Scenic

Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu

May 20, 2020

CONTENTS

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 was designed and implemented by Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. For a description of the language and some of its applications, see our PLDI 2019 paper; a more in-depth discussion is in Chapters 5 and 8 of this thesis.

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 3.6** or newer. You can install Scenic from PyPI by simply running:

```
pip install scenic
```

Alternatively, you can download or clone the Scenic repository, which contains examples we'll use below. Install Poetry and then run:

```
poetry install
```

This will install Scenic into your current virtual environment (or create a new one if needed). If you will be developing Scenic, add the `-E dev` option when invoking Poetry.

Either of the options above should install all of the dependencies which are required to run Scenic. Scenarios using the *`scenic.simulators.webots.guideways`* model also require the `pyproj` package, and will prompt you if you don't have it.

Note: For Windows, we recommend using bashonwindows (the Windows subsystem for Linux) on Windows 10. Instructions for installing poetry on bashonwindows can be found here.

If you do not use bashonwindows, note that 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.

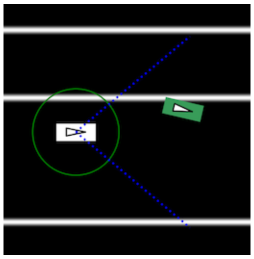
Note: You may also want to install the `Polygon3` package to get faster and more robust polygon triangulation. However, this package is based on the GPC library, which is only free for non-commercial use.

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. 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 the `scenic` module as a script, giving a path to a Scenic file:

```
python -m scenic examples/gta/badlyParkedCar2.scenic
```

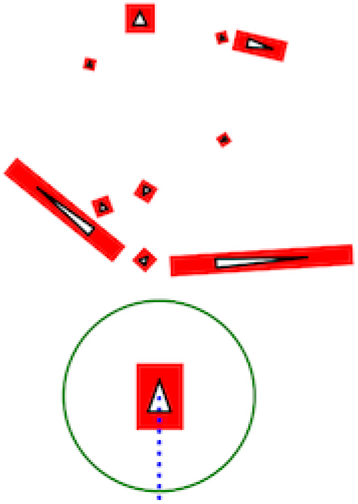
This will compile the Scenic program and sample from it, displaying a schematic of the resulting scene. Since this is the badly-parked car example from our GTA case study, you should get something like this:

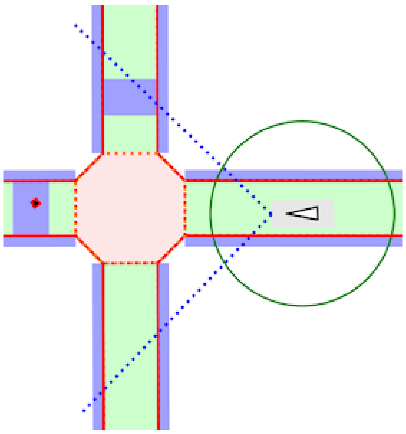


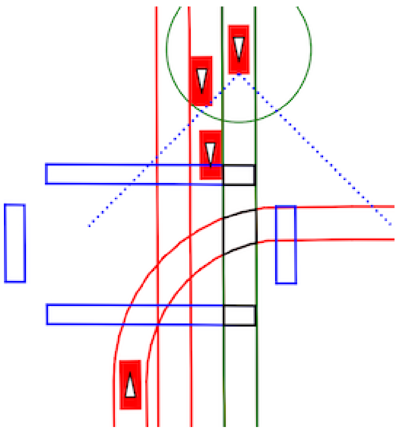
Here the circled rectangle is the ego car; its view cone extends to the right, where we see another car parked rather poorly at the side of the road (the white lines are curbs). If you close the window, Scenic will sample another scene from the same scenario and display it. This will repeat until you 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 Webots:

```
python -m scenic examples/webots/mars/narrowGoal.scenic
python -m scenic examples/webots/road/crossing.scenic
python -m scenic examples/webots/guideways/uberCrash.scenic
```







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 learn how to write Scenic programs, see the *tutorial*.
- If you want to use Scenic with a simulator, see the *Supported Simulators page* (which also describes how to interface Scenic to a new simulator, if the one you want isn't listed).
- If you want to add a feature to the language or otherwise need to understand Scenic's inner workings, see our page on *Scenic Internals*.

1.2 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 [F19].

1.2.1 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 (-10, 10) @ (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 the interval notation `(X, Y)` creates a uniform distribution on the interval, and `X @ Y` creates a vector from *xy* coordinates (as in Smalltalk [GR83]).

1.2.2 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 (-10, 10) @ (20, 40),
    facing (-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:

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 `(-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 `(-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.

1.2.3 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)

[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.

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-indepdendent parts:

```
1 Car left of spot by 0.5,
2     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.

1.2.4 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` with 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) * (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.

1.2.5 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. 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 (-10, 10) @ (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.

1.2.6 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`*).

[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`*).

1.2.7 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 `height` 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 (-2, 2) @ (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 (-1.5, 1.5) @ (0.5, 1.5),
                           facing (-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 (60, 120) deg relative to bottleneck
rightEnd = OrientedPoint right of bottleneck by halfGapWidth,
                         facing (-120, -60) deg relative to bottleneck
Pipe ahead of leftEnd, with height (1, 2)
Pipe ahead of rightEnd, with height (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 (-0.5, 0.5) @ (0.5, 1)
BigRock beyond bottleneck by (-0.5, 0.5) @ (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.

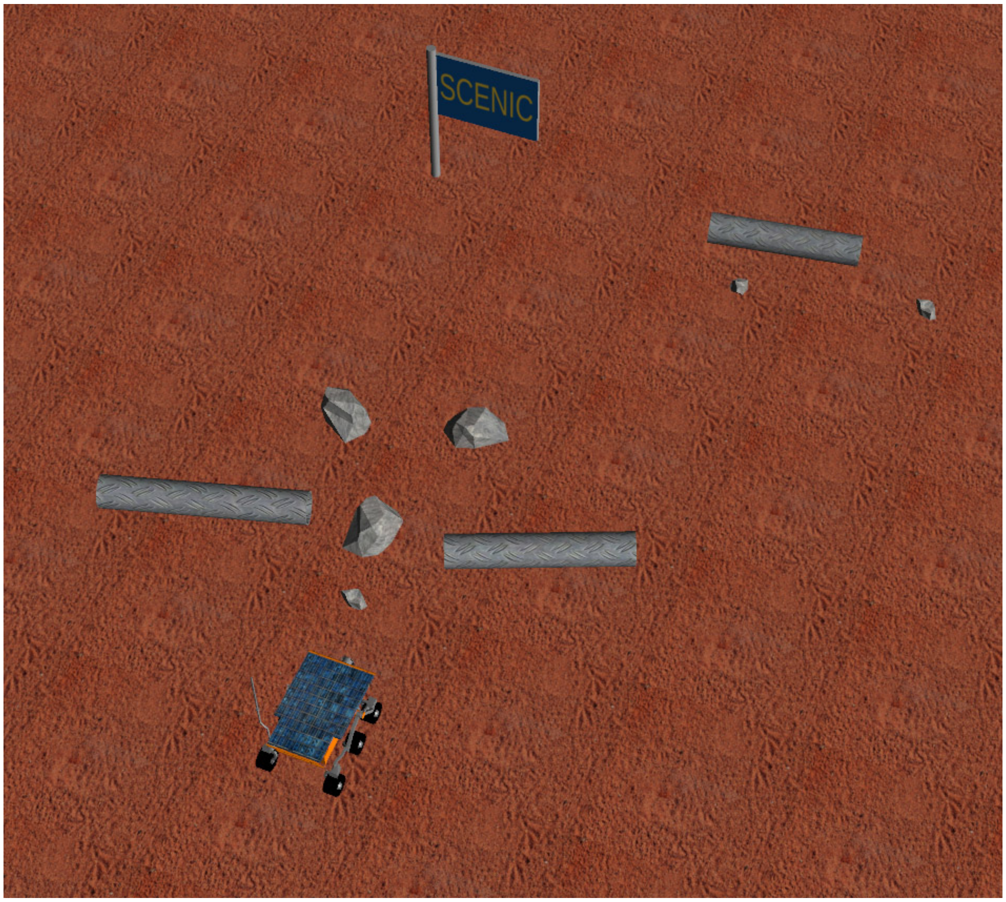
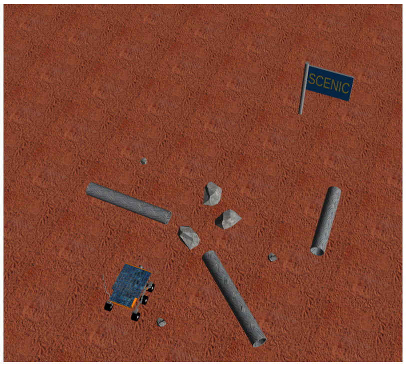
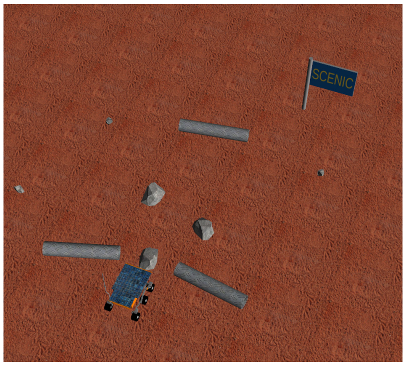
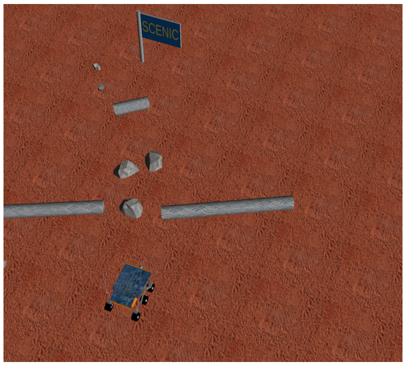


Fig. 3: A scene sampled from the Mars rover scenario, rendered in Webots.







1.2.8 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.

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

References

1.3 Guide to Scenic Syntax

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 *Scenic Syntax Reference*.

1.3.1 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

1.3.2 Distributions

(low, high)	uniformly distributed in the interval
Normal(mean, stdDev)	normal distribution with the given mean and standard deviation
Uniform(value, …)	uniform over a finite set of values
Discrete({value: weight, … })	discrete with given values and weights

1.3.3 Objects

Property	**Default**	**Meaning**
position	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
heading	0	heading in global coordinates
viewAngle	360 degrees	angle for the 'can see' operator
headingStdDev	5 degrees	mutation standard deviation for heading
width	1	width of bounding box (X axis)
height	1	height of bounding box (Y axis)
regionContainedIn	workspace	Region the object must lie within
allowCollisions	false	whether collisions are allowed
requireVisible	true	whether object must be visible from ego

1.3.4 Specifiers

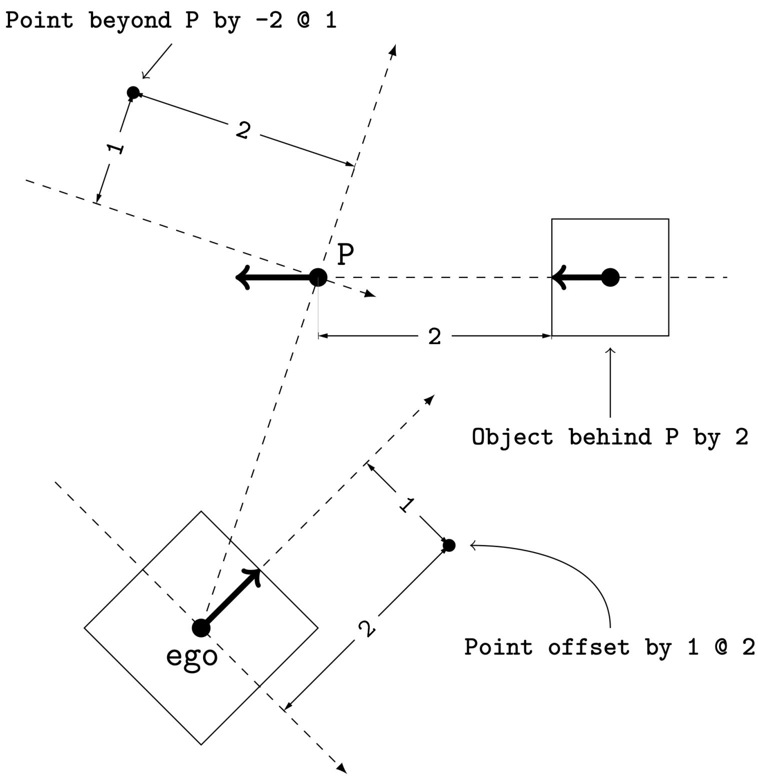


Fig. 4: Illustration of the `beyond`, `behind`, and `offset by` specifiers. Each `OrientedPoint` (e.g. `P`) is shown as a bold arrow.

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 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

Specifiers 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 height
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

Specifiers 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 from the position given by the optional from vector)

1.3.5 Operators

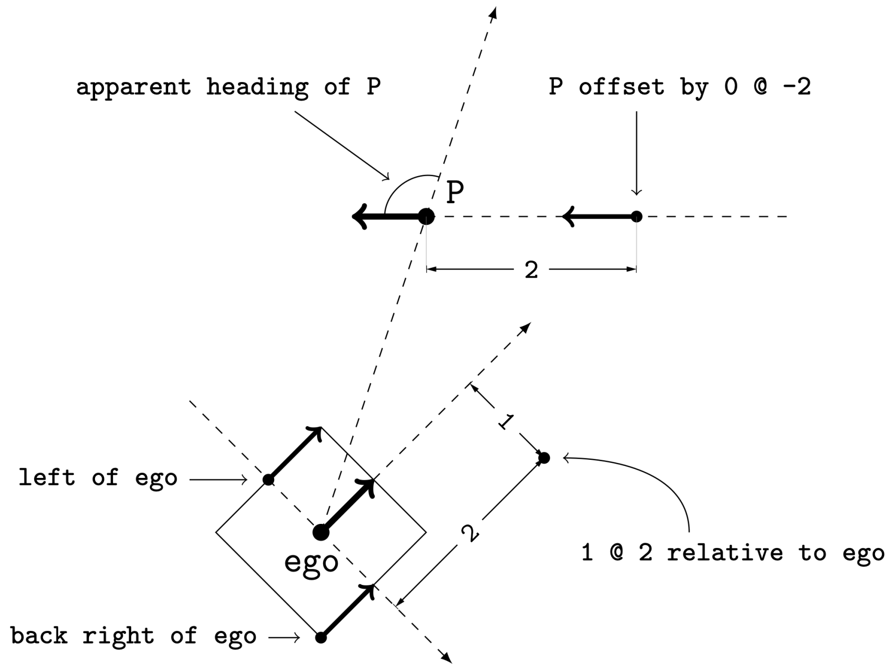


Fig. 5: Illustration of several operators. Each `OrientedPoint` (e.g. `P`) is shown as a bold arrow.

Scalar Operators	Meaning
relative heading of *heading* [from *heading*]	The relative heading of the given heading with respect to ego (or the 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)

Boolean Operators	Meaning
(Point | OrientedPoint) can see (vector | Object)	Whether or not a position or Objectis visible from a Point or OrientedPoint. V
(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
region visible from (*Point* | *OrientedPoint*)	The part of the given region visible from the given Point/OrientedPoint

OrientedPoint Operators	Meaning
vector relative to OrientedPoint	The given vector, interpreted in the local coordinate system of the OrientedPoint
OrientedPoint offset by *vector*	Equivalent to vector relative to OrientedPoint above
(front | back | left | right) of *Object*	The midpoint of the corresponding 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

1.3.6 Statements

Syntax	Meaning
import module	Imports a Scenic or Python module
param identifier = value, . . .	Defines global parameters of the scenario
require boolean	Defines a hard requirement
mutate identifier, . . . [by number]	Enables mutation of the given list of objects

1.4 Scenic Syntax Reference

1.4.1 Primitive Data Types

Scalars

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

Vectors

representing positions and offsets in space, constructed from coordinates with the syntax X @ Y (inspired by Smalltalk). 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. However, it is important to note that the fundamental ideas of Scenic are not specific to 2D, and it would be easy to extend our implementation of the language to support 3D space.

Headings

representing 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, -2 @ 3 means 2 meters left and 3 ahead.

Vector Fields

associating 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

Regions

representing 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. 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.

1.4.2 Position Specifiers

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 height.

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 ego. For example, beyond taxi by 0 @ 3 means 3 meters directly behind the taxi as viewed by the camera.

(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.

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. Uses a forward Euler approximation of the continuous vector field

1.4.3 Heading Specifiers

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 from the position given by the optional from vector). For example, apparently facing 90 deg orients the object so that the camera views its left side head-on

1.4.4 Scalar Operators

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

1.4.5 Boolean Operators

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

Whether or not a position or Objectis 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 that Scenic's visibility model 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 contained in the region. This allows us to use the predicate in two ways

1.4.6 Heading Operators

scalar deg

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

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 deg. If either direction is a vector field, then this operator yields an expression depending on the position property of the object being specified

1.4.7 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 this vector version, 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)

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

1.4.8 Statements

import *module*

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

param *identifier* = *value*, . . .

Defines 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

require *boolean*

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

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

Enables mutation of the given list of objects, adding Gaussian noise with the given standard deviation (default 1) to their position and heading properties. If no objects are specified, mutation applies to every Object already created

1.5 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:

- *Grand Theft Auto V*
- *Webots*
- *X-Plane*

1.5.1 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.

1.5.2 Webots

We have several interfaces to the Webots robotics simulator, for different use cases.

- An interface for the Mars rover example used in our PLDI paper. This interface is extremely simple and might be a good baseline for developing your own interface. See the examples in `examples/webots/mars` and the documentation of *`scenic.simulators.webots.mars`* for details.
- A general interface for traffic scenarios, used in our VerifAI paper. Examples using this interface can be found in the VerifAI repository; see also the documentation of *`scenic.simulators.webots.road`*.
- A more specific interface 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: Our interfaces were written for the R2018 version of Webots, which is not free but has lower hardware requirements than R2019. Relatively minor changes would be required to make our interfaces 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!

1.5.3 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.

1.6 Interfacing to New Simulators

To interface Scenic to a new simulator, there are two steps: using the Scenic API to compile scenarios and generate scenes, and writing a Scenic library defining the virtual world provided by the simulator.

1.6.1 Using the Scenic API

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).

1.6.2 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);
- any other information that might be useful in scenarios.

Then any Scenic programs for your simulator can import this world model and make use of the information within.

Each of the simulators natively supported by Scenic has a corresponding `model.sc` 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.

1.7 Scenic Internals

This section of the documentation describes the implementation of Scenic. It is not intended for ordinary users of Scenic, and will probably only be useful for people who need to make some change to the language (e.g. adding a new type of distribution).

The documentation is organized by the submodules of the main `scenic` module:

scenic.core	Scenic's core types and associated support code.
scenic.simulators	World models and associated code for particular simulators.
scenic.syntax	The Scenic compiler and associated support code.

1.7.1 scenic.core

Scenic's core types and associated support code.

distributions	Objects representing distributions that can be sampled from.
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.
scenarios	Scenario and scene objects.
specifiers	Specifiers and associated objects.
type_support	Support for checking Scenic types.
utils	Assorted utility functions and common exceptions.
vectors	Scenic vectors and vector fields.
workspaces	Workspaces.

scenic.core.distributions

Objects representing distributions that can be sampled from.

Summary of Module Members

Functions

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	

continues on next page

Table 3 – continued from previous page

monotonicDistributionFunction	Like distributionFunction, but additionally specifies that the function is monotonic.
needsSampling	Whether this value requires sampling.
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.

Classes

AttributeDistribution	Distribution resulting from accessing an attribute of a distribution
CustomDistribution	Distribution with a custom sampler given by an arbitrary function
DefaultIdentityDict	Dictionary which is the identity map by default.
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.
TruncatedNormal	Truncated normal distribution.
TupleDistribution	Distributions over tuples (or namedtuples, or lists).

Exceptions

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.

exception RejectionException
Bases: `Exception`

Exception used to signal that the sample currently being generated must be rejected.

class DefaultIdentityDict
Bases: `dict`

Dictionary which is the identity map by default.

class Samplable(*dependencies*)
Bases: `scenic.core.lazy_eval.LazilyEvaluable`

Abstract class for values which can be sampled, possibly depending on other values.

Samplables may specify a proxy object 'self._conditioned' 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.

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.

The default implementation simply returns a dictionary of dependency values. Subclasses must override this method to specify how actual sampling is done.

conditionTo(*value*)
Condition this value to another value with the same conditional distribution.

evaluateIn(*context*)
See LazilyEvaluable.evaluateIn.

dependencyTree()
Debugging method to print the dependency tree of a Samplable.

class Distribution(**dependencies*, *valueType=None*)
Bases: `scenic.core.distributions.Samplable`

Abstract class for distributions.

defaultValueType
alias of `builtins.float`

clone()
Construct an independent copy of this Distribution.

property isPrimitive
Whether this is a primitive Distribution.

bucket(*buckets=None*)
Construct a bucketed approximation of this Distribution.

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.

class CustomDistribution(*sampler*, **dependencies*, *name='CustomDistribution'*, *evaluator=None*)
Bases: `scenic.core.distributions.Distribution`

Distribution with a custom sampler given by an arbitrary function

class TupleDistribution(**coordinates*, *builder=<class 'tuple'>*)
Bases: `scenic.core.distributions.Distribution`, `collections.abc.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(*func*, *args*, *kwargs*, *support=None*)
Bases: `scenic.core.distributions.Distribution`

Distribution resulting from passing distributions to a function

distributionFunction(*method*, *support=None*)
Decorator for wrapping a function so that it can take distributions as arguments.

monotonicDistributionFunction(*method*)
Like distributionFunction, but additionally specifies that the function is monotonic.

class MethodDistribution(*method*, *obj*, *args*, *kwargs*)
Bases: `scenic.core.distributions.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(*attribute*, *obj*)
Bases: `scenic.core.distributions.Distribution`

Distribution resulting from accessing an attribute of a distribution

class OperatorDistribution(*operator*, *obj*, *operands*)
Bases: `scenic.core.distributions.Distribution`

Distribution resulting from applying an operator to one or more distributions

class MultiplexerDistribution(*index*, *options*)
Bases: `scenic.core.distributions.Distribution`

Distribution selecting among values based on another distribution.

class Range(*low*, *high*)
Bases: `scenic.core.distributions.Distribution`

Uniform distribution over a range

class Normal(*mean*, *stddev*)
Bases: `scenic.core.distributions.Distribution`

Normal distribution

class TruncatedNormal(*mean*, *stddev*, *low*, *high*)
Bases: `scenic.core.distributions.Normal`

Truncated normal distribution.

class DiscreteRange(*low*, *high*, *weights=None*)
Bases: *scenic.core.distributions.Distribution*

Distribution over a range of integers.

class Options(*opts*)
Bases: *scenic.core.distributions.MultiplexerDistribution*

Distribution over a finite list of options.

Specified by a dict giving probabilities; otherwise uniform over a given iterable.

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 various global parameters (`param` statements).
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 (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.

continues on next page

Table 6 – continued from previous page

VerifaiSampler	An external sampler exposing the samplers in the VerifAI toolkit.

Member Details

class ExternalSampler(*params*, *globalParams*)

Bases: `object`

Abstract class for objects called to sample values for each external parameter.

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*. 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: *scenic.core.external_params.ExternalSampler*

An external sampler exposing the samplers in the VerifAI toolkit.

The sampler can be configured using the following Scenic global parameters:

- `verifaiSamplerType` – sampler type (see the `verifai.server.choose_sampler` function); the default is `'halton'`
- `verifaiSamplerParams` – `DotMap` of options passed to the sampler

The *VerifaiSampler* supports external parameters which are instances of *VerifaiParameter*.

class ExternalParameter

Bases: *scenic.core.distributions.Distribution*

A value determined by external code rather than Scenic's internal sampler.

sampleGiven (*value*)
Specialization of *Samplable.sampleGiven* for external parameters.

By default, this method simply looks up the value previously sampled by *ExternalSampler.sample*.

class VerifaiParameter (*domain*)
Bases: *scenic.core.external_params.ExternalParameter*

An external parameter sampled using one of VerifAI's samplers.

static withPrior (*dist*, *buckets=None*)
Creates a *VerifaiParameter* using the given distribution as a prior.

Since the VerifAI cross-entropy sampler currently only supports piecewise-constant distributions, if the prior is not of that form it may be approximated. For most built-in distributions, the approximation is exact: for a particular distribution, check its *bucket* method.

class VerifaiRange (*low*, *high*, *buckets=None*, *weights=None*)
Bases: *scenic.core.external_params.VerifaiParameter*

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

class VerifaiDiscreteRange (*low*, *high*, *weights=None*)
Bases: *scenic.core.external_params.VerifaiParameter*

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

class VerifaiOptions (*opts*)
Bases: *scenic.core.distributions.Options*

An *Options* (discrete set) sampled by VerifAI.

scenic.core.geometry

Utility functions for geometric computation.

Summary of Module Members

Functions

addVectors
apparentHeadingAtPoint
averageVectors
circumcircleOfAnnulus
cleanChain
cleanPolygon
cos
findMinMax
headingOfSegment
hypot
max
min
normalizeAngle
plotPolygon
pointIsInCone

continues on next page

Table 7 – continued from previous page

polygonUnion	
positionRelativeToPoint	
radialToCartesian	
rotateVector	
sin	
subtractVectors	
triangulatePolygon	Triangulate the given Shapely polygon.
triangulatePolygon_gpc	
triangulatePolygon_pypoly2tri	
viewAngleToPoint	

Classes

RotatedRectangle	mixin providing collision detection for rectangular objects and regions

Member Details

givePP2TWarning = True

Whether to warn when falling back to pypoly2tri for triangulation

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).

We currently use the GPC library (wrapped by the `Polygon3` package) if it is installed. Since it is not free for commercial use, we don't require it as a dependency, falling back on the BSD-compatible `pypoly2tri` as needed. In this case we issue a warning, since GPC is more robust and handles large polygons. The warning can be disabled by setting *givePP2TWarning* to `False`.

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

Bases: `object`

mixin providing collision detection for rectangular objects and regions

static edgeSeparates (*rectA*, *rectB*)

Whether an edge of rectA separates it from rectB

scenic.core.lazy_eval

Support for lazy evaluation of expressions and specifiers.

Summary of Module Members

Functions

`makeDelayedFunctionCall`	Utility function for creating a lazily-evaluated function call.
`makeDelayedOperatorHandler`	
`needsLazyEvaluation`	
`requiredProperties`	
`toDelayedArgument`	
`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*)

Bases: `object`

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.

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.

class DelayedArgument(*requiredProps*, *value*)

Bases: `scenic.core.lazy_eval.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.

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.

scenic.core.object_types

Implementations of the built-in Scenic classes.

Summary of Module Members

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`	Implementation of the Scenic class `Object`.
`OrientedPoint`	Implementation of the Scenic class `OrientedPoint`.
`Point`	Implementation of the Scenic class `Point`.
`PositionMutator`	Mutator adding Gaussian noise to `position`.

Member Details

class Constructible (**args*, ***kwargs*)
Bases: `scenic.core.distributions.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.

class Mutator
Bases: `object`

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.

appliedTo (*obj*)
Return a mutated copy of the object. Implemented by subclasses.

class PositionMutator (*stddev*)
Bases: `scenic.core.object_types.Mutator`

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

Attributes **stddev** (*float*) – standard deviation of noise

class HeadingMutator (*stddev*)
Bases: `scenic.core.object_types.Mutator`

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

Attributes **stddev** (*float*) – standard deviation of noise

class Point(**args*, ***kwargs*)
Bases: *scenic.core.object_types.Constructible*

Implementation of the Scenic class `Point`.

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

Attributes

- **position** (*Vector*) – 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*).
- **height** (*float*) – Default value zero.

class OrientedPoint(**args*, ***kwargs*)
Bases: *scenic.core.object_types.Point*

Implementation of 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*.

Attributes

- **heading** (*float*) – Heading of the *OrientedPoint*. Default value 0 (North).
- **viewAngle** (*float*) – View cone angle for `can see` operator. Default value 2π.

class Object(**args*, ***kwargs*)
Bases: *scenic.core.object_types.OrientedPoint*, *scenic.core.geometry.RotatedRectangle*

Implementation of the Scenic class `Object`.

Attributes

- **width** (*float*) – Width of the object, i.e. extent along its X axis. Default value 1.
- **height** (*float*) – Height 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`.

scenic.core.pruning

Pruning parts of the sample space which violate requirements.

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 triplet 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.

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.

Summary of Module Members

Functions

`regionFromShapelyObject`	Build a 'Region' from Shapely geometry.
`toPolygon`	

Classes

AllRegion	Region consisting of all space.
CircularRegion	
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	
Region	Abstract class for regions.
SectorRegion	

Member Details

regionFromShapelyObject(*obj*, *orientation=None*)
Build a 'Region' from Shapely geometry.

class PointInRegionDistribution(*region*)
Bases: *scenic.core.vectors.VectorDistribution*

Uniform distribution over points in a Region

class Region(*name*, **dependencies*, *orientation=None*)
Bases: *scenic.core.distributions.Samplable*

Abstract class for regions.

intersect(*other*, *triedReversed=False*)
Get a *Region* representing the intersection of this one with another.

static uniformPointIn(*region*)
Get a uniform *Distribution* over points in a *Region*.

uniformPoint()
Sample a uniformly-random point in this *Region*.

Can only be called on fixed Regions with no random parameters.

uniformPointInner()
Do the actual random sampling. Implemented by subclasses.

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

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.

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: `scenic.core.regions.Region`

Region consisting of all space.

class EmptyRegion(*name*, **dependencies*, *orientation=None*)
Bases: `scenic.core.regions.Region`

Region containing no points.

class PolylineRegion(*points=None*, *polyline=None*, *orientation=True*)
Bases: `scenic.core.regions.Region`

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

class PolygonalRegion(*points=None*, *polygon=None*, *orientation=None*)
Bases: `scenic.core.regions.Region`

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

class PointSetRegion(*name*, *points*, *kdTree=None*, *orientation=None*, *tolerance=1e-06*)
Bases: `scenic.core.regions.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** (`iterable`) – 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) – 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: `scenic.core.regions.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.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(*workspace*, *objects*, *egoObject*, *params*)
Bases: `object`

A scene generated from a Scenic scenario.

Attributes

- **objects** (tuple(*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.

show(*zoom=None*, *block=True*)
Render a schematic of the scene for debugging.

class Scenario(*workspace*, *objects*, *egoObject*, *params*, *externalParams*, *requirements*, *requirementDeps*)
Bases: `object`

A compiled Scenic scenario, from which scenes can be sampled.

validate()
Make some simple static checks for inconsistent built-in requirements.

generate(*maxIterations=2000*, *verbosity=0*, *feedback=None*)
Sample a *Scene* from this scenario.

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.

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={}*)
Bases: `object`

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*)
Bases: `object`

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.

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, printing an error if impossible.

continues on next page

Table 17 – continued from previous page

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, printing an error if impossible.
toScalar	Convert something to a scalar, printing an error if impossible.
toType	Convert something to a given type, printing an error if impossible.
toTypes	Convert something to any of the given types, printing an error if impossible.
toVector	Convert something to a vector, printing 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	Lazily evaluates a function, after checking that two lazy values have the same type.

Member Details

class Heading

Bases: `object`

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*)

Coerce something into the given type.

coerceToAny (*thing*, *types*, *error*)
Coerce something into any of the given types, printing an error if impossible.

toTypes (*thing*, *types*, *typeError='wrong type'*)
Convert something to any of the given types, printing an error if impossible.

toType (*thing*, *ty*, *typeError='wrong type'*)
Convert something to a given type, printing an error if impossible.

toScalar (*thing*, *typeError='non-scalar in scalar context'*)
Convert something to a scalar, printing an error if impossible.

toHeading (*thing*, *typeError='non-heading in heading context'*)
Convert something to a heading, printing an error if impossible.

toVector (*thing*, *typeError='non-vector in vector context'*)
Convert something to a vector, printing an error if impossible.

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.

class TypeChecker (*arg*, *types*, *error*)
Bases: `scenic.core.lazy_eval.DelayedArgument`

Checks that a given lazy value has one of a given list of types.

class TypeEqualityChecker (*func*, *checkA*, *checkB*, *error*)
Bases: `scenic.core.lazy_eval.DelayedArgument`

Lazily evaluates a function, after checking that two lazy values have the same type.

scenic.core.utils

Assorted utility functions and common exceptions.

Summary of Module Members

Functions

`areEquivalent`	Whether two objects are equivalent, i.e.
`argsToString`	
`cached`	Decorator for making a method with no arguments cache its result

Exceptions

`InconsistentScenarioError`	Error for scenarios with inconsistent requirements.
`InvalidScenarioError`	Error raised for syntactically-valid but otherwise problematic Scenic programs.
`ParseError`	An error produced by attempting to parse an invalid Scenic program.

continues on next page

Table 20 – continued from previous page

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

Member Details

cached(*oldMethod*)
Decorator for making a method with no arguments cache its result

areEquivalent(*a, b*)
Whether two objects are equivalent, i.e. have the same properties.

This is only used for debugging, e.g. to check that a Distribution is the same before and after pickling. We don't want to define __eq__ for such objects since for example two values sampled with the same distribution are equivalent but not semantically identical: the code:

```
X = (0, 1)
Y = (0, 1)
```

does not make X and Y always have equal values!

exception ParseError
Bases: `Exception`

An error produced by attempting to parse an invalid Scenic program.

exception RuntimeParseError
Bases: *scenic.core.utils.ParseError*

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

exception InvalidScenarioError
Bases: `Exception`

Error raised for syntactically-valid but otherwise problematic Scenic programs.

exception InconsistentScenarioError(*line, message*)
Bases: *scenic.core.utils.InvalidScenarioError*

Error for scenarios with inconsistent requirements.

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

CustomVectorDistribution	Distribution with a custom sampler given by an arbitrary function.
OrientedVector	
PolygonalVectorField	
Vector	A 2D vector, whose coordinates can be distributions.
VectorDistribution	A distribution over Vectors.
VectorField	
VectorMethodDistribution	Vector version of MethodDistribution.
VectorOperatorDistribution	Vector version of OperatorDistribution.

Member Details

class VectorDistribution(**dependencies*, *valueType=None*)

Bases: *scenic.core.distributions.Distribution*

A distribution over Vectors.

defaultValueType

alias of *Vector*

class CustomVectorDistribution(*sampler*, **dependencies*, *name='CustomVectorDistribution'*, *evaluator=None*)

Bases: *scenic.core.vectors.VectorDistribution*

Distribution with a custom sampler given by an arbitrary function.

class VectorOperatorDistribution(*operator*, *obj*, *operands*)

Bases: *scenic.core.vectors.VectorDistribution*

Vector version of OperatorDistribution.

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

Bases: *scenic.core.vectors.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: *scenic.core.distributions.Samplable*, `collections.abc.Sequence`

A 2D vector, whose coordinates can be distributions.

rotatedBy(*angle*)

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

scenic.core.workspaces

Workspaces.

Summary of Module Members

Classes

Workspace	A workspace describing the fixed world of a scenario

Member Details

class Workspace(*region=<scenic.core.regions.AllRegion object>*)

Bases: *scenic.core.regions.Region*

A workspace describing the fixed world of a scenario

show(*plt*)

Render a schematic of the workspace for debugging

zoomAround(*plt, objects, expansion=2*)

Zoom the schematic around the specified objects

scenicToSchematicCoords(*coords*)

Convert Scenic coordinates to those used for schematic rendering.

1.7.2 scenic.simulators

World models and associated code for particular simulators.

carla	Scenic world model for the CARLA driving simulator.
gta	Scenic world model for Grand Theft Auto V (GTAV).
webots	Scenic world models for the Webots robotics simulator.
xplane	Scenic world model for the X-Plane flight simulator.
formats	Support for file formats not specific to particular simulators.

scenic.simulators.carla

Scenic world model for the CARLA driving simulator.

This model is designed to be used with the CARLA interface to the VerifAI toolkit. See the VerifAI repository for further documentation and examples.

The model currently supports vehicles, pedestrians, and props. Vehicles have an `agent` parameter, which specifies the agent to be used to control the vehicle.

In addition, the model uses several global parameters to control weather (descriptions are from the CARLA Python API reference):

- `cloudiness` (float): Weather cloudiness. It only affects the RGB camera sensor. Values range from 0 to 100.

- `precipitation` (float): Precipitation amount for controlling rain intensity. It only affects the RGB camera sensor. Values range from 0 to 100.
- `precipitation_deposits` (float): Precipitation deposits for controlling the area of puddles on roads. It only affects the RGB camera sensor. Values range from 0 to 100.
- `wind_intensity` (float): Wind intensity, it affects the clouds moving speed, the raindrop direction, and vegetation. This doesn't affect the car physics. Values range from 0 to 100.
- `sun_azimuth_angle` (float): The azimuth angle of the sun in degrees. Values range from 0 to 360 (degrees).
- `sun_altitude_angle` (float): Altitude angle of the sun in degrees. Values range from -90 to 90 (where 0 degrees is the horizon).

model	Scenic world model for traffic scenarios in CARLA.
map	Stub to allow changing the map without having to change the model.
interface	Support code for the CARLA world model.
car_models	
prop_models	

scenic.simulators.carla.model

Scenic world model for traffic scenarios in CARLA.

Summary of Module Members

Classes

Bicycle
Car
Cone
Motorcycle
Pedestrian
Prop
Trash
Truck
Vehicle

Member Details

scenic.simulators.carla.map

Stub to allow changing the map without having to change the model.

Summary of Module Members

Functions

`setMapPath`

Member Details

scenic.simulators.carla.interface

Support code for the CARLA world model.

Summary of Module Members

Classes

`CarlaWorkspace`

Member Details

scenic.simulators.carla.car_models

scenic.simulators.carla.prop_models

scenic.simulators.gta

Scenic world model for Grand Theft Auto V (GTAV).

model	World model for GTA.
interface	Python supporting code for the GTA model.
center_detection	This file contains helper functions
img_modf	This file has basic image modification functions
map	
messages	

scenic.simulators.gta.model

World model for GTA.

Summary of Module Members

Functions

createPlatoonAt	Create a platoon starting from the given car.

Classes

Bus	Convenience subclass for buses.
Car	Scenic class for cars.
Compact	Convenience subclass for compact cars.
EgoCar	Convenience subclass with defaults for ego cars.

Member Details

roadDirection = <scenic.core.vectors.VectorField object>
Vector field representing the nominal traffic direction at a point on the road

road = <scenic.core.regions.GridRegion object>
Region representing the roads in the GTA map.

curb = <scenic.core.regions.PointSetRegion object>
Region representing the curbs in the GTA map.

workspace = <scenic.simulators.gta.interface.MapWorkspace object>
Workspace over the *road* Region.

class Car(**args*, ***kwargs*)
Bases: *scenic.core.object_types.Object*

Scenic class for cars.

Attributes

- **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** (*CarColor* or RGB tuple) – Color of the car.

class EgoCar(**args*, ***kwargs*)
Bases: *scenic.simulators.gta.model.Car*

Convenience subclass with defaults for ego cars.

class Bus (**args, **kwargs*)
Bases: `scenic.simulators.gta.model.Car`

Convenience subclass for buses.

class Compact (**args, **kwargs*)
Bases: `scenic.simulators.gta.model.Car`

Convenience subclass for compact cars.

createPlatoonAt (*car, numCars, model=None, dist=<scenic.core.distributions.Range object>, shift=<scenic.core.distributions.Range object>, wiggle=0*)
Create a platoon starting from the given car.

scenic.simulators.gta.interface

Python supporting code for the GTA model.

Summary of Module Members

Classes

`CarColor`	A car color as an RGB tuple.
`CarColorMutator`	Mutator that adds Gaussian HSL noise to the `color` property.
`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
`NoisyColorDistribution`	A distribution given by HSL noise around a base color.

Member Details

class Map (*imagePath, Ax, Ay, Bx, By*)
Bases: `object`

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: `scenic.core.workspaces.Workspace`

Workspace whose rendering is handled by a Map

class CarModel(*name*, *width*, *height*, *viewAngle=1.5707963267948966*)
Bases: `object`

A model of car in GTA.

Attributes

- **name** (*str*) – name of model in GTA
- **width** (*float*) – width of this model of car
- **height** (*float*) – height 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`

class CarColor
Bases: `scenic.simulators.gta.interface.CarColor`

A car color as an RGB tuple.

static uniformColor()
Return a uniformly random color.

static defaultColor()
Default color distribution for cars.

The distribution starts with a base distribution over 9 discrete colors, then adds Gaussian HSL noise. The base distribution uses color popularity statistics from a 2012 DuPont survey.

class NoisyColorDistribution(*baseColor*, *hueNoise*, *satNoise*, *lightNoise*)
Bases: `scenic.core.distributions.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 CarColorMutator
Bases: `scenic.core.object_types.Mutator`

Mutator that adds Gaussian HSL noise to the `color` property.

scenic.simulators.gta.center_detection

This file contains helper functions

Summary of Module Members

Functions

`compute_bb`	
`compute_gradient_manual`	
`compute_gradient_sobel`	
`compute_heading`	
`compute_midpoints`	
`find_center`	Find which edge x lies in
`generate_circle`	
`generate_connected_edges`	
`generate_neighbors`	
`sample_from_road`	
`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: `tuple`

property init_theta
: Alias for field number 0

property tangent
: Alias for field number 1

property opp_loc
: Alias for field number 2

property mid_loc
: Alias for field number 3

_asdict()
: Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)
: 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`
`plot_voronoi_plot`
`voronoi_edge`

Member Details

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`

continues on next page

Table 38 – continued from previous page

Stop
Vehicle

Member Details

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. World models for particular uses of Webots are in submodules.

mars	World model for a simple Mars rover example in Webots.
road	World model and associated code for traffic scenarios in Webots.
guideways	World model for road intersection scenarios in Webots.
common	Common Webots interface.
world_parser	Parser for WBT files using ANTLR.

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

Bases: `scenic.core.object_types.Object`

Flag indicating the goal location.

class Rover(**args*, ***kwargs*)

Bases: `scenic.core.object_types.Object`

Mars rover.

class Debris(**args*, ***kwargs*)

Bases: `scenic.core.object_types.Object`

Abstract class for debris scattered randomly in the workspace.

class BigRock(**args*, ***kwargs*)

Bases: `scenic.simulators.webots.mars.model.Debris`

Large rock.

class Rock(**args*, ***kwargs*)

Bases: `scenic.simulators.webots.mars.model.Debris`

Small rock.

class Pipe(**args*, ***kwargs*)

Bases: `scenic.simulators.webots.mars.model.Debris`

Pipe with variable length.

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.

model	Scenic world model for traffic scenarios in Webots.
world	Stub to allow changing the Webots world without changing the model.
interface	Python library supporting the main Scenic module.
car_models	Car models built into Webots.

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

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.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*)

Bases: `object`

Objects with OSM id tags

class Road(*attrs*, *driveOnLeft=False*)

Bases: *scenic.simulators.webots.road.interface.OSMObject*

OSM roads

class Crossroad(*attrs*)

Bases: *scenic.simulators.webots.road.interface.OSMObject*

OSM crossroads

class PedestrianCrossing(*attrs*)

Bases: `object`

PedestrianCrossing nodes

scenic.simulators.webots.road.car_models

Car models built into Webots.

Summary of Module Members

Classes

CarModel

Member Details

class CarModel(*name*, *width*, *height*)
Bases: `tuple`

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)
Make a new CarModel object from a sequence or iterable

_replace(***kwds*)
Return a new CarModel object replacing specified fields with new values

property height
Alias for field number 2

property name
Alias for field number 0

property width
Alias for field number 1

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.

model
intersection
interface

scenic.simulators.webots.guideways.model

Summary of Module Members

Classes

`Car`
`Marker`

Member Details

scenic.simulators.webots.guideways.intersection

Summary of Module Members

Functions

`setLocalIntersection`

Member Details

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.common

Common Webots interface.

Summary of Module Members

Functions

`scenicToWebotsPosition`	
`scenicToWebotsRotation`	
`webotsToScenicPosition`	Convert Webots positions to Scenic positions.
`webotsToScenicRotation`	

Member Details

`webotsToScenicPosition`(*pos*)
Convert Webots positions to Scenic positions.

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*)
Bases: `object`

A generic VRML node

class ErrorReporter
Bases: `antlr4.error.ErrorListener.ErrorListener`

ANTLR listener for reporting parse errors

class Evaluator(*nodeClasses*)
Bases: `scenic.simulators.webots.WBTVisitor.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(**args*, ***kwargs*)
Bases: `scenic.core.object_types.Object`

Placeholder object for the plane.

scenic.simulators.formats

Support for file formats not specific to particular simulators.

`opendrive`	Support for loading OpenDRIVE maps.

scenic.simulators.formats.opendrive

Support for loading OpenDRIVE maps.

workspace	Workspaces based on OpenDRIVE maps.
xodr_parser	Parser for OpenDRIVE (.xodr) files.

scenic.simulators.formats.opendrive.workspace

Workspaces based on OpenDRIVE maps.

Summary of Module Members

Classes

`OpenDriveWorkspace`

Member Details

scenic.simulators.formats.opendrive.xodr_parser

Parser for OpenDRIVE (.xodr) files.

Summary of Module Members

Functions

`buffer_union`

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].

continues on next page

Table 62 – continued from previous page

Poly3	Cubic polynomial.
Road	
RoadLink	Indicates Roads a and b, with ids id_a and id_b respectively, are connected.
RoadMap	

Member Details

class Poly3(*a*, *b*, *c*, *d*)

Bases: `object`

Cubic polynomial.

class Curve(*x0*, *y0*, *hdg*, *length*)

Bases: `object`

Geometric elements which compose road reference lines. See the OpenDRIVE Format Specification for coordinate system details.

abstract to_points(*num*)

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.

rel_to_abs(*points*)

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: *scenic.simulators.formats.opendrive.xodr_parser.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: *scenic.simulators.formats.opendrive.xodr_parser.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: *scenic.simulators.formats.opendrive.xodr_parser.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: *scenic.simulators.formats.opendrive.xodr_parser.Curve*

A line segment between (x0, y0) and (x1, y1).

class RoadLink(*id_a*, *id_b*, *contact_a*, *contact_b*)

Bases: `object`

Indicates Roads a and b, with ids id_a and id_b respectively, are connected.

1.7.3 scenic.syntax

The Scenic compiler and associated support code.

relations	Extracting relations (for later pruning) from the syntax of requirements.
translator	Translator turning Scenic programs into Scenario objects.
veneer	Python implementations of Scenic language constructs.

scenic.syntax.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*)
Bases: `object`

Abstract relation bounding something about another object.

class RelativeHeadingRelation(*target*, *lower*, *upper*)
Bases: *scenic.syntax.relations.BoundRelation*

Relation bounding another object's relative heading with respect to this one.

class DistanceRelation(*target*, *lower*, *upper*)
Bases: *scenic.syntax.relations.BoundRelation*

Relation bounding another object's distance from this one.

scenic.syntax.translator

Translator turning Scenic programs into Scenario objects.

The top-level interface to Scenic is provided by two functions:

- *scenarioFromString* – compile a string of Scenic code;
- *scenarioFromFile* – compile a Scenic file.

These output a *Scenario* object, from which scenes can be generated. See the documentation for *Scenario* for details.

When imported, this module hooks the Python import system 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. 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

Functions

compileStream	Compile a stream of Scenic code and execute it in a namespace.
compileTranslatedTree	
constructScenarioFrom	Build a Scenario object from an executed Scenic module.
executeCodeIn	Execute the final translated Python code in the given namespace.
executePythonFunction	Execute a Python function, giving correct Scenic backtraces for any exceptions.
findConstructorsIn	Find all constructors (Scenic classes) defined in a namespace.
generateTracebackFrom	Trim an exception's traceback to the last line of Scenic code.
hooked_import	Version of __import__ hooked by Scenic to capture Scenic modules.
parseTranslatedSource	
partitionByImports	Partition the tokens into blocks ending with import statements.
peek	

continues on next page

Table 66 – continued from previous page

`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`.
`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`	
`Peekable`	Utility class to allow iterator lookahead.
`ScenicLoader`	
`ScenicMetaFinder`	
`TokenTranslator`	Translates a Scenic token stream into valid Python syntax.

Exceptions

`ASTParseError`	Parse error occuring during modification of the Python AST.
`InterpreterParseError`	Parse error occuring during Python execution.
`PythonParseError`	Parse error occurring during Python parsing or compilation.
`TokenParseError`	Parse error occurring during token translation.

Member Details

`scenarioFromString`(*string*, *filename='<string>'*, *cacheImports=False*)

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

The optional **filename** is used for error messages.

`scenarioFromFile`(*path*, *cacheImports=False*)

Compile a Scenic file into a `Scenario`.

Parameters

- **`path`** (*`str`*) – path to a Scenic file
- **`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.

Returns A *Scenario* object representing the Scenic scenario.

scenarioFromStream(*stream*, *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.

compileStream(*stream*, *namespace*, *filename='<stream>'*)
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*, *parent*, *specifiers*)
Bases: `tuple`

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)
Make a new Constructor object from a sequence or iterable

_replace(***kwds*)
Return a new Constructor object replacing specified fields with new values

property name
Alias for field number 0

property parent
Alias for field number 1

property specifiers
Alias for field number 2

class InfixOp(*syntax*, *implementation*, *arity*, *token*, *node*)
Bases: `tuple`

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)
Make a new InfixOp object from a sequence or iterable

_replace(***kwds*)
Return a new InfixOp object replacing specified fields with new values

property arity
Alias for field number 2

property implementation
Alias for field number 1

property node
Alias for field number 4

property syntax
Alias for field number 0

property token
Alias for field number 3

hooked_import(**args*, ***kwargs*)
Version of __import__ hooked by Scenic to capture Scenic modules.

partitionByImports(*tokens*)
Partition the tokens into blocks ending with import statements.

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

exception TokenParseError(*tokenOrLine*, *message*)
Bases: `scenic.core.utils.ParseError`

Parse error occurring during token translation.

class Peekable(*gen*)
Bases: `object`

Utility class to allow iterator lookahead.

class TokenTranslator(*constructors=()*)
Bases: `object`

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*)
Do the actual translation of the token stream.

exception PythonParseError
Bases: `SyntaxError`, `scenic.core.utils.ParseError`

Parse error occurring during Python parsing or compilation.

class AttributeFinder(*target*)
Bases: `ast.NodeVisitor`

Utility class for finding all referenced attributes of a given name.

exception ASTParseError(*line*, *message*)
Bases: `scenic.core.utils.ParseError`

Parse error occuring during modification of the Python AST.

translateParseTree(*tree*, *constructors*)
Modify the Python AST to produce the desired Scenic semantics.

generateTracebackFrom(*exc*, *sourceFile*)
Trim an exception's traceback to the last line of Scenic code.

exception InterpreterParseError(*exc*, *line*)
Bases: `scenic.core.utils.ParseError`

Parse error occuring during Python execution.

executeCodeIn(*code*, *namespace*, *filename*)
Execute the final translated Python code in the given namespace.

executePythonFunction(*func*, *filename*)
Execute a Python function, giving correct Scenic backtraces for any exceptions.

storeScenarioStateIn(*namespace*, *requirementSyntax*, *filename*)
Post-process an executed Scenic module, extracting state from the veneer.

constructScenarioFrom(*namespace*)
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 <vector> [to <vector>]' operator.
`Facing`	The 'facing X' polymorphic specifier.
`FacingToward`	The 'facing toward <vector>' specifier.
`FieldAt`	The '<VectorField> at <vector>' operator.

continues on next page

Table 69 – continued from previous page

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.
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.
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.
`Uniform`	
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.
`deactivate`	Deactivate the veneer after compiling a Scenic module.
ego	Function implementing loads and stores to the 'ego' pseudo-variable.
`getAllGlobals`	Find all names the given lambda depends on, along with their current bindings.
`isActive`	Are we in the middle of compiling a Scenic module?
`leftSpecHelper`	
mutate	Function implementing the mutate statement.
param	Function implementing the param statement.
`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.
resample	The built-in resample function.
verbosePrint	Built-in function printing a message when the verbosity is >0.

Member Details

class Vector(*x*, *y*)

Bases: `scenic.core.distributions.Samplable`, `collections.abc.Sequence`

A 2D vector, whose coordinates can be distributions.

rotatedBy(*angle*)

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

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

Bases: `scenic.core.distributions.Samplable`

Abstract class for regions.

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

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

static uniformPointIn(*region*)

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

uniformPoint()

Sample a uniformly-random point in this `Region`.

Can only be called on fixed Regions with no random parameters.

uniformPointInner()

Do the actual random sampling. Implemented by subclasses.

containsPoint(*point*)

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

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.

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: `scenic.core.regions.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** (*iterable*) – 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) – orientation for the region
- **tolerance** (*float, optional*) – distance tolerance for checking whether a point lies in the region

class PolygonalRegion(*points=None*, *polygon=None*, *orientation=None*)
Bases: `scenic.core.regions.Region`

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

class PolylineRegion(*points=None*, *polyline=None*, *orientation=True*)
Bases: `scenic.core.regions.Region`

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

class Workspace(*region=<scenic.core.regions.AllRegion object>*)
Bases: `scenic.core.regions.Region`

A workspace describing the fixed world of a scenario

show(*plt*)
Render a schematic of the workspace for debugging

zoomAround(*plt*, *objects*, *expansion=2*)
Zoom the schematic around the specified objects

scenicToSchematicCoords(*coords*)
Convert Scenic coordinates to those used for schematic rendering.

class Range(*low*, *high*)
Bases: `scenic.core.distributions.Distribution`

Uniform distribution over a range

class Options(*opts*)
Bases: `scenic.core.distributions.MultiplexerDistribution`

Distribution over a finite list of options.

Specified by a dict giving probabilities; otherwise uniform over a given iterable.

class Normal(*mean*, *stddev*)
Bases: `scenic.core.distributions.Distribution`

Normal distribution

Discrete
alias of `scenic.core.distributions.Options`

class VerifaiParameter(*domain*)
Bases: `scenic.core.external_params.ExternalParameter`

An external parameter sampled using one of VerifAI's samplers.

static withPrior(*dist*, *buckets=None*)
Creates a `VerifaiParameter` using the given distribution as a prior.

Since the VerifAI cross-entropy sampler currently only supports piecewise-constant distributions, if the prior is not of that form it may be approximated. For most built-in distributions, the approximation is exact: for a particular distribution, check its `bucket` method.

class VerifaiRange(*low*, *high*, *buckets=None*, *weights=None*)
Bases: `scenic.core.external_params.VerifaiParameter`

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

class VerifaiDiscreteRange(*low*, *high*, *weights=None*)
Bases: `scenic.core.external_params.VerifaiParameter`

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

class VerifaiOptions(*opts*)
Bases: *scenic.core.distributions.Options*

An *Options* (discrete set) sampled by VerifAI.

class Mutator
Bases: `object`

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.

appliedTo(*obj*)
Return a mutated copy of the object. Implemented by subclasses.

class Point(**args*, ***kwargs*)
Bases: *scenic.core.object_types.Constructible*

Implementation of the Scenic class `Point`.

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

Attributes

- **position** (*Vector*) – 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*).
- **height** (*float*) – Default value zero.

class OrientedPoint(**args*, ***kwargs*)
Bases: *scenic.core.object_types.Point*

Implementation of 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*.

Attributes

- **heading** (*float*) – Heading of the *OrientedPoint*. Default value 0 (North).
- **viewAngle** (*float*) – View cone angle for `can see` operator. Default value 2π.

class Object(**args*, ***kwargs*)
Bases: *scenic.core.object_types.OrientedPoint*, *scenic.core.geometry.RotatedRectangle*

Implementation of the Scenic class `Object`.

Attributes

- **width** (*float*) – Width of the object, i.e. extent along its X axis. Default value 1.
- **height** (*float*) – Height 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`.

class PropertyDefault(*requiredProperties*, *attributes*, *value*)
Bases: `object`

A default value, possibly with dependencies.

resolveFor(*prop*, *overriddenDefs*)
Create a Specifier for a property from this default and any superclass defaults.

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.

require(*reqID*, *req*, *line*, *prob=1*)
Function implementing the require statement.

resample(*dist*)
The built-in resample function.

verbosePrint(*msg*)
Built-in function printing a message when the verbosity is >0.

param(**quotedParams*, ***params*)
Function implementing the param statement.

mutate(**objects*)
Function implementing the mutate statement.

Visible(*region*)
The 'visible <region>' operator.

FieldAt(*X*, *Y*)
The '<VectorField> at <vector>' operator.

RelativeTo(*X*, *Y*)
The 'X relative to Y' polymorphic operator.

Allowed forms: F relative to G (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>

RelativePosition(*X*, *Y=None*)
The 'relative position of <vector> [from <vector>]' operator.

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

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.

`DistanceFrom`(*X*, *Y=None*)
The ‘distance from <vector> [to <vector>]’ operator.

If the ‘to <vector>’ is omitted, the position of ego 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.

`CanSee`(*X*, *Y*)
The ‘X can see Y’ polymorphic operator.

Allowed forms: <point> can see <object> <point> can see <vector>

`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.

`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 'height'. 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 'height'. 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.

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.

The `scenic` module itself provides two functions as the top-level interface to Scenic:

scenarioFromFile (*path*, *cacheImports=False*)
Compile a Scenic file into a `Scenario`.

Parameters

- **path** (*str*) – path to a Scenic file
- **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.

Returns A `Scenario` object representing the Scenic scenario.

scenarioFromString (*string*, *filename='<string>'*, *cacheImports=False*)
Compile a string of Scenic code into a `Scenario`.

The optional **filename** is used for error messages.

1.8 Credits

If you use Scenic, we request that you cite our PLDI 2019.

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 developed by Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia.

Edward Kim assisted in putting together this documentation.

The Scenic tool has benefitted from code contributions from:

- Johnathan Chiu
- Francis Indaheng
- Martin Jansa (LG Electronics, Inc.)
- Wilson Wu

Finally, many other people provided helpful advice and discussions, including:

- Ankush Desai
- Alastair Donaldson
- Andrew Gordon
- Jonathan Ragan-Kelley
- Sriram Rajamani
- Marcell Vazquez-Chanlatte

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

THREE

LICENSE

Scenic is distributed under the 3-Clause BSD License.

BIBLIOGRAPHY

[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

S

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

W

Z