.
simulationInProgress	
simulator	
startScenario	
str	
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(**quotedParams*, ***params*)

Function implementing the param statement.

mutate(**objects*)

Function implementing the mutate statement.

verbosePrint(**objects*, *level=1*, *indent=True*, *sep=' '*, *end='\n'*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *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.

`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.

`require_always`(*reqID*, *req*, *line*, *name*)

Function implementing the 'require always' statement.

`require_eventually`(*reqID*, *req*, *line*, *name*)

Function implementing the 'require eventually' statement.

`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.

`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.

AngleTo(*X*)

The `angle to <vector>` operator (using the position of ego as the reference).

AngleFrom(*X*, *Y*)

The `angle from <vector> to <vector>` operator.

Follow(*F*, *X*, *D*)

The `follow <field> from <vector> for <number>` operator.

FieldAt(*X*, *Y*)

The `<VectorField> 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>
```

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 <object>
<point> can see <vector>
```

class Vector(*x*, *y*)

Bases: *Samplable*, Sequence

A 2D vector, whose coordinates can be distributions.

rotatedBy(*angle*)

Return a vector equal to this one rotated counterclockwise by the given angle.

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 a heading 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 Region`(*name*, **dependencies*, *orientation=None*)

Bases: *Samplable*

Abstract class for regions.

`intersect`(*other*)

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

`intersects`(*other*)

Check if this *Region* intersects another.

Return type
bool

`difference`(*other*)

Get a *Region* representing the difference of this one and another.

Return type
Region

`union`(*other*)

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

Not supported by all region types.

Return type
Region

`static uniformPointIn`(*region*)

Get a uniform *Distribution* over points in a *Region*.

`uniformPointInner`()

Do the actual random sampling. Implemented by subclasses.

`containsPoint`(*point*)

Check if the *Region* contains a point. Implemented by subclasses.

Return type
bool

`containsObject`(*obj*)

Check if the *Region* contains an *Object*.

The default implementation assumes the *Region* is convex; subclasses must override the method if this is not the case.

Return type
bool

`distanceTo`(*point*)

Distance to this region from a given point.

Not supported by all region types.

Return type
float

`getAABB`()

Axis-aligned bounding box for this *Region*. Implemented by some subclasses.

`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: *Region*

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: *Region*

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: *Region*

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*, *orientation=None*, *name=None*)

Bases: *Region*

Region given by one or more polygons (possibly with holes).

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).
- **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 heading 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 heading 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*).

show(*plt*)

Render a schematic of the workspace 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**(**args*, ***kwargs*)

Bases: *Distribution*

Uniform distribution over a range

class **DiscreteRange**(**args*, ***kwargs*)

Bases: *Distribution*

Distribution over a range of integers.

class **Options**(**args*, ***kwargs*)

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

Bases: *Distribution*

Normal distribution

class **TruncatedNormal**(**args*, ***kwargs*)

Bases: *Normal*

Truncated normal distribution.

class VerifaiParameter(**args*, ***kwargs*)

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

Bases: `VerifaiParameter`

A `Range` (real interval) sampled by VerifAI.

_defaultValueType

alias of `float`

class VerifaiDiscreteRange(**args*, ***kwargs*)

Bases: `VerifaiParameter`

A `DiscreteRange` (integer interval) sampled by VerifAI.

_defaultValueType

alias of `float`

class VerifaiOptions(**args*, ***kwargs*)

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.
- **visibleDistance** (*float*) – Distance for `can see` operator. Default value 50.
- **width** (*float*) – Default value zero (only provided for compatibility with operators that expect an `Object`).
- **length** (*float*) – Default value zero.
- **mutationScale** (*float*) – Overall scale of mutations, as set by the `mutate` statement. Default value zero (mutations disabled).
- **positionStdDev** (*float*) – Standard deviation of Gaussian noise to add to this object's `position` when mutation is enabled with scale 1. Default value 1.

property visibleRegion

The visible region of this object.

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

`class OrientedPoint(<specifiers>)`

Bases: `Point`

The Scenic class `OrientedPoint`.

The default mutator for `OrientedPoint` adds Gaussian noise to `heading` with a standard deviation given by the `headingStdDev` property, then applies the mutator for `Point`.

Properties

- **heading** (*float; dynamic*) – Heading of the `OrientedPoint`. Default value 0 (North).
- **viewAngle** (*float*) – View cone angle for `can see` operator. Default value 2.
- **headingStdDev** (*float*) – Standard deviation of Gaussian noise to add to this object's `heading` when mutation is enabled with scale 1. Default value 5°.

`property visibleRegion`

The visible region of this object.

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`.

`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 1.
- **length** (*float*) – Length of the object, i.e. extent along its Y axis. Default value 1.
- **allowCollisions** (*bool*) – Whether the object is allowed to intersect other objects. Default value `False`.
- **requireVisible** (*bool*) – Whether the object is required to be visible from the `ego` object. Default value `True`.
- **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.
- **cameraOffset** (`Vector`) – Position of the camera for the `can see` operator, relative to the object's `position`. Default `(0, 0)`.
- **speed** (*float; dynamic*) – Speed in dynamic simulations. Default value 0.
- **velocity** (`Vector`; *dynamic*) – Velocity in dynamic simulations. Default value is the velocity determined by `self.speed` and `self.heading`.
- **angularSpeed** (*float; dynamic*) – Angular speed in dynamic simulations. Default value 0.
- **behavior** – Behavior for dynamic agents, if any (see *Dynamic Scenarios*). Default value `None`.

`startDynamicSimulation()`

Hook called at the beginning of each dynamic simulation.

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

property visibleRegion

The visible region of this object.

The visible region of an `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).

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/on <region>` specifier.

Specifies `position`, with no dependencies. Optionally specifies `heading` if the given `Region` has a preferred orientation.

Beyond(*pos*, *offset*, *fromPt=None*)

The `beyond X by Y from Z` polymorphic specifier.

Specifies `position`, 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 `position`, with no dependencies.

This uses the given object's `visibleRegion` property, and so correctly handles the view regions of Points, OrientedPoints, and Objects.

VisibleSpec()

The `visible` specifier (equivalent to `visible from ego`).

Specifies `position`, with no dependencies.

NotVisibleSpec()

The `not visible` specifier (equivalent to `not visible from ego`).

Specifies `position`, depending on `regionContainedIn`.

OffsetBy(*offset*)

The `offset by <vector>` specifier.

Specifies `position`, with no dependencies.

OffsetAlongSpec(*direction*, *offset*)

The `offset along X by Y` polymorphic specifier.

Specifies `position`, with no dependencies.

Allowed forms:

```
offset along <heading> by <vector>
offset along <field> by <vector>
```

Facing(*heading*)

The `facing X` polymorphic specifier.

Specifies `heading`, with dependencies depending on the form:

```
facing <number>         # no dependencies;
facing <field>          # depends on 'position'
```

FacingToward(*pos*)

The `facing toward <vector>` specifier.

Specifies `heading`, depending on `position`.

ApparentlyFacing(*heading*, *fromPt=None*)

The `apparently facing <heading> [from <vector>]` specifier.

Specifies `heading`, depending on `position`.

If the `from <vector>` is omitted, the position of ego is used.

LeftSpec(*pos*, *dist=0*)

The `left of X by Y` polymorphic specifier.

Specifies `position`, depending on `width`. See other dependencies below.

Allowed forms:

```
left of <oriented point> [by <scalar/vector>] # optionally specifies 'heading';
left of <vector> [by <scalar/vector>]  # depends on 'heading'.
```

If the `by <scalar/vector>` is omitted, zero is used.

RightSpec(*pos*, *dist=0*)

The `right of X by Y` polymorphic specifier.

Specifies `position`, depending on `width`. See other dependencies below.

Allowed forms:

```
right of <oriented point> [by <scalar/vector>] # optionally specifies 'heading';
right of <vector> [by <scalar/vector>]  # depends on 'heading'.
```

If the `by <scalar/vector>` is omitted, zero is used.

Ahead(*pos*, *dist=0*)

The `ahead of X by Y` polymorphic specifier.

Specifies `position`, depending on `length`. See other dependencies below.

Allowed forms:

```
ahead of <oriented point> [by <scalar/vector>]   # optionally specifies 'heading';
ahead of <vector> [by <scalar/vector>]   # depends on 'heading'.
```

If the `by <scalar/vector>` is omitted, zero is used.

`Behind`(*pos*, *dist=0*)

The `behind X by Y` polymorphic specifier.

Specifies `position`, depending on `length`. See other dependencies below.

Allowed forms:

```
behind <oriented point> [by <scalar/vector>]   # optionally specifies 'heading';
behind <vector> [by <scalar/vector>]   # depends on 'heading'.
```

If the `by <scalar/vector>` is omitted, zero is used.

`Following`(*field*, *dist*, *fromPt=None*)

The `following F from X for D` specifier.

Specifies `position`, and optionally `heading`, 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`*

Raised when a precondition is violated when invoking a behavior.

`exception InvariantViolation`(*behavior*, *lineno*)

Bases: *`GuardViolation`*

Raised when an invariant is violated when invoking/resuming a behavior.

`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.

`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.

The `scenic` module itself provides the top-level API for using Scenic: see *Using Scenic Programmatically*.

1.11.2 How Scenic is Compiled

Note: This section describes the current compiler, which goes through various convolutions in order to make use of the Python parser as much as possible. This has the advantage of allowing almost all Python constructs to be used in Scenic, but the disadvantage of sometimes producing cryptic error messages. We are in the process of writing a native parser for Scenic which will replace Phases 1-4 below.

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.

Phase 1: Import Handling

In this phase the program is segmented into blocks ending with *`import`* statements, which will then be compiled separately. This is done so that if a Scenic module defining new Scenic classes is imported, subsequent parsing can properly detect instantiations of those classes. For more details on the various rules for import statements in Scenic, see *`import`*.

Phase 2: Token Translation

In this phase the Scenic program is tokenized using the Python tokenizer. Various Scenic constructs yield sequences of tokens which are meaningless to the Python parser, which we will use to parse the code in the next phase. To remedy this, all Scenic-specific token sequences are translated into new sequences that will pass through the Python parser if they are syntactically-valid Scenic.

Phase 3: Python Parser

In this phase we parse the token stream resulting from Phase 2 using the Python parser. If the tokens do not parse properly, they come from a malformed Scenic program, and an error will be raised here. Unfortunately, since the tokens being parsed are not the original Scenic tokens and likely not semantically-valid Python either, the error messages can be confusing.

Phase 4: Python AST Modification

In this phase we walk the AST generated in Phase 3 to transform it into a Python AST with equivalent semantics to the original Scenic program. This undoes the translation performed in Phase 2 to get syntactically-valid Python, restoring the intended semantics of the program.

Phase 5: AST Compilation

Compile the Python AST down to a Python `code` object.

Phase 6: Python Execution

In this phase the Python `code` object compiled in Phase 5 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.

Phase 7: Scenario Construction

In this phase the various pieces of the internal representation of the program resulting from Phase 6 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] 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.

[2] Although there are some syntax errors which are currently not detected until those stages.

1.12 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.

1.12.1 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`.

1.12.2 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.

1.13 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*

1.13.1 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.

1.13.2 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 0.9.13) 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. For CARLA 0.9.12 and onward, you can simply run **`pip install carla`** (unless you built CARLA from source; see detailed instructions here).

Note: For older versions of CARLA, you'll need to install its Python API from the `.egg` by executing the following command:

```
$ 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. Instead, check that the `carla` package was correctly installed by running **`pip show carla`**.

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*.

1.13.3 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.

1.13.4 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*.

1.13.5 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 several more specialized world models for use with Webots:

- A model for the Mars rover example used in our PLDI paper. This model is extremely simple and might be a good baseline for developing your own model. See the examples in `examples/webots/mars` (including a dynamic version of the scenario from the paper) and the documentation of *`scenic.simulators.webots.mars`* for details.
- 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`*.
- A more specific model for traffic scenarios at intersections, using guideways from the Intelligent Intersections Toolkit. See the examples in `examples/webots/guideways` and the documentation of *`scenic.simulators.webots.guideways`* for details.

Note: The last two models above, and the example `.wbt` files for them, were written for the R2018 version of Webots. Relatively minor changes would be required to make them 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!

1.13.6 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.

1.14 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.

1.14.1 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.

1.14.2 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 :file:`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. The `scenic.simulators.webots.mars` model is particularly simple and would be a good place to start. For a more complex example, see the `scenic.simulators.lgsvl` model, which specializes the simulator-agnostic model provided by the *Driving Domain* (in `scenic.domains.driving.model`).

1.15 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.

1.15.1 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 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.)

1.16 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.

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.

[GR83] Goldberg and Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983. [PDF]

PYTHON MODULE INDEX

INDEX