Scenic

**Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Edward I

Apr 14, 2021

INTRODUCTION

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

Scenic was designed and implemented by Daniel J. Fremont, Edward Kim, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. For a description of the language and some of its applications, see our preprint, which extends our PLDI 2019 paper (a more in-depth discussion of the PLDI paper is in Chapters 5 and 8 of this thesis). Our *publications* page lists additional papers using Scenic.

Note: The syntax of Scenic 2.x is not completely backwards-compatible with 1.x, which was used in our papers prior to late 2020. See *What's New in Scenic* for a list of syntax changes and new features. If your existing code no longer works, install the latest 1.x release from GitHub.

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

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Getting Started with Scenic

1.1.1 Installation

Scenic requires **Python 3.7** or newer. You can install Scenic from PyPI by simply running:

```
$ pip install scenic
```

Alternatively, if you want to run some of our example scenarios or modify Scenic, you can download or clone the Scenic repository. Install Poetry, optionally activate the virtual environment in which you would like to run Scenic, and then run:

```
$ poetry install
```

If you will be developing Scenic, add the `-E dev` option when invoking Poetry.

Note: If you are not already using a virtual environment, **`poetry install`** will create one. You can then run **`poetry shell`** to create a terminal inside the environment for running the commands below.

Note: If you get an error saying that your machine does not have a compatible version, this means that you do not have Python 3.7 or later on your PATH. Install a newer version of Python, either directly from the Python website or using pyenv (e.g. running **`pyenv install 3.8.5`**). If you install it somewhere that is not on your PATH (so running **`python --version`** doesn't give you the correct version), you'll need to run **`poetry env use /full/path/to/python`** before running **`poetry install`**.

Installing via either **`pip`** or Poetry will install all of the dependencies which are required to run Scenic.

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.

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: On some platforms, in particular OS X, you may get an error during the installation of pygame due to missing SDL files. Try installing SDL: on OS X, if you use Homebrew you can simply run **`brew install sdl`**.

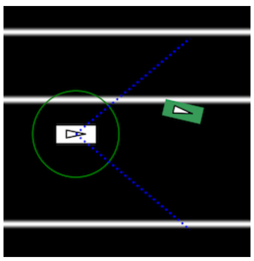
On OS X you may also get an error during the installation of Pillow due to missing zlib or jpeg libraries. If you use Homebrew, you can install these with **brew install zlib** and **brew install libjpeg**.

1.1.2 Trying Some Examples

The Scenic repository contains many example scenarios, found in the `examples` directory. They are organized by the simulator they are written for, e.g. GTA (Grand Theft Auto V) or Webots; there are also cross-platform scenarios written for Scenic's abstract application domains, e.g. the *driving domain*. Each simulator has a specialized Scenic interface which requires additional setup (see *Supported Simulators*); however, for convenience Scenic provides an easy way to visualize scenarios without running a simulator. Simply run **scenic**, giving a path to a Scenic file:

```
$ scenic examples/gta/badlyParkedCar2.scenic
```

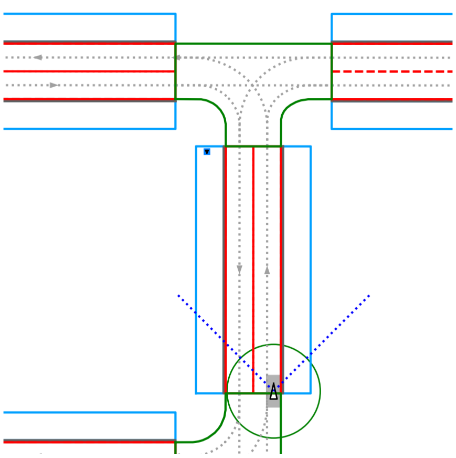
This will compile the Scenic program and sample from it, displaying a schematic of the resulting scene. Since this is the badly-parked car example from our GTA case study, you should get something like this:

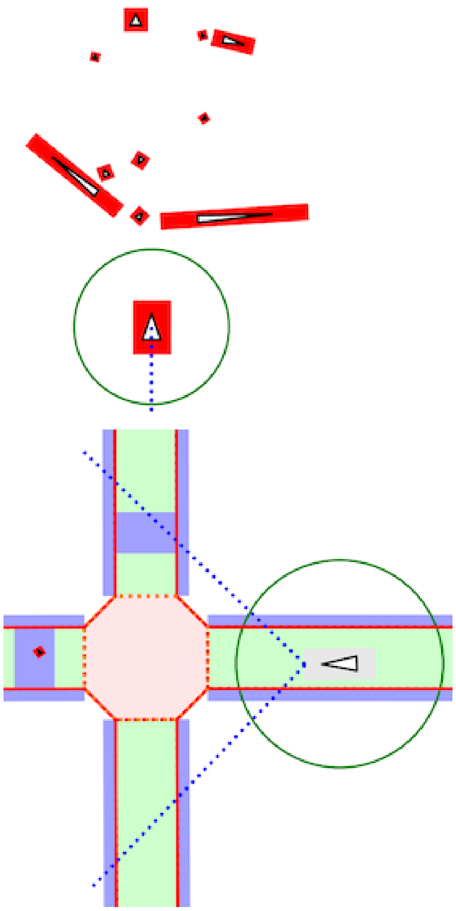


Here the circled rectangle is the ego car; its view cone extends to the right, where we see another car parked rather poorly at the side of the road (the white lines are curbs). If you close the window, Scenic will sample another scene from the same scenario and display it. This will repeat until you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

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

```
$ scenic examples/driving/pedestrian.scenic
$ scenic examples/webots/mars/narrowGoal.scenic
$ scenic examples/webots/road/crossing.scenic
```





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

1.1.3 Learning More

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

- If you want to start learning how to write Scenic programs, see the *Scenic Tutorial*.
- If you want to learn how to write dynamic scenarios in Scenic, see *Dynamic Scenarios*.
- If you want to use Scenic with a simulator, see *Supported Simulators* (which also describes how to interface Scenic to a new simulator, if the one you want isn't listed).
- If you want to 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].

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

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:

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

Here `road` is a *region*, one of Scenic's primitive types, defined in the `gta` model to specify which points in the workspace are on a road. Similarly, `roadDirection` is a *vector field* specifying the nominal traffic direction at such points. The operator `F at X` simply gets the direction of the field *F* at point *X*, so line 3 sets a `Car`'s default heading to be the road direction at its `position`. The default `position`, in turn, is a `Point on road` (we will explain this syntax shortly), which means a uniformly random point on the road. Thus, in our simple scenario above both cars will be placed on the road facing a reasonable direction, without our having to specify this explicitly.

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

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

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

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

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

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

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

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

[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.

cally 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)
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:

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

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

1.2.4 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) * Range(10, 20) deg
Car left of spot by 0.25,
    facing badAngle relative to roadDirection
```

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

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 Range(-10, 10) @ Range(20, 40), with viewAngle 30 deg
require car2 can see ego
```

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

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

Requirements, called *observations* in other probabilistic programming languages, are very convenient for defining scenarios because they make it easy to restrict attention to particular cases of interest. Note how difficult it would be to write the scenario above without the `require` statement: when defining the ego car, we would have to somehow specify those positions where it is possible to put a roughly-oncoming car 20–40 meters ahead (for example, this is not possible on a one-way road). Instead, we can simply place `ego` uniformly over all roads and let Scenic work out how to condition the distribution so that the requirement is satisfied[2]. As this example illustrates, the ability to declaratively impose constraints gives Scenic greater versatility than purely-generative formalisms. Requirements also improve encapsulation by allowing us to restrict an existing scenario without altering it. For example:

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

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

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

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

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

1.2.6 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`).

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 `length` property (which corresponds to Scenic's *y* axis).

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

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

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

Next we pick a position for the bottleneck, requiring it to lie roughly on the way from the robot to its goal, and place a rock there.

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

Note how we define `bottleneck` as an `OrientedPoint`, with a range of possible orientations: this is to set up a local coordinate system for positioning the pipes making up the bottleneck. Specifically, we position two pipes of varying lengths on either side of the bottleneck, with their ends far enough apart for the robot to be able to pass between:

```
halfGapWidth = (1.2 * ego.width) / 2
leftEnd = OrientedPoint left of bottleneck by halfGapWidth,
                        facing Range(60, 120) deg relative to bottleneck
rightEnd = OrientedPoint right of bottleneck by halfGapWidth,
                         facing Range(-120, -60) deg relative to bottleneck
Pipe ahead of leftEnd, with length Range(1, 2)
Pipe ahead of rightEnd, with length Range(1, 2)
```

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

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

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

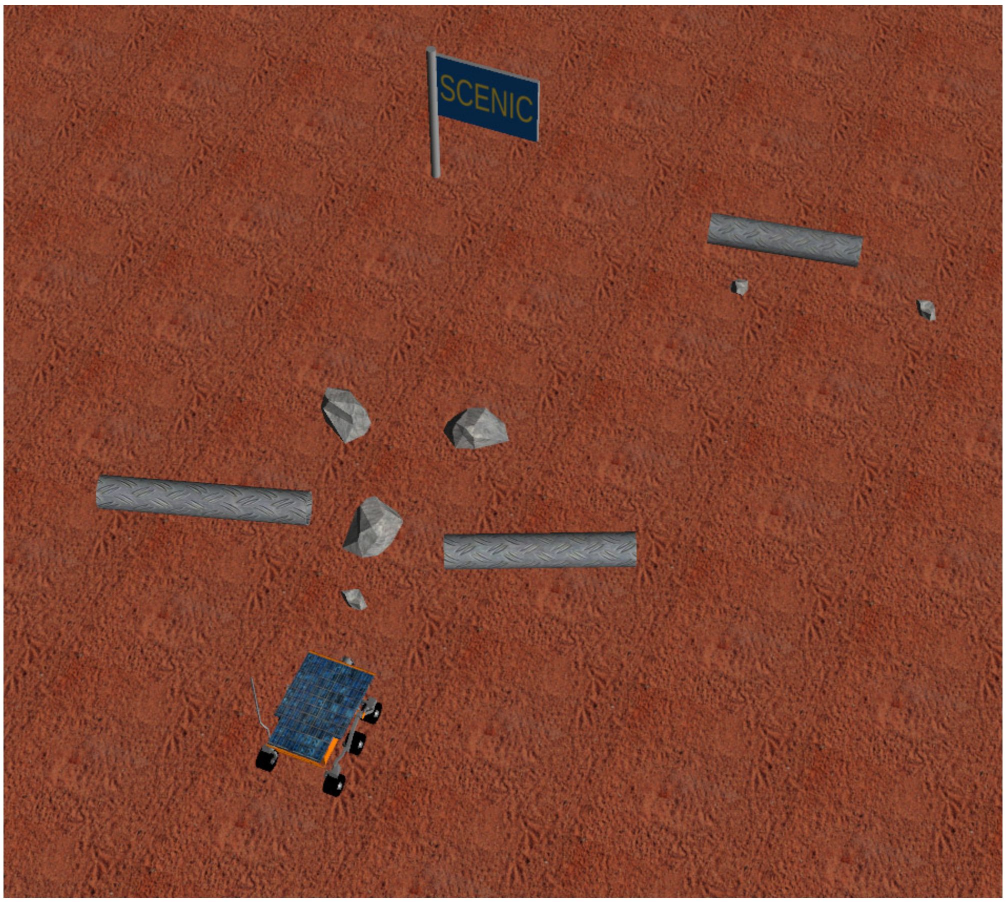
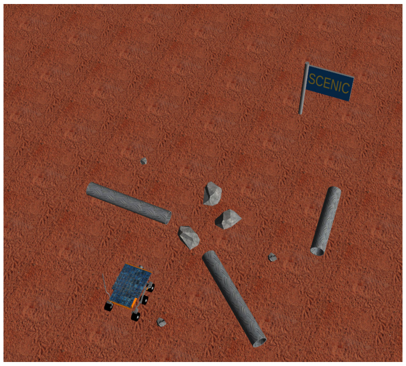
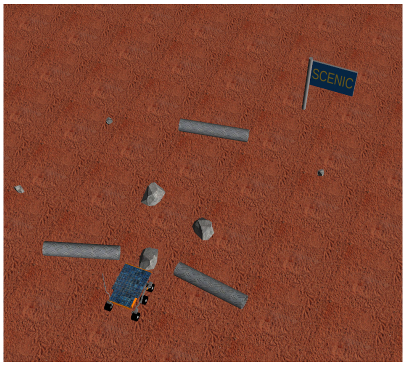
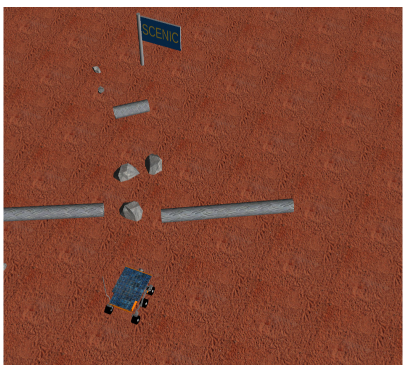


Fig. 3: A scene sampled from the Mars rover scenario, rendered in Webots.







1.2.8 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.

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

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

References

1.3 Dynamic Scenarios

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

1.3.1 Agents, Actions, and Behaviors

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

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

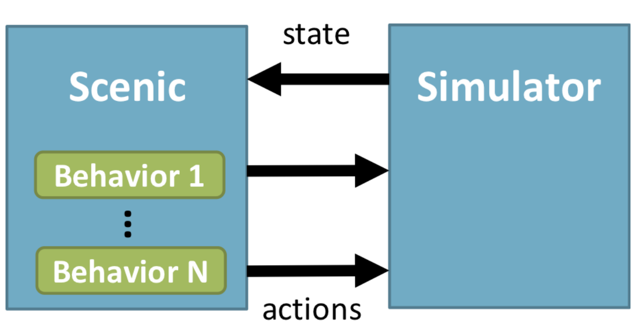
A behavior defines a sequence of *actions* for the agent to take, which need not be fixed but can be probabilistic and depend on the state of the agent or other objects. In Scenic, an action is an instantaneous operation executed by an agent, like setting the steering angle of a car or turning on its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

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

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

We intend this behavior to run for the entire scenario, so we use an infinite loop. In each step of the loop, we compute appropriate throttle and steering controls, then use the `take` statement to take the corresponding actions. When that statement is executed, Scenic pauses the behavior until the next time step of the simulation, when the function resumes and the loop repeats.

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



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

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

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

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

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

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

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

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

Here, the value of `threshold` is sampled only once, at the beginning of the scenario when the behavior starts running. The value `strength`, on the other hand, is sampled every time control reaches line 5, so that every time step when the car is braking we use a slightly different braking strength (0.8 on average, but with Gaussian noise added with standard deviation 0.02, truncating the possible values to between 0.5 and 1).

1.3.2 Interrupts

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
behavior DriveWithSuddenBrake():
    haveBraked = False
```

(continues on next page)

(continued from previous page)

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

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

1.3.3 Stateful Behaviors

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

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

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

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

1.3.4 Requirements and Monitors

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

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

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

```
require eventually ego in intersection
```

You can also use the ordinary `require` statement inside a behavior to require that a given condition hold at a certain point during the execution of the behavior. For example, here is a simple elaboration of the `WaitUntilClose` behavior we saw above:

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

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

To enforce more complex temporal properties like this one without modifying behaviors, you can define a *monitor*. Like behaviors, monitors are functions which run in parallel with the scenario, but they are not associated with any agent and any actions they take are ignored (so you might as well only use the `wait` statement). Here is a monitor for the property "`car1` and `car2` enter the intersection before `car3`":

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

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

1.3.5 Preconditions and Invariants

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

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

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

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

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

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

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

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

1.3.6 Terminating the Scenario

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

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

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

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

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

1.3.7 Trying Some Examples

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

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

```
$ scenic examples/driving/badlyParkedCarPullingIn.scenic
```

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

```
$ scenic examples/driving/badlyParkedCarPullingIn.scenic \
    --simulate \
    --model scenic.simulators.carla.model \
    --time 200
```

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

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

Try playing around with different example scenarios and different choices of maps (making sure that you keep the `map` and `lgsvl_map`/`carla_map` parameters consistent). For both CARLA and LGSVL, you don't have to restart the simulator between scenarios: just kill Scenic[1] and restart it with different arguments.

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

1.3.8 Further Reading

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

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

Supported Simulators Details on which simulator interfaces support dynamic scenarios.

1.4 Composing Scenarios

under construction. . .

1.5 Syntax Guide

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

1.5.1 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

1.5.2 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

1.5.3 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

1.5.4 Specifiers

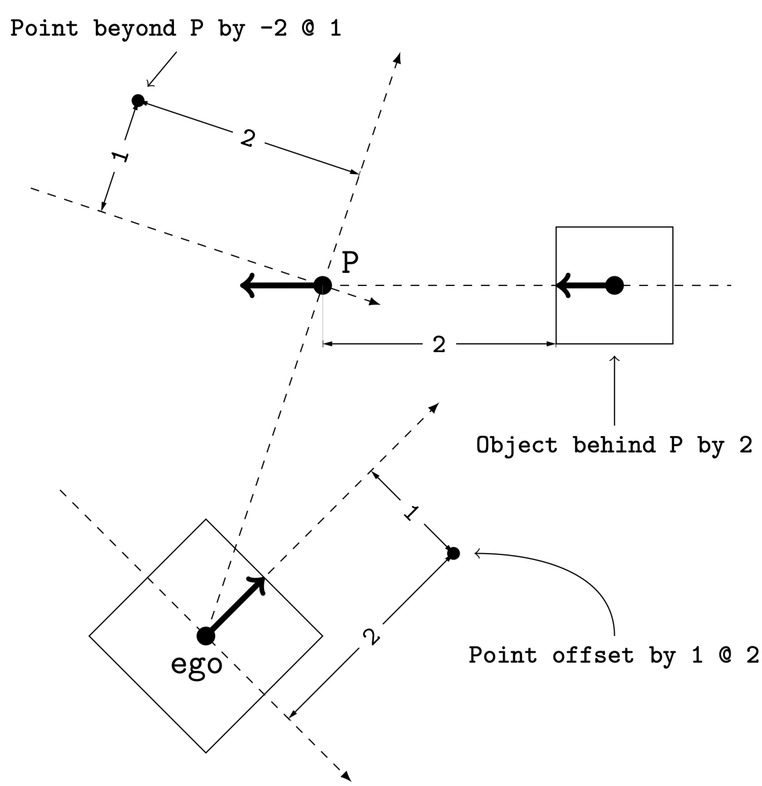


Fig. 4: Illustration of the `beyond`, `behind`, and `offset by` specifiers. Each `OrientedPoint` (e.g. `P`) is shown as a bold arrow.

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)

1.5.5 Operators

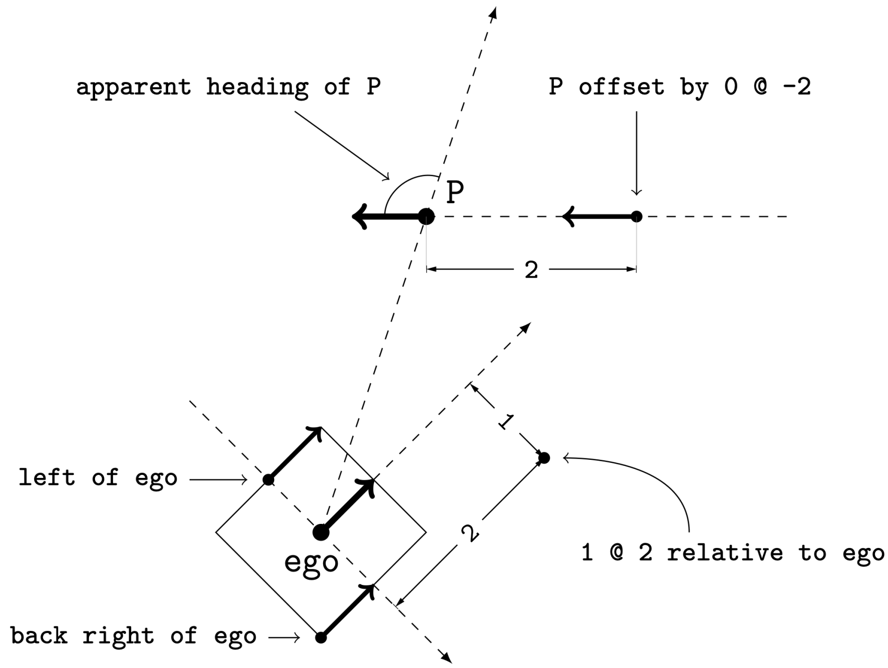


Fig. 5: Illustration of several operators. Each `OrientedPoint` (e.g. `P`) is shown as a bold arrow.

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

1.5.6 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

1.6 Syntax Reference

1.6.1 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.

1.6.2 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

1.6.3 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

1.6.4 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

1.6.5 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

1.6.6 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

1.6.7 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

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

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

1.7 Command-Line Options

The `scenic` command supports a variety of options. Run `scenic -h` for a full list with short descriptions; we elaborate on some of the most important options below.

1.7.1 General Scenario Control

-m `<model>`, **--model** `<model>`
Specify the world model to use for the scenario, overriding any `model` statement in the scenario. The argument must be the fully-qualified name of a Scenic module found on your `PYTHONPATH` (it does not necessarily need to be built into Scenic).

-p `<param> <value>`, **--param** `<param> <value>`
Specify the value of a global parameter. This assignment overrides any `param` statements in the scenario. If the given value can be interpreted as an int or float, it is; otherwise it is kept as a string.

1.7.2 Dynamic Simulations

-S, --simulate
Run dynamic simulations from scenes instead of plotting scene diagrams. This option will only work for scenarios which specify a simulator, which is done automatically by the world models for the simulator interfaces that support dynamic scenarios, e.g. *scenic.simulators.carla.model* and *scenic.simulators.lgsvl.model*. If your scenario is written for an abstract domain, like *scenic.domains.driving*, you will need to use the *--model* option to specify the specific model for the simulator you want to use.

--time `<steps>`
Maximum number of time steps to run each simulation (the default is infinity). Simulations may end earlier if termination criteria defined in the scenario are met.

--count `<number>`
Number of successful simulations to run (i.e., not counting rejected simulations). The default is to run forever.

1.7.3 Debugging

-v <verbosity>, **--verbosity** <verbosity>
Set the verbosity level, from 0 to 3 (default 1):

0 Nothing is printed except error messages and warnings (to `stderr`). Warnings can be suppressed using the `PYTHONWARNINGS` environment variable.

1 The main steps of compilation and scene generation are indicated, with timing statistics.

2 Additionally, details on which modules are being compiled and the reasons for any scene/simulation rejections are printed.

3 Additionally, the actions taken by each agent at each time step of a dynamic simulation are printed.

1.8 Developing Scenic

This page covers information useful if you will be developing Scenic, either changing the language itself or adding new built-in libraries or simulator interfaces.

1.8.1 Getting Started

Start by cloning our repository on GitHub and installing Poetry as described in *Getting Started with Scenic*. When using Poetry to install Scenic in your virtual environment, use the command **`poetry install -E dev`** to make sure you get all the dependencies needed for development.

1.8.2 Running the Test Suite

Scenic has an extensive test suite exercising most of the features of the language. We use the pytest Python testing tool. To run the entire test suite, run the command **`pytest`** inside the virtual environment.

Some of the tests are quite slow, e.g. those which test the parsing and construction of road networks. We add a `--fast` option to pytest which skips such tests, while still covering all of the core features of the language. So it is convenient to often run **`pytest --fast`** as a quick check, remembering to run the full **`pytest`** before making any final commits. You can also run specific parts of the test suite with a command like **`pytest tests/syntax/test_specifiers.py`**, or use pytest's `-k` option to filter by test name, e.g. **`pytest -k specifiers`**.

Note that many of Scenic's tests are probabilistic, so in order to reproduce a test failure you may need to set the random seed. We use the pytest-randomly plugin to help with this: at the beginning of each run of `pytest`, it prints out a line like:

```
Using --randomly-seed=344295085
```

Adding this as an option, i.e. running **`pytest --randomly-seed=344295085`**, will reproduce the same sequence of tests with the same Python/Scenic random seed.

1.9 Scenic Internals

This section of the documentation describes the implementation of Scenic. Much of this information will probably only be useful for people who need to make some change to the language (e.g. adding a new type of distribution). However, the detailed documentation on Scenic's abstract application domains (in `scenic.domains`) and simulator interfaces (in `scenic.simulators`) may be of interest to people using those features.

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

`scenic.core`	Scenic's core types and associated support code.
`scenic.domains`	General scenario domains used across simulators.
`scenic.formats`	Support for file formats not specific to particular simulators.
`scenic.simulators`	World models and interfaces for particular simulators.
`scenic.syntax`	The Scenic compiler and associated support code.

1.9.1 scenic.core

Scenic's core types and associated support code.

`distributions`	Objects representing distributions that can be sampled from.
`errors`	Common exceptions and error handling.
`external_params`	Support for values which are sampled outside of Scenic.
`geometry`	Utility functions for geometric computation.
`lazy_eval`	Support for lazy evaluation of expressions and specifiers.
`object_types`	Implementations of the built-in Scenic classes.
`pruning`	Pruning parts of the sample space which violate requirements.
`regions`	Objects representing regions in space.
`scenarios`	Scenario and scene objects.
`simulators`	Interface between Scenic and simulators.
`specifiers`	Specifiers and associated objects.
`type_support`	Support for checking Scenic types.
`utils`	Assorted utility functions.
`vectors`	Scenic vectors and vector fields.
`workspaces`	Workspaces.

scenic.core.distributions

Objects representing distributions that can be sampled from.

Summary of Module Members

Functions

`Uniform`	Uniform distribution over a finite list of options.
`canUnpackDistributions`	Whether the function supports iterable unpacking of distributions.
`dependencies`	Dependencies which must be sampled before this value.
`distributionFunction`	Decorator for wrapping a function so that it can take distributions as arguments.
`distributionMethod`	Decorator for wrapping a method so that it can take distributions as arguments.
`makeOperatorHandler`	
`monotonicDistributionFunction`	Like distributionFunction, but additionally specifies that the function is monotonic.
`needsSampling`	Whether this value requires sampling.
`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.
`unpacksDistributions`	Decorator indicating the function supports iterable unpacking of distributions.

Classes

`AttributeDistribution`	Distribution resulting from accessing an attribute of a distribution
`CustomDistribution`	Distribution with a custom sampler given by an arbitrary function
`DiscreteRange`	Distribution over a range of integers.
`Distribution`	Abstract class for distributions.
`FunctionDistribution`	Distribution resulting from passing distributions to a function
`MethodDistribution`	Distribution resulting from passing distributions to a method of a fixed object
`MultiplexerDistribution`	Distribution selecting among values based on another distribution.
`Normal`	Normal distribution
`OperatorDistribution`	Distribution resulting from applying an operator to one or more distributions
`Options`	Distribution over a finite list of options.
`Range`	Uniform distribution over a range
`Samplable`	Abstract class for values which can be sampled, possibly depending on other values.
`StarredDistribution`	A placeholder for the iterable unpacking operator * applied to a distribution.
`TruncatedNormal`	Truncated normal distribution.
`TupleDistribution`	Distributions over tuples (or namedtuples, or lists).
`UniformDistribution`	Uniform distribution over a variable number of options.

Exceptions

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

Member Details

`dependencies`(*thing*)
Dependencies which must be sampled before this value.

`needsSampling`(*thing*)
Whether this value requires sampling.

`supportInterval`(*thing*)
Lower and upper bounds on this value, if known.

`underlyingFunction`(*thing*)
Original function underlying a distribution wrapper.

`canUnpackDistributions`(*func*)
Whether the function supports iterable unpacking of distributions.

`unpacksDistributions`(*func*)
Decorator indicating the function supports iterable unpacking of distributions.

`exception RejectionException`
Bases: `Exception`

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

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

Abstract class for distributions.

defaultValueType
alias of `object`

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(**args*, ***kwargs*)
Bases: *scenic.core.distributions.Distribution*

Distribution with a custom sampler given by an arbitrary function

class TupleDistribution(**args*, ***kwargs*)
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(**args*, ***kwargs*)
Bases: *scenic.core.distributions.Distribution*

Distribution resulting from passing distributions to a function

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

monotonicDistributionFunction(*method*, *valueType=None*)
Like distributionFunction, but additionally specifies that the function is monotonic.

class StarredDistribution(**args*, ***kwargs*)
Bases: *scenic.core.distributions.Distribution*

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

class MethodDistribution(**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(**args*, ***kwargs*)
Bases: *scenic.core.distributions.Distribution*

Distribution resulting from accessing an attribute of a distribution

class OperatorDistribution(**args*, ***kwargs*)
Bases: *scenic.core.distributions.Distribution*

Distribution resulting from applying an operator to one or more distributions

class MultiplexerDistribution(**args*, ***kwargs*)
Bases: *scenic.core.distributions.Distribution*

Distribution selecting among values based on another distribution.

class Range(**args*, ***kwargs*)
Bases: *scenic.core.distributions.Distribution*

Uniform distribution over a range

class Normal(**args*, ***kwargs*)
Bases: *scenic.core.distributions.Distribution*

Normal distribution

class TruncatedNormal(**args*, ***kwargs*)
Bases: *scenic.core.distributions.Normal*

Truncated normal distribution.

class DiscreteRange(**args*, ***kwargs*)
Bases: *scenic.core.distributions.Distribution*

Distribution over a range of integers.

class Options(**args*, ***kwargs*)
Bases: *scenic.core.distributions.MultiplexerDistribution*

Distribution over a finite list of options.

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

Uniform(**opts*)
Uniform distribution over a finite list of options.

Implemented as an instance of *Options* when the set of options is known statically, and an instance of *UniformDistribution* otherwise.

class UniformDistribution(**args*, ***kwargs*)
Bases: *scenic.core.distributions.Distribution*

Uniform distribution over a variable number of options.

See *Options* for the more common uniform distribution over a fixed number of options. This class is for the special case where iterable unpacking is applied to a distribution, so that the number of options is unknown at compile time.

scenic.core.errors

Common exceptions and error handling.

Summary of Module Members

Module Attributes

showInternalBacktrace	Whether or not to elide Scenic's innards from backtraces.
postMortemDebugging	Whether or not to do post-mortem debugging of uncaught exceptions.
hiddenFolders	Folders elided from backtraces when showInternalBacktrace is false.

Functions

callBeginningScenicTrace	Call the given function, starting the Scenic backtrace at that point.
excepthook	
getText	
includeFrame	
saveErrorLocation	

Exceptions

ASTParseError	Parse error occuring during modification of the Python AST.
InconsistentScenarioError	Error for scenarios with inconsistent requirements.
InvalidScenarioError	Error raised for syntactically-valid but otherwise problematic Scenic programs.
PythonParseError	Parse error occurring during Python parsing or compilation.
RuntimeParseError	A Scenic parse error generated during execution of the translated Python.
ScenicError	An error produced during Scenic compilation, scene generation, or simulation.
ScenicSyntaxError	An error produced by attempting to parse an invalid Scenic program.
TokenParseError	Parse error occurring during token translation.

Member Details

showInternalBacktrace = False
Whether or not to elide Scenic's innards from backtraces.

Set to True by default so that any errors during import of the scenic module will get full backtraces; the `scenic` module's *__init__.py* sets it to False.

postMortemDebugging = False
Whether or not to do post-mortem debugging of uncaught exceptions.

hiddenFolders = [PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/scenic-lang/cl
Folders elided from backtraces when *showInternalBacktrace* is false.

exception ScenicError
Bases: `Exception`

An error produced during Scenic compilation, scene generation, or simulation.

exception ScenicSyntaxError
Bases: *scenic.core.errors.ScenicError*

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

This is intentionally not a subclass of SyntaxError so that pdb can be used for post-mortem debugging of the parser.

exception TokenParseError(*tokenOrLine*, *filename*, *message*)
Bases: *scenic.core.errors.ScenicSyntaxError*

Parse error occurring during token translation.

exception PythonParseError(*exc*)
Bases: *scenic.core.errors.ScenicSyntaxError*

Parse error occurring during Python parsing or compilation.

exception ASTParseError(*node*, *message*, *filename*)
Bases: *scenic.core.errors.ScenicSyntaxError*

Parse error occuring during modification of the Python AST.

exception RuntimeParseError(*msg*, *loc=None*)
Bases: *scenic.core.errors.ScenicSyntaxError*

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

exception InvalidScenarioError
Bases: *scenic.core.errors.ScenicError*

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

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

Error for scenarios with inconsistent requirements.

callBeginningScenicTrace(*func*)
Call the given function, starting the Scenic backtrace at that point.

This function is just a convenience to make Scenic backtraces cleaner when running Scenic programs from the command line.

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

Member Details

class ExternalSampler(*params*, *globalParams*)

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

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 (**args*, ***kwargs*)
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 (**args*, ***kwargs*)
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 (**args*, ***kwargs*)
Bases: *scenic.core.external_params.VerifaiParameter*

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

class VerifaiDiscreteRange (**args*, ***kwargs*)
Bases: *scenic.core.external_params.VerifaiParameter*

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

class VerifaiOptions (**args*, ***kwargs*)
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`
`checkPolygon`
`circumcircleOfAnnulus`
`cleanChain`
`cleanPolygon`
`cos`
`distanceToLine`
`distanceToSegment`
`findMinMax`
`headingOfSegment`
`hypot`
`max`
`min`
`normalizeAngle`
`plotPolygon`
`pointIsInCone`
`polygonUnion`
`positionRelativeToPoint`
`radialToCartesian`

continues on next page

Table 10 – continued from previous page

removeHoles	
rotateVector	
sin	
splitSelfIntersections	
subtractVectors	
triangulatePolygon	Triangulate the given Shapely polygon.
triangulatePolygon_mapbox	
triangulatePolygon_pypoly2tri	
viewAngleToPoint	

Exceptions

TriangulationError	Signals that the installed triangulation libraries are insufficient.

Member Details

exception TriangulationError

Bases: `RuntimeError`

Signals that the installed triangulation libraries are insufficient.

Specifically, raised when pypoly2tri hits the recursion limit trying to triangulate a large polygon.

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 use `mapbox_earcut` by default. If it is not installed, we allow fallback to `pypoly2tri` for historical reasons (we originally used the GPC library, which is not free for commercial use, falling back to `pypoly2tri` if not installed).

Parameters **polygon** (*shapely.geometry.Polygon*) – Polygon to triangulate.

Returns A list of disjoint (except for edges) triangles whose union is the original polygon.

class _RotatedRectangle

mixin providing collision detection for rectangular objects and regions

scenic.core.lazy_eval

Support for lazy evaluation of expressions and specifiers.

Summary of Module Members

Functions

makeDelayedFunctionCall	Utility function for creating a lazily-evaluated function call.
makeDelayedOperatorHandler	
needsLazyEvaluation	
requiredProperties	
valueInContext	Evaluate something in the context of an object being constructed.

Classes

DelayedArgument	Specifier arguments requiring other properties to be evaluated first.
LazilyEvaluable	Values which may require evaluation in the context of an object being constructed.

Member Details

class LazilyEvaluable(*requiredProps*)
Values which may require evaluation in the context of an object being constructed.

If a LazilyEvaluable specifies any properties it depends on, then it cannot be evaluated to a normal value except during the construction of an object which already has values for those properties.

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.

static makeContext(***props*)
Make a context with the given properties for testing purposes.

class DelayedArgument(**args*, *_internal=False*, ***kwargs*)
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

Functions

`disableDynamicProxyFor`
`enableDynamicProxyFor`
`setDynamicProxyFor`

Classes

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 base class `Point`.
PositionMutator	Mutator adding Gaussian noise to `position`.

Member Details

class Mutator
An object controlling how the `mutate` statement affects an *Object*.

A *Mutator* can be assigned to the `mutator` property of an *Object* to control the effect of the `mutate` statement. When mutation is enabled for such an object using that statement, the mutator's *appliedTo* method is called to compute a mutated version.

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 (<*specifiers*>)

Implementation of the Scenic base class `Point`.

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

Properties

- **position** (`Vector`; dynamic) – Position of the point. Default value is the origin.
- **visibleDistance** (*float*) – Distance for `can see` operator. Default value 50.
- **width** (*float*) – Default value zero (only provided for compatibility with operators that expect an `Object`).
- **length** (*float*) – Default value zero.

Note: If you're looking into Scenic's internals, note that `Point` is actually a subclass of the internal Python class `_Constructible`.

---

class OrientedPoint (<*specifiers*>)

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`.

Properties

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

class Object (<*specifiers*>)

Bases: `scenic.core.object_types.OrientedPoint`

Implementation of the Scenic class `Object`.

This is the default base class for Scenic classes.

Properties

- **width** (*float*) – Width of the object, i.e. extent along its X axis. Default value 1.
- **length** (*float*) – Length of the object, i.e. extent along its Y axis. Default value 1.
- **allowCollisions** (*bool*) – Whether the object is allowed to intersect other objects. Default value `False`.
- **requireVisible** (*bool*) – Whether the object is required to be visible from the `ego` object. Default value `True`.
- **regionContainedIn** (`Region` or `None`) – A `Region` the object is required to be contained in. If `None`, the object need only be contained in the scenario's workspace.
- **cameraOffset** (`Vector`) – Position of the camera for the `can see` operator, relative to the object's `position`. Default `0 @ 0`.
- **speed** (*float; dynamic*) – Speed in dynamic simulations. Default value 0.

- **velocity** (*Vector*; *dynamic*) – Velocity in dynamic simulations. Default value is the velocity determined by `self.speed` and `self.heading`.
- **angularSpeed** (*float; *dynamic**) – Angular speed in dynamic simulations. Default value 0.
- **behavior** – Behavior for dynamic agents, if any (see *Dynamic Scenarios*). Default value `None`.

class _Constructible(**args*, *_internal=False*, ***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.

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	
DifferenceRegion	
EmptyRegion	Region containing no points.
GridRegion	A Region given by an obstacle grid.
IntersectionRegion	
PointInRegionDistribution	Uniform distribution over points in a Region
PointSetRegion	Region consisting of a set of discrete points.
PolygonalRegion	Region given by one or more polygons (possibly with holes)
PolylineRegion	Region given by one or more polylines (chain of line segments)
RectangularRegion	
Region	Abstract class for regions.
SectorRegion	

Member Details

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

Build a 'Region' from Shapely geometry.

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

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.

intersects (*other*)
Check if this `Region` intersects another.

difference (*other*)
Get a `Region` representing the difference of this one and another.

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

Not supported by all region types.

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*, *name=None*)
Bases: `scenic.core.regions.Region`

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

signedDistanceTo (*point*)
Compute the signed distance of the PolylineRegion to a point.

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

class PolygonalRegion (*points=None*, *polygon=None*, *orientation=None*, *name=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

A scene generated from a Scenic scenario.

Attributes

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

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

Render a schematic of the scene for debugging.

class Scenario

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

generate(*maxIterations=2000*, *verbosity=0*, *feedback=None*)

Sample a `Scene` from this scenario.

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.simulators

Interface between Scenic and simulators.

Summary of Module Members

Classes

`Action`	An action which can be taken by an agent for one step of a simulation.
`DummySimulation`	
`DummySimulator`	Simulator which does nothing, for debugging purposes.
`EndScenarioAction`	Special action indicating it is time to end the current scenario.

continues on next page

Table 20 – continued from previous page

`EndSimulationAction`	Special action indicating it is time to end the simulation.
`Simulation`	A single simulation run, possibly in progress.
`SimulationResult`	Result of running a simulation.
`Simulator`	A simulator which can import/execute scenes from Scenic.

Exceptions

`RejectSimulationException`	Exception indicating a requirement was violated at runtime.
`SimulationCreationError`	Exception indicating a simulation could not be run from the given scene.

Member Details

exception SimulationCreationError

Bases: `Exception`

Exception indicating a simulation could not be run from the given scene.

Can also be issued during a simulation if dynamic object creation fails.

exception RejectSimulationException

Bases: `Exception`

Exception indicating a requirement was violated at runtime.

class Simulator

A simulator which can import/execute scenes from Scenic.

simulate(*scene*, *maxSteps=None*, *maxIterations=100*, *verbosity=0*, *raiseGuardViolations=False*)

Run a simulation for a given scene.

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

A single simulation run, possibly in progress.

run(*maxSteps*)

Run the simulation.

Throws a RejectSimulationException if a requirement is violated.

createObject(*obj*)

Dynamically create an object.

createObjectInSimulator(*obj*)

Create the given object in the simulator.

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

scheduleForAgents()

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

actionsAreCompatible(*agent*, *actions*)

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

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

executeActions(*allActions*)
Execute the actions selected by the agents.

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

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

updateObjects()
Update the positions and other properties of objects from the simulation.

getProperties(*obj*, *properties*)
Read the values of the given properties of the object from the simulation.

currentState()
Return the current state of the simulation.

The definition of 'state' is up to the simulator; the 'state' is simply saved at each time step to define the 'trajectory' of the simulation.

The default implementation returns a tuple of the positions of all objects.

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

class DummySimulator(*timestep=1*)
Bases: `scenic.core.simulators.Simulator`

Simulator which does nothing, for debugging purposes.

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

class EndSimulationAction(*line*)
Bases: `scenic.core.simulators.Action`

Special action indicating it is time to end the simulation.

Only for internal use.

class EndScenarioAction(*line*)
Bases: `scenic.core.simulators.Action`

Special action indicating it is time to end the current scenario.

Only for internal use.

class SimulationResult(*trajectory*, *actions*, *terminationReason*)
Result of running a simulation.

scenic.core.specifiers

Specifiers and associated objects.

Summary of Module Members

Classes

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

Member Details

class Specifier(*prop*, *value*, *deps=None*, *optionals={}*, *internal=False*)
Specifier providing a value for a property given dependencies.

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

applyTo(*obj*, *optionals*)
Apply specifier to an object, including the specified optional properties.

class PropertyDefault(*requiredProperties*, *attributes*, *value*)
A default value, possibly with dependencies.

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

scenic.core.type_support

Support for checking Scenic types.

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.
coerceToBehavior	
coerceToFloat	
coerceToHeading	
coerceToVector	

continues on next page

Table 23 – 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.
TypecheckedDistribution	

Exceptions

CoercionFailure

Member Details

class Heading(*x=0, /*)

Bases: `float`

Dummy class used as a target for type coercions to headings.

underlyingType(*thing*)

What type this value ultimately evaluates to, if we can tell.

isA(*thing, ty*)

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

unifyingType(*opts*)

Most specific type unifying the given types.

canCoerceType (*typeA*, *typeB*)
Can values of typeA be coerced into typeB?

canCoerce (*thing*, *ty*)
Can this value be coerced into the given type?

coerce (*thing*, *ty*, *error='wrong type'*)
Coerce something into the given type.

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 (**args*, *_internal=False*, ***kwargs*)
Bases: `scenic.core.lazy_eval.DelayedArgument`

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

class TypeEqualityChecker (**args*, *_internal=False*, ***kwargs*)
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.

Summary of Module Members

Functions

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

continues on next page

Table 26 – continued from previous page

cached_property	
get_type_args	
get_type_origin	

Classes

DefaultIdentityDict	Dictionary which is the identity map by default.

Member Details

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

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!

class DefaultIdentityDict
Dictionary which is the identity map by default.

The map works on all objects, even unhashable ones, but doesn't support all of the standard mapping operations.

scenic.core.vectors

Scenic vectors and vector fields.

Summary of Module Members

Functions

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

Member Details

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

Bases: *scenic.core.distributions.Distribution*

A distribution over Vectors.

defaultValueType

alias of *scenic.core.vectors.Vector*

class CustomVectorDistribution(**args*, ***kwargs*)

Bases: *scenic.core.vectors.VectorDistribution*

Distribution with a custom sampler given by an arbitrary function.

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

Bases: *scenic.core.vectors.VectorDistribution*

Vector version of OperatorDistribution.

class VectorMethodDistribution(**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.

Return type *scenic.core.vectors.Vector*

angleWith(*other*)
Compute the signed angle between self and other.

The angle is positive if other is counterclockwise of self (considering the smallest possible rotation to align them).

Return type float

class VectorField(*name*, *value*, *minSteps=4*, *defaultStepSize=5*)
A vector field, providing a heading at every point.

Parameters

- **name** (*str*) – name for debugging.
- **value** – function computing the heading at the given *Vector*.
- **minSteps** (*int*) – Minimum number of steps for *followFrom*; default 4.
- **defaultStepSize** (*float*) – Default step size for *followFrom*; default 5.

followFrom(*pos*, *dist*, *steps=None*, *stepSize=None*)
Follow the field from a point for a given distance.

Uses the forward Euler approximation, covering the given distance with equal-size steps. The number of steps can be given manually, or computed automatically from a desired step size.

Parameters

- **pos** (*Vector*) – point to start from.
- **dist** (*float*) – distance to travel.
- **steps** (*int*) – number of steps to take, or `None` to compute the number of steps based on the distance (default `None`).
- **stepSize** (*float*) – length used to compute how many steps to take, or `None` to use the field's default step size.

static forUnionOf(*regions*)
Creates a *PiecewiseVectorField* from the union of the given regions.

If none of the regions have an orientation, returns `None` instead.

class PolygonalVectorField(*name*, *cells*, *headingFunction=None*, *defaultHeading=None*)
Bases: *scenic.core.vectors.VectorField*

A piecewise-constant vector field defined over polygonal cells.

Parameters

- **name** (*str*) – name for debugging.
- **cells** – a sequence of cells, with each cell being a pair consisting of a Shapely geometry and a heading. If the heading is `None`, we call the given **headingFunction** for points in the cell instead.
- **headingFunction** – function computing the heading for points in cells without specified headings, if any (default `None`).
- **defaultHeading** – heading for points not contained in any cell (default `None`, meaning reject such points).

class PiecewiseVectorField(*name*, *regions*, *defaultHeading=None*)
Bases: *scenic.core.vectors.VectorField*

A vector field defined by patching together several regions.

The heading at a point is determined by checking each region in turn to see if it has an orientation and contains the point, returning the corresponding heading if so. If we get through all the regions, then we return the **defaultHeading**, if any, and otherwise reject the scene.

Parameters

- **name** (*str*) – name for debugging.
- **regions** (sequence of *Region* objects) – the regions making up the field.
- **defaultHeading** (*float*) – the heading for points not in any region with an orientation (default None, meaning reject such points).

scenic.core.workspaces

Workspaces.

Summary of Module Members

Classes

Workspace	A workspace describing the fixed world of a scenario

Member Details

class Workspace(*region=<AllRegion everywhere>*)

Bases: *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=1*)

Zoom the schematic around the specified objects

scenicToSchematicCoords(*coords*)

Convert Scenic coordinates to those used for schematic rendering.

1.9.2 scenic.domains

General scenario domains used across simulators.

driving	Domain for driving scenarios.

scenic.domains.driving

Domain for driving scenarios.

The *world model* defines Scenic classes for cars, pedestrians, etc., actions for dynamic agents which walk or drive, as well as simple behaviors like lane-following. Scenarios for the driving domain should import the model as follows:

```
model scenic.domains.driving.model
```

Scenarios written for the driving domain should work without changes[1] in any of the following simulators:

- CARLA, using the model *scenic.simulators.carla.model*
- LGSVL, using the model *scenic.simulators.lgsvl.model*

For example, the `examples/driving/badlyParkedCarPullingIn.scenic` scenario is written for the driving domain and can be run in multiple simulators:

- no simulator, for debugging:

```
$ scenic examples/driving/badlyParkedCarPullingIn.scenic
```

- CARLA, using the default map specified in the scenario:

```
$ scenic -S --model scenic.simulators.carla.model \
    examples/driving/badlyParkedCarPullingIn.scenic
```

- LGSVL, specifying a map which it supports:

```
$ scenic -S --model scenic.simulators.lgsvl.model \
    --param map tests/formats/opendrive/maps/LGSVL/borregasave.xodr \
    --param lgsvl_map BorregasAve \
    examples/driving/badlyParkedCarPullingIn.scenic
```

`model`	Scenic world model for scenarios using the driving domain.
`behaviors`	Library of useful behaviors for dynamic agents in driving scenarios.
`actions`	Actions for dynamic agents in the driving domain.
`roads`	Library for representing road network geometry and traffic information.
`controllers`	Low-level controllers useful for vehicles.
`workspace`	Workspaces for the driving domain.

[1] Assuming the simulator supports the selected map. If necessary, the map may be changed from the command line using the `--param` option; see the *model documentation* for details.

scenic.domains.driving.model

Scenic world model for scenarios using the driving domain.

Imports actions and behaviors for dynamic agents from *scenic.domains.driving.actions* and *scenic.domains.driving.behaviors*.

The map file to use for the scenario must be specified before importing this model by defining the global parameter `map`. This path is passed to the `Network.fromFile` function to create a `Network` object representing the road network. Extra options may be passed to the function by defining the global parameter `map_options`, which should be a dictionary of keyword arguments. For example, we could write:

```
param map = localPath('mymap.xodr')
param map_options = { 'tolerance': 0.1 }
model scenic.domains.driving.model
```

If you are writing a generic scenario that supports multiple maps, you may leave the `map` parameter undefined; then running the scenario will produce an error unless the user uses the `--param` command-line option to specify the map.

Note: If you are using a simulator, you may have to also define simulator-specific global parameters to tell the simulator which world to load. For example, our LGSVL interface uses a parameter `lgsvl_map` to specify the name of the Unity scene. See the *documentation* of the simulator interfaces for details.

Summary of Module Members

Module Attributes

`network`	The road network being used for the scenario, as a `Network` object.
`road`	The union of all drivable roads, including intersections but not shoulders or parking lanes.
`curb`	The union of all curbs.
`sidewalk`	The union of all sidewalks.
`shoulder`	The union of all shoulders, including parking lanes.
`roadOrShoulder`	All drivable areas, including both ordinary roads and shoulders.
`intersection`	The union of all intersections.
`roadDirection`	A `VectorField` representing the nominal traffic direction at a given point.

Functions

`withinDistanceToAnyCars`	returns boolean
`withinDistanceToAnyObjs`	checks whether there exists any obj (1) in front of the vehicle, (2) within thresholdDistance
`withinDistanceToObjsInLane`	checks whether there exists any obj (1) in front of the vehicle, (2) on the same lane, (3) within thresholdDistance

Classes

Car	A car.
DrivingObject	Abstract class for objects in a road network.
NPCCar	Car for which accurate physics is not required.
Pedestrian	A pedestrian.
Steers	Mixin protocol for agents which can steer.
Vehicle	Vehicles which drive, such as cars.
Walks	Mixin protocol for agents which can walk with a given direction and speed.

Member Details

network: ***scenic.domains.driving.roads.Network***
The road network being used for the scenario, as a *Network* object.

road: ***scenic.core.regions.Region***
The union of all drivable roads, including intersections but not shoulders or parking lanes.

curb: ***scenic.core.regions.Region***
The union of all curbs.

sidewalk: ***scenic.core.regions.Region***
The union of all sidewalks.

shoulder: ***scenic.core.regions.Region***
The union of all shoulders, including parking lanes.

roadOrShoulder: ***scenic.core.regions.Region***
All drivable areas, including both ordinary roads and shoulders.

intersection: ***scenic.core.regions.Region***
The union of all intersections.

roadDirection: ***scenic.core.vectors.VectorField***
A `VectorField` representing the nominal traffic direction at a given point.

Inside intersections or anywhere else where there can be multiple nominal traffic directions, the choice is arbitrary. At such points, the function *Network.nominalDirectionsAt* can be used to get all nominal directions.

class DrivingObject(*<specifiers>*)
Bases: *scenic.core.object_types.Object*

Abstract class for objects in a road network.

Provides convenience properties for the lane, road, intersection, etc. at the object's current position (if any).

Also defines the `elevation` property as a standard way to access the Z component of an object's position, since the Scenic built-in property `position` is only 2D. If `elevation` is set to `None`, the simulator is responsible for choosing an appropriate Z coordinate so that the object is on the ground, then updating the property. 2D simulators should set the property to zero.

Properties

- **elevation** (*float or None; dynamic*) – default `None` (see above).
- **requireVisible** (*bool*) – Default value `False` (overriding the default from *Object*).

property lane
The *Lane* at the object's current position.

The simulation is rejected if the object is not in a lane.

property _lane
The *Lane* at the object's current position, if any.

property laneSection
The *LaneSection* at the object's current position.

The simulation is rejected if the object is not in a lane.

property _laneSection
The *LaneSection* at the object's current position, if any.

property laneGroup
The *LaneGroup* at the object's current position.

The simulation is rejected if the object is not in a lane.

property _laneGroup
The *LaneGroup* at the object's current position, if any.

property oppositeLaneGroup
The *LaneGroup* on the other side of the road from the object.

The simulation is rejected if the object is not on a two-way road.

property road
The *Road* at the object's current position.

The simulation is rejected if the object is not on a road.

property _road
The *Road* at the object's current position, if any.

property intersection
The *Intersection* at the object's current position.

The simulation is rejected if the object is not in an intersection.

property _intersection
The *Intersection* at the object's current position, if any.

property crossing
The *PedestrianCrossing* at the object's current position.

The simulation is rejected if the object is not in a crosswalk.

property _crossing
The *PedestrianCrossing* at the object's current position, if any.

property element
The highest-level *NetworkElement* at the object's current position.

See *Network.elementAt* for the details of how this is determined. The simulation is rejected if the object is not in any network element.

property _element
The highest-level *NetworkElement* at the object's current position, if any.

distanceToClosest(*type*)
Compute the distance to the closest object of the given type.

For example, one could write `self.distanceToClosest(Car)` in a behavior.

class Vehicle(<*specifiers*>)
Bases: `scenic.domains.driving.model.DrivingObject`

Vehicles which drive, such as cars.

Properties

- **position** – The default position is uniformly random over the `road`.
- **heading** – The default heading is aligned with `roadDirection`, plus an offset given by **roadDeviation**.
- **roadDeviation** (*float*) – Relative heading with respect to the road direction at the `Vehicle`'s position. Used by the default value for **heading**.
- **regionContainedIn** – The default container is `roadOrShoulder`.
- **viewAngle** – The default view angle is 90 degrees.
- **width** – The default width is 2 meters.
- **length** – The default length is 4.5 meters.
- **color** (`Color` or RGB tuple) – Color of the vehicle. The default value is a distribution derived from car color popularity statistics; see `Color.defaultCarColor`.

class Car(<*specifiers*>)
Bases: `scenic.domains.driving.model.Vehicle`

A car.

class NPCCar(<*specifiers*>)
Bases: `scenic.domains.driving.model.Car`

Car for which accurate physics is not required.

class Pedestrian(<*specifiers*>)
Bases: `scenic.domains.driving.model.DrivingObject`

A pedestrian.

Properties

- **position** – The default position is uniformly random over sidewalks and crosswalks.
- **heading** – The default heading is uniformly random.
- **viewAngle** – The default view angle is 90 degrees.
- **width** – The default width is 0.75 m.
- **length** – The default length is 0.75 m.
- **color** – The default color is turquoise. Pedestrian colors are not necessarily used by simulators, but do appear in the debugging diagram.

class Steers
Bases: `abc.ABC`

Mixin protocol for agents which can steer.

Specifically, agents must support throttling, braking, steering, setting the hand brake, and going into reverse.

class Walks
Bases: `abc.ABC`

Mixin protocol for agents which can walk with a given direction and speed.

We provide a simplistic implementation which directly sets the velocity of the agent. This implementation needs to be explicitly opted-into, since simulators may provide a more sophisticated API that properly animates pedestrians.

withinDistanceToAnyCars (*car*, *thresholdDistance*)
returns boolean

withinDistanceToAnyObjs (*vehicle*, *thresholdDistance*)
checks whether there exists any obj (1) in front of the vehicle, (2) within thresholdDistance

withinDistanceToObjsInLane (*vehicle*, *thresholdDistance*)
checks whether there exists any obj (1) in front of the vehicle, (2) on the same lane, (3) within thresholdDistance

scenic.domains.driving.behaviors

Library of useful behaviors for dynamic agents in driving scenarios.

These behaviors are automatically imported when using the driving domain.

Summary of Module Members

Functions

concatenateCenterlines
setLaneChangingPIDControllers
setLaneFollowingPIDControllers
setTurnPIDControllers

Classes

AccelerateForwardBehavior	
ConstantThrottleBehavior	
DriveAvoidingCollisions	
FollowLaneBehavior	Follow's the lane on which the vehicle is at, unless the laneToFollow is specified.
FollowTrajectoryBehavior	Follows the given trajectory.
LaneChangeBehavior	is_oppositeTraffic should be specified as True only if the laneSectionToSwitch to has the opposite traffic direction to the initial lane from which the vehicle started LaneChangeBehavior e.g.
TurnBehavior	This behavior uses a PID controller specifically tuned for turning at an intersection.
WalkForwardBehavior	Walk forward behavior for pedestrians.

Member Details

class WalkForwardBehavior(**args*, ***kwargs*)
Bases: `scenic.core.dynamics.Behavior`

Walk forward behavior for pedestrians.

It will uniformly randomly choose either end of the sidewalk that the pedestrian is on, and have the pedestrian walk towards the endpoint.

class FollowLaneBehavior(**args*, ***kwargs*)
Bases: `scenic.core.dynamics.Behavior`

Follow's the lane on which the vehicle is at, unless the laneToFollow is specified. Once the vehicle reaches an intersection, by default, the vehicle will take the straight route. If straight route is not available, then any availble turn route will be taken, uniformly randomly. If turning at the intersection, the vehicle will slow down to make the turn, safely, and resume initial speed upon exiting the intersection.

This behavior does not terminate. A recommended use of the behavior is to accompany it with condition, e.g. do FollowLaneBehavior() until . . .

Parameters

- **target_speed** – Its unit is in m/s. By default, it is set to 10 m/s
- **laneToFollow** – If the lane to follow is different from the lane that the vehicle is on, this parameter can be used to specify that lane. By default, this variable will be set to None, which means that the vehicle will follow the lane that it is currently on.

class FollowTrajectoryBehavior(**args*, ***kwargs*)
Bases: `scenic.core.dynamics.Behavior`

Follows the given trajectory. The behavior terminates once the end of the trajectory is reached.

Parameters

- **target_speed** – Its unit is in m/s. By default, it is set to 10 m/s
- **trajectory** – It is a list of sequential lanes to track, from the lane that the vehicle is initially on to the lane it should end up on.

class TurnBehavior(**args*, ***kwargs*)
Bases: `scenic.core.dynamics.Behavior`

This behavior uses a PID controller specifically tuned for turning at an intersection. This behavior is only operational within an intersection, it will terminate if the vehicle is outside of an intersection.

class LaneChangeBehavior(**args*, ***kwargs*)
Bases: `scenic.core.dynamics.Behavior`

is_oppositeTraffic should be specified as True only if the laneSectionToSwitch to has the opposite traffic direction to the initial lane from which the vehicle started LaneChangeBehavior e.g. refer to the use of this flag in examples/carla/Carla_Challenge/carlaChallenge6.scenic

scenic.domains.driving.actions

Actions for dynamic agents in the driving domain.

These actions are automatically imported when using the driving domain.

The *RegulatedControlAction* is based on code from the CARLA project, licensed under the following terms:

> Copyright (c) 2018-2020 CVC.
>
> This work is licensed under the terms of the MIT license. For a copy, see <https://opensource.org/licenses/MIT>.

Summary of Module Members

Classes

OffsetAction	Teleports actor forward (in direction of its heading) by some offset.
RegulatedControlAction	Regulated control of throttle, braking, and steering.
SetBrakeAction	Set the amount of brake.
SetHandBrakeAction	Set or release the hand brake.
SetPositionAction	Teleport an agent to the given position.
SetReverseAction	Engage or release reverse gear.
SetSpeedAction	Set the speed of an agent (keeping its heading fixed).
SetSteerAction	Set the steering 'angle'.
SetThrottleAction	Set the throttle.
SetVelocityAction	Set the velocity of an agent.
SetWalkingDirectionAction	Set the walking direction.
SetWalkingSpeedAction	Set the walking speed.
SteeringAction	Abstract class for actions usable by agents which can steer.
WalkingAction	Abstract class for actions usable by agents which can walk.

Member Details

class SetPositionAction(*pos*)

Bases: *scenic.core.simulators.Action*

Teleport an agent to the given position.

class OffsetAction(*offset*)

Bases: *scenic.core.simulators.Action*

Teleports actor forward (in direction of its heading) by some offset.

class SetVelocityAction(*xVel*, *yVel*, *zVel=0*)

Bases: *scenic.core.simulators.Action*

Set the velocity of an agent.

class SetSpeedAction(*speed*)

Bases: *scenic.core.simulators.Action*

Set the speed of an agent (keeping its heading fixed).

class SteeringAction

Bases: *scenic.core.simulators.Action*

Abstract class for actions usable by agents which can steer.

Such agents must implement the *Steers* protocol.

class SetThrottleAction(*throttle*)

Bases: *scenic.domains.driving.actions.SteeringAction*

Set the throttle.

Parameters **throttle** – Throttle value between 0 and 1.

class SetSteerAction(*steer*)

Bases: *scenic.domains.driving.actions.SteeringAction*

Set the steering 'angle'.

Parameters **steer** – Steering 'angle' between -1 and 1.

class SetBrakeAction(*brake*)

Bases: *scenic.domains.driving.actions.SteeringAction*

Set the amount of brake.

Parameters **brake** – Amount of braking between 0 and 1.

class SetHandBrakeAction(*handBrake*)

Bases: *scenic.domains.driving.actions.SteeringAction*

Set or release the hand brake.

Parameters **handBrake** – Whether or not the hand brake is set.

class SetReverseAction(*reverse*)

Bases: *scenic.domains.driving.actions.SteeringAction*

Engage or release reverse gear.

Parameters **reverse** – Whether or not the car is in reverse.

class RegulatedControlAction(*throttle*, *steer*, *past_steer*, *max_throttle=0.5*, *max_brake=0.5*, *max_steer=0.8*)

Bases: *scenic.domains.driving.actions.SteeringAction*

Regulated control of throttle, braking, and steering.

Controls throttle and braking using one signal that may be positive or negative. Useful with simple controllers that output a single value.

Parameters

- **throttle** – Control signal for throttle and braking (will be clamped as below).
- **steer** – Control signal for steering (also clamped).
- **past_steer** – Previous steering signal, for regulating abrupt changes.
- **max_throttle** – Maximum value for **throttle**, when positive.
- **max_brake** – Maximum (absolute) value for **throttle**, when negative.
- **max_steer** – Maximum absolute value for **steer**.

class WalkingAction

Bases: *scenic.core.simulators.Action*

Abstract class for actions usable by agents which can walk.

Such agents must implement the *Walks* protocol.

class SetWalkingDirectionAction(*heading*)

Bases: *scenic.domains.driving.actions.WalkingAction*

Set the walking direction.

class SetWalkingSpeedAction(*speed*)

Bases: *scenic.domains.driving.actions.WalkingAction*

Set the walking speed.

scenic.domains.driving.roads

Library for representing road network geometry and traffic information.

A road network is represented by an instance of the *Network* class, which can be created from a map file using *Network.fromFile*.

Note: This library is a prototype under active development. We will try not to make backwards-incompatible changes, but the API may not be entirely stable. Some network information, such as traffic signals, has not yet been made available.

Summary of Module Members

Module Attributes

Vectorlike	Alias for types which can be interpreted as positions in Scenic.

Classes

Intersection	An intersection where multiple roads meet.
Lane	A lane for cars, bicycles, or other vehicles.
LaneGroup	A group of parallel lanes with the same type and direction.
LaneSection	Part of a lane in a single *RoadSection*.
LinearElement	A part of a road network with (mostly) linear 1- or 2-way flow.
Maneuver	A maneuver which can be taken upon reaching the end of a lane.
ManeuverType	A type of *Maneuver*, e.g., going straight or turning left.
Network	A road network.
NetworkElement	Abstract class for part of a road network.

continues on next page

Table 40 – continued from previous page

PedestrianCrossing	A pedestrian crossing (crosswalk).
Road	A road consisting of one or more lanes.
RoadSection	Part of a road with a fixed number of lanes.
Shoulder	A shoulder of a road, including parking lanes by default.
Sidewalk	A sidewalk.
Signal	Traffic lights, stop signs, etc.
VehicleType	A type of vehicle, including pedestrians.

Member Details

Vectorlike

Alias for types which can be interpreted as positions in Scenic.

This includes instances of *Point* and *Object*, and pairs of numbers.

alias of Union[scenic.core.vectors.Vector, scenic.core.object_types.Point, Tuple[numbers.Real, numbers.Real]]

class VehicleType(*value*)

Bases: `enum.Enum`

A type of vehicle, including pedestrians. Used to classify lanes.

class ManeuverType(*value*)

Bases: `enum.Enum`

A type of *Maneuver*, e.g., going straight or turning left.

STRAIGHT = 1

Straight, including one lane merging into another.

LEFT_TURN = 2

Left turn.

RIGHT_TURN = 3

Right turn.

U_TURN = 4

U-turn.

static guessTypeFromLanes(*start*, *end*, *connecting*, *turnThreshold=0.3490658503988659*)

For formats lacking turn information, guess it from the geometry.

Parameters

- **start** (`scenic.domains.driving.roads.Lane`) – starting lane of the maneuver.
- **end** (`scenic.domains.driving.roads.Lane`) – ending lane of the maneuver.
- **connecting** (*Optional[*`scenic.domains.driving.roads.Lane`*]*) – connecting lane of the maneuver, if any.
- **turnThreshold** (*float*) – angle beyond which to consider a maneuver a turn.

class Maneuver

A maneuver which can be taken upon reaching the end of a lane.

type: ***scenic.domains.driving.roads.ManeuverType***

type of maneuver (straight, left turn, etc.)

startLane: *scenic.domains.driving.roads.Lane*
: starting lane of the maneuver

endLane: *scenic.domains.driving.roads.Lane*
: ending lane of the maneuver

connectingLane: Optional[*scenic.domains.driving.roads.Lane*]
: connecting lane from the start to the end lane, if any (`None` for lane mergers)

intersection: Optional[*scenic.domains.driving.roads.Intersection*]
: intersection where the maneuver takes place, if any (`None` for lane mergers)

property conflictingManeuvers
: Maneuvers whose connecting lanes intersect this one's.

  Type Tuple[*Maneuver*]

class NetworkElement
Bases: `scenic.core.regions.PolygonalRegion`

Abstract class for part of a road network.

Includes roads, lane groups, lanes, sidewalks, pedestrian crossings, and intersections.

This is a subclass of `Region`, so you can do things like `Car in lane` or `Car on road` if `lane` and `road` are elements, as well as computing distances to an element, etc.

name: str
: Human-readable name, if any.

uid: str
: Unique identifier; from underlying format, if possible. (In OpenDRIVE, for example, ids are not necessarily unique, so we invent our own.)

id: Optional[str]
: Identifier from underlying format, if any.

network: *scenic.domains.driving.roads.Network*
: Link to parent network.

vehicleTypes: FrozenSet[*scenic.domains.driving.roads.VehicleType*]
: Which types of vehicles (car, bicycle, etc.) can be here.

speedLimit: Optional[float]
: Optional speed limit, which may be inherited from parent.

tags: FrozenSet[str]
: Uninterpreted semantic tags, e.g. 'roundabout'.

nominalDirectionsAt(*point*)
: Get nominal traffic direction(s) at a point in this element.

  There must be at least one such direction. If there are multiple, we pick one arbitrarily to be the orientation of the element as a `Region`. (So `Object in element` will align by default to that orientation.)

  Parameters **point** (`Vectorlike`) –

  Return type Tuple[float]

class LinearElement
Bases: `scenic.domains.driving.roads.NetworkElement`

A part of a road network with (mostly) linear 1- or 2-way flow.

Includes roads, lane groups, lanes, sidewalks, and pedestrian crossings, but not intersections.

LinearElements have a direction, namely from the first point on their centerline to the last point. This is called 'forward', even for 2-way roads. The 'left' and 'right' edges are interpreted with respect to this direction.

The left/right edges are oriented along the direction of traffic near them; so for 2-way roads they will point opposite directions.

flowFrom(*point*, *distance*, *steps=None*, *stepSize=5*)

Advance a point along this element by a given distance.

Equivalent to `follow element.orientation from point for distance`, but possibly more accurate. The default implementation uses the forward Euler approximation with a step size of 5 meters; subclasses may ignore the **steps** and **stepSize** parameters if they can compute the flow exactly.

Parameters

- **point** (*Vectorlike*) – point to start from.
- **distance** (*float*) – distance to travel.
- **steps** (*Optional[int]*) – number of steps to take, or `None` to compute the number of steps based on the distance (default `None`).
- **stepSize** (*float*) – length used to compute how many steps to take, if **steps** is not specified (default 5 meters).

Return type *scenic.core.vectors.Vector*

class Road

Bases: *scenic.domains.driving.roads.LinearElement*

A road consisting of one or more lanes.

Lanes are grouped into 1 or 2 instances of *LaneGroup*:

- **forwardLanes**: the lanes going the same direction as the road
- **backwardLanes**: the lanes going the opposite direction

One of these may be None if there are no lanes in that direction.

Because of splits and mergers, the Lanes of a *Road* do not necessarily start or end at the same point as the *Road*. Such intermediate branching points cause the *Road* to be partitioned into multiple road sections, within which the configuration of lanes is fixed.

sectionAt(*point*, *reject=False*)

Get the *RoadSection* passing through a given point.

Parameters **point** (*Vectorlike*) –

Return type Optional[*scenic.domains.driving.roads.RoadSection*]

laneSectionAt(*point*, *reject=False*)

Get the *LaneSection* passing through a given point.

Parameters **point** (*Vectorlike*) –

Return type Optional[*scenic.domains.driving.roads.LaneSection*]

laneAt(*point*, *reject=False*)

Get the *Lane* passing through a given point.

Parameters **point** (*Vectorlike*) –

Return type Optional[*scenic.domains.driving.roads.Lane*]

laneGroupAt(*point*, *reject=False*)

Get the *LaneGroup* passing through a given point.

Parameters **point** (*Vectorlike*) –

Return type Optional[*scenic.domains.driving.roads.LaneGroup*]

crossingAt (*point*, *reject=False*)

Get the *PedestrianCrossing* passing through a given point.

Parameters **point** (*Vectorlike*) –

Return type Optional[*scenic.domains.driving.roads.PedestrianCrossing*]

shiftLanes (*point*, *offset*)

Find the point equivalent to this one but shifted over some # of lanes.

Parameters

- **point** (*Vectorlike*) –
- **offset** (*int*) –

Return type Optional[*scenic.core.vectors.Vector*]

class LaneGroup

Bases: *scenic.domains.driving.roads.LinearElement*

A group of parallel lanes with the same type and direction.

road: ***scenic.domains.driving.roads.Road***

Parent road.

lanes: Tuple[*scenic.domains.driving.roads.Lane*]

Lanes, partially ordered with lane 0 being closest to the curb.

curb: ***scenic.core.regions.PolylineRegion***

Region representing the associated curb, which is not necessarily adjacent if there are parking lanes or some other kind of shoulder.

_sidewalk: Optional[*scenic.domains.driving.roads.Sidewalk*]

Adjacent sidewalk, if any.

_shoulder: Optional[*scenic.domains.driving.roads.Shoulder*]

Adjacent shoulder, if any.

_opposite: Optional[*scenic.domains.driving.roads.LaneGroup*]

Opposite lane group of the same road, if any.

property sidewalk

The adjacent sidewalk; rejects if there is none.

property shoulder

The adjacent shoulder; rejects if there is none.

property opposite

The opposite lane group of the same road; rejects if there is none.

laneAt (*point*, *reject=False*)

Get the *Lane* passing through a given point.

Parameters **point** (*Vectorlike*) –

Return type Optional[*scenic.domains.driving.roads.Lane*]

class Lane

Bases: *scenic.domains.driving.roads.LinearElement*

A lane for cars, bicycles, or other vehicles.

sectionAt (*point*, *reject=False*)
Get the LaneSection passing through a given point.

Parameters **point** (*Vectorlike*) –

Return type Optional[*scenic.domains.driving.roads.LaneSection*]

class RoadSection
Bases: *scenic.domains.driving.roads.LinearElement*

Part of a road with a fixed number of lanes.

A RoadSection has a fixed number of lanes: when a lane begins or ends, we move to a new section (which will be the successor of the current one).

laneAt (*point*, *reject=False*)
Get the lane section passing through a given point.

Parameters **point** (*Vectorlike*) –

Return type Optional[*scenic.domains.driving.roads.LaneSection*]

class LaneSection
Bases: *scenic.domains.driving.roads.LinearElement*

Part of a lane in a single *RoadSection*.

Since the lane configuration in a *RoadSection* is fixed, a *LaneSection* can have at most one adjacent lane to left or right. These are accessible using the *laneToLeft* and *laneToRight* properties, which for convenience reject the simulation if the desired lane does not exist. If rejection is not desired (for example if you want to handle the case where there is no lane to the left yourself), you can use the *_laneToLeft* and *_laneToRight* properties instead.

lane: *scenic.domains.driving.roads.Lane*
Parent lane.

group: *scenic.domains.driving.roads.LaneGroup*
Grandparent lane group.

road: *scenic.domains.driving.roads.Road*
Great-grandparent road.

isForward: bool
Whether this lane has the same direction as its parent road.

adjacentLanes: Tuple[*scenic.domains.driving.roads.LaneSection*]
Adjacent lanes of the same type, if any.

_laneToLeft: Optional[*scenic.domains.driving.roads.LaneSection*]
Adjacent lane of same type to the left, if any.

_laneToRight: Optional[*scenic.domains.driving.roads.LaneSection*]
Adjacent lane of same type to the right, if any.

_fasterLane: Optional[*scenic.domains.driving.roads.LaneSection*]
Faster adjacent lane of same type, if any. Could be to left or right depending on the country.

_slowerLane: Optional[*scenic.domains.driving.roads.LaneSection*]
Slower adjacent lane of same type, if any.

property laneToLeft
The adjacent lane of the same type to the left; rejects if there is none.

property laneToRight
The adjacent lane of the same type to the right; rejects if there is none.

property fasterLane
The faster adjacent lane of the same type; rejects if there is none.

property slowerLane
The slower adjacent lane of the same type; rejects if there is none.

shiftedBy(*offset*)
Find the lane a given number of lanes over from this lane.

Parameters **offset** (*int*) –

Return type Optional[*scenic.domains.driving.roads.LaneSection*]

class Sidewalk
Bases: `scenic.domains.driving.roads.LinearElement`

A sidewalk.

class PedestrianCrossing
Bases: `scenic.domains.driving.roads.LinearElement`

A pedestrian crossing (crosswalk).

class Shoulder
Bases: `scenic.domains.driving.roads.LinearElement`

A shoulder of a road, including parking lanes by default.

class Intersection
Bases: `scenic.domains.driving.roads.NetworkElement`

An intersection where multiple roads meet.

property is3Way
Whether or not this is a 3-way intersection.

Type bool

property is4Way
Whether or not this is a 4-way intersection.

Type bool

property isSignalized
Whether or not this is a signalized intersection.

Type bool

maneuversAt(*point*)
Get all maneuvers possible at a given point in the intersection.

Parameters **point** (*Vectorlike*) –

Return type List[*scenic.domains.driving.roads.Maneuver*]

class Signal(**, uid=None, openDriveID, country, type*)
Traffic lights, stop signs, etc.

openDriveID: int
ID number as in OpenDRIVE (unique ID of the signal within the database)

country: str
Country code of the signal

type: str
Type identifier according to country code.

property isTrafficLight

Whether or not this signal is a traffic light.

Type bool

class Network

A road network.

Networks are composed of roads, intersections, sidewalks, etc., which are all instances of `NetworkElement`.

elements: Dict[str, NetworkElement]

All network elements, indexed by unique ID.

roads: Tuple[Road]

All ordinary roads in the network (i.e. those not part of an intersection).

connectingRoads: Tuple[Road]

All roads connecting one exit of an intersection to another.

allRoads: Tuple[Road]

All roads of either type.

laneGroups: Tuple[LaneGroup]

All lane groups in the network.

lanes: Tuple[Lane]

All lanes in the network.

intersections: Tuple[Intersection]

All intersections in the network.

crossings: Tuple[PedestrianCrossing]

All pedestrian crossings in the network.

sidewalks: Tuple[Sidewalk]

All sidewalks in the network.

shoulders: Tuple[Shoulder]

All shoulders in the network (by default, includes parking lanes).

roadSections: Tuple[RoadSection]

All sections of ordinary roads in the network.

laneSections: Tuple[LaneSection]

All sections of lanes in the network.

driveOnLeft: bool

Whether or not cars drive on the left in this network.

tolerance: float

Distance tolerance for testing inclusion in network elements.

roadDirection: VectorField

Traffic flow vector field aggregated over all roads (0 elsewhere).

pickledExt = '.snet'

File extension for cached versions of processed networks.

exception DigestMismatchError

Bases: `Exception`

Exception raised when loading a cached map not matching the original file.

classmethod fromFile(*path*, *useCache=True*, *writeCache=True*, ***kwargs*)

Create a `Network` from a map file.

This function calls an appropriate parsing routine based on the extension of the given file. Supported map formats are:

- OpenDRIVE (`.xodr`): `Network.fromOpenDrive`

See the functions listed above for format-specific options to this function. If no file extension is given in **path**, this function searches for any file with the given name in one of the formats above (in order).

Parameters

- **path** – A string or other path-like object giving a path to a file. If no file extension is included, we search for any file type we know how to parse.
- **useCache** (*bool*) – Whether to use a cached version of the map, if one exists and matches the given map file (default true; note that if the map file changes, the cached version will still not be used).
- **writeCache** (*bool*) – Whether to save a cached version of the processed map after parsing has finished (default true).
- **kwargs** – Additional keyword arguments specific to particular map formats.

Raises

- **FileNotFoundError** – no readable map was found at the given path.
- **ValueError** – the given map is of an unknown format.

classmethod fromOpenDrive(*path*, *ref_points=20*, *tolerance=0.05*, *fill_gaps=True*, *fill_intersections=True*, *elide_short_roads=False*)

Create a `Network` from an OpenDRIVE file.

Parameters

- **path** – Path to the file, as in `Network.fromFile`.
- **ref_points** (*int*) – Number of points to discretize continuous reference lines into.
- **tolerance** (*float*) – Tolerance for merging nearby geometries.
- **fill_gaps** (*bool*) – Whether to attempt to fill gaps between adjacent lanes.
- **fill_intersections** (*bool*) – Whether to attempt to fill gaps inside intersections.
- **elide_short_roads** (*bool*) – Whether to attempt to fix geometry artifacts by eliding roads with length less than **tolerance**.

findPointIn(*point*, *elems*, *reject*)

Find the first of the given elements containing the point.

Elements which *actually* contain the point have priority; if none contain the point, then we search again allowing an error of up to **tolerance**. If there are still no matches, we return None, unless **reject** is true, in which case we reject the current sample.

Parameters

- **point** (*Vectorlike*) –
- **elems** (*Sequence[scenic.domains.driving.roads.NetworkElement]*) –
- **reject** (*Union[bool, str]*) –

Return type Optional[*scenic.domains.driving.roads.NetworkElement*]

elementAt (*point*, *reject=False*)
Get the highest-level `NetworkElement` at a given point, if any.

If the point lies in an `Intersection`, we return that; otherwise if the point lies in a `Road`, we return that; otherwise we return `None`, or reject the simulation if **reject** is true (default false).

Parameters **point** (*Vectorlike*) –

Return type Optional[*scenic.domains.driving.roads.NetworkElement*]

roadAt (*point*, *reject=False*)
Get the `Road` passing through a given point.

Parameters **point** (*Vectorlike*) –

Return type Optional[*scenic.domains.driving.roads.Road*]

laneAt (*point*, *reject=False*)
Get the `Lane` passing through a given point.

Parameters **point** (*Vectorlike*) –

Return type Optional[*scenic.domains.driving.roads.Lane*]

laneSectionAt (*point*, *reject=False*)
Get the `LaneSection` passing through a given point.

Parameters **point** (*Vectorlike*) –

Return type Optional[*scenic.domains.driving.roads.LaneSection*]

laneGroupAt (*point*, *reject=False*)
Get the `LaneGroup` passing through a given point.

Parameters **point** (*Vectorlike*) –

Return type Optional[*scenic.domains.driving.roads.LaneGroup*]

crossingAt (*point*, *reject=False*)
Get the `PedestrianCrossing` passing through a given point.

Parameters **point** (*Vectorlike*) –

Return type Optional[*scenic.domains.driving.roads.PedestrianCrossing*]

intersectionAt (*point*, *reject=False*)
Get the `Intersection` at a given point.

Parameters **point** (*Vectorlike*) –

Return type Optional[*scenic.domains.driving.roads.Intersection*]

nominalDirectionsAt (*point*, *reject=False*)
Get the nominal traffic direction(s) at a given point, if any.

There can be more than one such direction in an intersection, for example: a car at a given point could be going straight, turning left, etc.

Parameters **point** (*Vectorlike*) –

Return type Tuple[float]

show ()
Render a schematic of the road network for debugging.

If you call this function directly, you'll need to subsequently call `matplotlib.pyplot.show()` to actually display the diagram.

scenic.domains.driving.controllers

Low-level controllers useful for vehicles.

The Lateral/Longitudinal PID controllers are adapted from CARLA's PID controllers, which are licensed under the following terms:

> Copyright (c) 2018-2020 CVC.
>
> This work is licensed under the terms of the MIT license. For a copy, see <https://opensource.org/licenses/MIT>.

Summary of Module Members

Classes

`LQR`	
`PIDLateralController`	Lateral control using a PID to track a trajectory.
`PIDLongitudinalController`	Longitudinal control using a PID to reach a target speed.

Member Details

class PIDLongitudinalController(*K_P=0.5*, *K_D=0.1*, *K_I=0.2*, *dt=0.1*)

Longitudinal control using a PID to reach a target speed.

Parameters

- **K_P** – Proportional gain
- **K_D** – Derivative gain
- **K_I** – Integral gain
- **dt** – time step

run_step(*speed_error*)

Estimate the throttle/brake of the vehicle based on the PID equations.

Parameters **speed_error** – target speed minus current speed

Returns a signal between -1 and 1, with negative values indicating braking.

class PIDLateralController(*K_P=0.3*, *K_D=0.2*, *K_I=0*, *dt=0.1*)

Lateral control using a PID to track a trajectory.

Parameters

- **K_P** – Proportional gain
- **K_D** – Derivative gain
- **K_I** – Integral gain
- **dt** – time step

run_step(*cte*)

Estimate the steering angle of the vehicle based on the PID equations.

Parameters **cte** – cross-track error (distance to right of desired trajectory)

Returns a signal between -1 and 1, with -1 meaning maximum steering to the left.

scenic.domains.driving.workspace

Workspaces for the driving domain.

Summary of Module Members

Classes

`DrivingWorkspace`	Workspace created from a road `Network`.

Member Details

class DrivingWorkspace(*network*)

Bases: `scenic.core.workspaces.Workspace`

Workspace created from a road `Network`.

1.9.3 scenic.formats

Support for file formats not specific to particular simulators.

`opendrive`	Support for loading OpenDRIVE maps.

scenic.formats.opendrive

Support for loading OpenDRIVE maps.

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

scenic.formats.opendrive.workspace

Workspaces based on OpenDRIVE maps.

Summary of Module Members

Classes

OpenDriveWorkspace	

Member Details

scenic.formats.opendrive.xodr_parser

Parser for OpenDRIVE (.xodr) files.

Summary of Module Members

Functions

buffer_union	
warn	

Classes

Clothoid	An Euler spiral with curvature varying linearly between CURV0 and CURV1.
Cubic	A curve defined by the cubic polynomial a + bu + cu^2 + du^3.
Curve	Geometric elements which compose road reference lines.
Junction	
Lane	
LaneSection	
Line	A line segment between (x0, y0) and (x1, y1).
ParamCubic	A curve defined by the parametric equations u = a_u + b_up + c_up^2 + d_up^3, v = a_v + b_vp + c_vp^2 + d_up^3, with p in [0, p_range].
Poly3	Cubic polynomial.
Road	
RoadLink	Indicates Roads a and b, with ids id_a and id_b respectively, are connected.

continues on next page

Table 47 – continued from previous page

`RoadMap`	
`Signal`	Traffic lights, stop signs, etc.
`SignalReference`	

Exceptions

`OpenDriveWarning`

Member Details

class Poly3 (*a*, *b*, *c*, *d*)
Cubic polynomial.

class Curve (*x0*, *y0*, *hdg*, *length*)
Geometric elements which compose road reference lines. See the OpenDRIVE Format Specification for coordinate system details.

to_points (*num*, *extra_points=[]*)
Sample NUM evenly-spaced points from curve.

Points are tuples of (x, y, s) with (x, y) absolute coordinates and s the arc length along the curve. Additional points at s values in extra_points are included if they are contained in the curve (unless they are extremely close to one of the equally-spaced points).

abstract point_at (*s*)
Get an (x, y, s) point along the curve at the given s coordinate.

rel_to_abs (*point*)
Convert from relative coordinates of curve to absolute coordinates. I.e. rotate counterclockwise by self.hdg and translate by (x0, x1).

class Cubic (*x0*, *y0*, *hdg*, *length*, *a*, *b*, *c*, *d*)
Bases: *`scenic.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.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.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.formats.opendrive.xodr_parser.Curve`*

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

class RoadLink(*id_a*, *id_b*, *contact_a*, *contact_b*)
Indicates Roads a and b, with ids id_a and id_b respectively, are connected.

class Signal(*id_*, *country*, *type_*, *subtype*, *orientation*, *validity=None*)
Traffic lights, stop signs, etc.

1.9.4 scenic.simulators

World models and interfaces for particular simulators.

`carla`	Interface to the CARLA driving simulator.
`gta`	Scenic world model for Grand Theft Auto V (GTAV).
`lgsvl`	Interface to the LGSVL driving simulator.
`webots`	Scenic world models for the Webots robotics simulator.
`xplane`	Scenic world model for the X-Plane flight simulator.
`utils`	Various utilities useful across multiple simulators.

scenic.simulators.carla

Interface to the CARLA driving simulator.

This interface has been tested with CARLA versions 0.9.9, 0.9.10, and 0.9.11. It supports dynamic scenarios involving vehicles, pedestrians, and props.

The interface implements the `scenic.domains.driving` abstract domain, so any object types, behaviors, utility functions, etc. from that domain may be used freely. For details of additional CARLA-specific functionality, see the world model `scenic.simulators.carla.model`.

`model`	Scenic world model for traffic scenarios in CARLA.
`actions`	Actions for dynamic agents in CARLA scenarios.
`behaviors`	Behaviors for dynamic agents in CARLA scenarios.
`simulator`	Simulator interface for CARLA.
`blueprints`	CARLA blueprints for cars, pedestrians, etc.
`controller`	This module contains PID controllers to perform lateral and longitudinal control.
`misc`	Module with auxiliary functions.

scenic.simulators.carla.model

Scenic world model for traffic scenarios in CARLA.

The model currently supports vehicles, pedestrians, and props. It implements the basic `Car` and `Pedestrian` classes from the `scenic.domains.driving` domain, while also providing convenience classes for specific types of objects like bicycles, traffic cones, etc. Vehicles and pedestrians support the basic actions and behaviors from the driving domain; several more are automatically imported from `scenic.simulators.carla.actions` and `scenic.simulators.carla.behaviors`.

The model defines several global parameters, whose default values can be overridden in scenarios using the `param` statement or on the command line using the `--param` option:

Global Parameters

- **carla_map** (*str*) – Name of the CARLA map to use, e.g. 'Town01'. Can also be set to `None`, in which case CARLA will attempt to create a world from the **map** file used in the scenario (which must be an `.xodr` file).

- **timestep** (*float*) – Timestep to use for simulations (i.e., how frequently Scenic interrupts CARLA to run behaviors, check requirements, etc.), in seconds. Default is 0.1 seconds.
- **weather** (*str or dict*) – Weather to use for the simulation. Can be either a string identifying one of the CARLA weather presets (e.g. 'ClearSunset') or a dictionary specifying all the weather parameters (see carla.WeatherParameters). Default is a uniform distribution over all the weather presets.
- **address** (*str*) – IP address at which to connect to CARLA. Default is localhost (127.0.0.1).
- **port** (*int*) – Port on which to connect to CARLA. Default is 2000.
- **timeout** (*float*) – Maximum time to wait when attempting to connect to CARLA, in seconds. Default is 10.
- **render** (*int*) – Whether or not to have CARLA create a window showing the simulations from the point of view of the ego object: 1 for yes, 0 for no. Default 1.
- **record** (*str*) – If nonempty, folder in which to save CARLA record files for replaying the simulations.

Summary of Module Members

Functions

`freezeTrafficLights`	Freezes all traffic lights in the scene.
`getClosestTrafficLightStatus`	
`getTrafficLightStatus`	
`setAllIntersectionTrafficLightsStatus`	
`setClosestTrafficLightStatus`	
`setTrafficLightStatus`	
`unfreezeTrafficLights`	Unfreezes all traffic lights in the scene.
`withinDistanceToRedYellowTrafficLight`	
`withinDistanceToTrafficLight`	

Classes

ATM	
Advertisement	
Barrel	
Barrier	
Bench	
Bicycle	
Box	
BusStop	
Car	A car.
CarlaActor	Abstract class for CARLA objects.
Case	
Chair	
Cone	
Container	
CreasedBox	
Debris	
Garbage	
Gnome	
IronPlate	
Kiosk	
Mailbox	
Motorcycle	
NPCCar	
Pedestrian	A pedestrian.
PlantPot	
Prop	Abstract class for props, i.e. non-moving objects.

continues on next page

Table 52 – continued from previous page

Table	
TrafficWarning	
Trash	
Truck	
Vehicle	Abstract class for steerable vehicles.
VendingMachine	

Member Details

class Car(<*specifiers*>)

Bases: *scenic.simulators.carla.model.Vehicle*

A car.

The default blueprint (see *CarlaActor*) is a uniform distribution over the blueprints listed in *scenic.simulators.carla.blueprints.carModels*.

class CarlaActor(<*specifiers*>)

Bases: *scenic.domains.driving.model.DrivingObject*

Abstract class for CARLA objects.

Properties

- **carlaActor** (*dynamic*) – Set during simulations to the carla.Actor representing this object.
- **blueprint** (*str*) – Identifier of the CARLA blueprint specifying the type of object.
- **rolename** (*str*) – Can be used to differentiate specific actors during runtime. Default value None.
- **physics** (*bool*) – Whether physics is enabled for this object in CARLA. Default true.

class Pedestrian(<*specifiers*>)

Bases: *scenic.domains.driving.model.Pedestrian*, *scenic.simulators.carla.model.CarlaActor*, *scenic.domains.driving.model.Walks*

A pedestrian.

The default blueprint (see *CarlaActor*) is a uniform distribution over the blueprints listed in *scenic.simulators.carla.blueprints.walkerModels*.

class Prop(<*specifiers*>)

Bases: *scenic.simulators.carla.model.CarlaActor*

Abstract class for props, i.e. non-moving objects.

Properties

- **heading** (*float*) – Default value overridden to be uniformly random.
- **physics** (*bool*) – Default value overridden to be false.

class Vehicle(*<specifiers>*)
Bases: *scenic.domains.driving.model.Vehicle*, *scenic.simulators.carla.model.CarlaActor*, *scenic.domains.driving.model.Steers*

Abstract class for steerable vehicles.

freezeTrafficLights()
Freezes all traffic lights in the scene.

Frozen traffic lights can be modified by the user but the time will not update them until unfrozen.

unfreezeTrafficLights()
Unfreezes all traffic lights in the scene.

_getClosestTrafficLight(*vehicle*, *distance=100*)
Returns the closest traffic light affecting 'vehicle', up to a maximum of 'distance'

scenic.simulators.carla.actions

Actions for dynamic agents in CARLA scenarios.

Summary of Module Members

Classes

PedestrianAction	
SetAngularVelocityAction	
SetAutopilotAction	
SetGearAction	
SetJumpAction	
SetManualFirstGearShiftAction	
SetManualGearShiftAction	
SetTrafficLightAction	Set the traffic light to desired color.
SetTransformAction	
SetWalkAction	
TrackWaypointsAction	
VehicleAction	

Member Details

class SetTrafficLightAction(*color*, *distance=100*, *group=False*)
Bases: `scenic.simulators.carla.actions.VehicleAction`

Set the traffic light to desired color. It will only take effect if the car is within a given distance of the traffic light.

Parameters

- **color** – the string red/yellow/green/off/unknown
- **distance** – the maximum distance to search for traffic lights from the current position

scenic.simulators.carla.behaviors

Behaviors for dynamic agents in CARLA scenarios.

Summary of Module Members

Classes

AutopilotBehavior	Behavior causing a vehicle to use CARLA's built-in autopilot.
CrossingBehavior	This behavior dynamically controls the speed of an actor that will perpendicularly (or close to) cross the road, so that it arrives at a spot in the road at the same time as a reference actor.
WalkBehavior	
WalkForwardBehavior	

Member Details

class AutopilotBehavior(**args*, ***kwargs*)
Bases: `scenic.core.dynamics.Behavior`

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

class CrossingBehavior(**args*, ***kwargs*)
Bases: `scenic.core.dynamics.Behavior`

This behavior dynamically controls the speed of an actor that will perpendicularly (or close to) cross the road, so that it arrives at a spot in the road at the same time as a reference actor.

Parameters

- **min_speed** (*float*) – minimum speed of the crossing actor. As this is a type of "synchronization action", a minimum speed is needed, to allow the actor to keep moving even if the reference actor has stopped
- **threshold** (*float*) – starting distance at which the crossing actor starts moving
- **final_speed** (*float*) – speed of the crossing actor after the reference one surpasses it

scenic.simulators.carla.simulator

Simulator interface for CARLA.

Summary of Module Members

Classes

`CarlaSimulation`	
`CarlaSimulator`	Implementation of *`Simulator`* for CARLA.

Member Details

`class CarlaSimulator`(*carla_map, map_path, address='127.0.0.1', port=2000, timeout=10, render=True, record='', timestep=0.1*)
Bases: `scenic.domains.driving.simulators.DrivingSimulator`

Implementation of *`Simulator`* for CARLA.

scenic.simulators.carla.blueprints

CARLA blueprints for cars, pedestrians, etc.

`carModels = ['vehicle.audi.a2', 'vehicle.audi.etron', 'vehicle.audi.tt', 'vehicle.bmw.granc`
blueprints for cars

`bicycleModels = ['vehicle.bh.crossbike', 'vehicle.diamondback.century', 'vehicle.gazelle.or`
blueprints for bicycles

`motorcycleModels = ['vehicle.harley-davidson.low_rider', 'vehicle.kawasaki.ninja', 'vehicle`
blueprints for motorcycles

`truckModels = ['vehicle.carlamotors.carlacola', 'vehicle.tesla.cybertruck']`
blueprints for trucks

`trashModels = ['static.prop.trashcan01', 'static.prop.trashcan02', 'static.prop.trashcan03`
blueprints for trash cans

`coneModels = ['static.prop.constructioncone', 'static.prop.trafficcone01', 'static.prop.tra`
blueprints for traffic cones

`debrisModels = ['static.prop.dirtdebris01', 'static.prop.dirtdebris02', 'static.prop.dirtde`
blueprints for road debris

`vendingMachineModels = ['static.prop.vendingmachine']`
blueprints for vending machines

`chairModels = ['static.prop.plasticchair']`
blueprints for chairs

`busStopModels = ['static.prop.busstop']`
blueprints for bus stops

`advertisementModels = ['static.prop.advertisement', 'static.prop.streetsign', 'static.prop`
blueprints for roadside billboards

garbageModels = ['static.prop.colacan', 'static.prop.garbage01', 'static.prop.garbage02',
: blueprints for pieces of trash

containerModels = ['static.prop.container', 'static.prop.clothcontainer', 'static.prop.glas
: blueprints for containers

tableModels = ['static.prop.table', 'static.prop.plastictable']
: blueprints for tables

barrierModels = ['static.prop.streetbarrier', 'static.prop.chainbarrier', 'static.prop.cha
: blueprints for traffic barriers

plantpotModels = ['static.prop.plantpot01', 'static.prop.plantpot02', 'static.prop.plantpot
: blueprints for flowerpots

mailboxModels = ['static.prop.mailbox']
: blueprints for mailboxes

gnomeModels = ['static.prop.gnome']
: blueprints for garden gnomes

creasedboxModels = ['static.prop.creasedbox01', 'static.prop.creasedbox02', 'static.prop.c
: blueprints for creased boxes

caseModels = ['static.prop.travelcase', 'static.prop.briefcase', 'static.prop.guitarcase']
: blueprints for briefcases, suitcases, etc.

boxModels = ['static.prop.box01', 'static.prop.box02', 'static.prop.box03']
: blueprints for boxes

benchModels = ['static.prop.bench01', 'static.prop.bench02', 'static.prop.bench03']
: blueprints for benches

barrelModels = ['static.prop.barrel']
: blueprints for barrels

atmModels = ['static.prop.atm']
: blueprints for ATMs

kioskModels = ['static.prop.kiosk_01']
: blueprints for kiosks

ironplateModels = ['static.prop.ironplank']
: blueprints for iron plates

trafficwarningModels = ['static.prop.trafficwarning']
: blueprints for traffic warning signs

walkerModels = ['walker.pedestrian.0001', 'walker.pedestrian.0002', 'walker.pedestrian.000
: blueprints for pedestrians

scenic.simulators.carla.controller

This module contains PID controllers to perform lateral and longitudinal control.

Summary of Module Members

Classes

PIDLateralController	PIDLateralController implements lateral control using a PID.
PIDLongitudinalController	PIDLongitudinalController implements longitudinal control using a PID.
VehiclePIDController	VehiclePIDController is the combination of two PID controllers (lateral and longitudinal) to perform the low level control a vehicle from client side

Member Details

class VehiclePIDController(*vehicle*, *args_lateral=None*, *args_longitudinal=None*, *max_throttle=0.75*, *max_brake=0.3*, *max_steering=0.8*)

VehiclePIDController is the combination of two PID controllers (lateral and longitudinal) to perform the low level control a vehicle from client side

run_step(*target_speed*, *waypoint*)

Execute one step of control invoking both lateral and longitudinal PID controllers to reach a target waypoint at a given target_speed.

Parameters

- **target_speed** – desired vehicle speed
- **waypoint** – target location encoded as a waypoint

Returns distance (in meters) to the waypoint

class PIDLongitudinalController(*vehicle*, *K_P=1.0*, *K_D=0.0*, *K_I=0.0*, *dt=0.03*)

PIDLongitudinalController implements longitudinal control using a PID.

run_step(*target_speed*, *debug=False*)

Execute one step of longitudinal control to reach a given target speed.

param target_speed target speed in Km/h

param debug boolean for debugging

return throttle control

_pid_control(*target_speed*, *current_speed*)

Estimate the throttle/brake of the vehicle based on the PID equations

param target_speed target speed in Km/h

param current_speed current speed of the vehicle in Km/h

return throttle/brake control

class PIDLateralController(*vehicle*, *K_P=1.0*, *K_D=0.0*, *K_I=0.0*, *dt=0.03*)

PIDLateralController implements lateral control using a PID.

run_step(*waypoint*)

Execute one step of lateral control to steer the vehicle towards a certain waypoin.

param waypoint target waypoint

return steering control in the range [-1, 1] where: -1 maximum steering to left +1 maximum steering to right

_pid_control(*waypoint*, *vehicle_transform*)
Estimate the steering angle of the vehicle based on the PID equations

param waypoint target waypoint

param vehicle_transform current transform of the vehicle

return steering control in the range [-1, 1]

scenic.simulators.carla.misc

Module with auxiliary functions.

Summary of Module Members

Functions

compute_distance	Euclidean distance between 3D points
compute_magnitude_angle	Compute relative angle and distance between a target_location and a current_location
distance_vehicle	Returns the 2D distance from a waypoint to a vehicle
draw_waypoints	Draw a list of waypoints at a certain height given in z.
get_speed	Compute speed of a vehicle in Km/h.
is_within_distance	Check if a target object is within a certain distance from a reference object.
is_within_distance_ahead	Check if a target object is within a certain distance in front of a reference object.
positive	Return the given number if positive, else 0
vector	Returns the unit vector from location_1 to location_2

Member Details

draw_waypoints(*world*, *waypoints*, *z=0.5*)
Draw a list of waypoints at a certain height given in z.

param world carla.world object

param waypoints list or iterable container with the waypoints to draw

param z height in meters

get_speed(*vehicle*)
Compute speed of a vehicle in Km/h.

param vehicle the vehicle for which speed is calculated

return speed as a float in Km/h

is_within_distance_ahead(*target_transform*, *current_transform*, *max_distance*)
Check if a target object is within a certain distance in front of a reference object.

Parameters

- **target_transform** – location of the target object
- **current_transform** – location of the reference object
- **orientation** – orientation of the reference object
- **max_distance** – maximum allowed distance

Returns True if target object is within max_distance ahead of the reference object

is_within_distance(*target_location*, *current_location*, *orientation*, *max_distance*, *d_angle_th_up*, *d_angle_th_low=0*)

Check if a target object is within a certain distance from a reference object. A vehicle in front would be something around 0 deg, while one behind around 180 deg.

param target_location location of the target object

param current_location location of the reference object

param orientation orientation of the reference object

param max_distance maximum allowed distance

param d_angle_th_up upper thereshold for angle

param d_angle_th_low low thereshold for angle (optional, default is 0)

return True if target object is within max_distance ahead of the reference object

compute_magnitude_angle(*target_location*, *current_location*, *orientation*)

Compute relative angle and distance between a target_location and a current_location

param target_location location of the target object

param current_location location of the reference object

param orientation orientation of the reference object

return a tuple composed by the distance to the object and the angle between both objects

distance_vehicle(*waypoint*, *vehicle_transform*)

Returns the 2D distance from a waypoint to a vehicle

param waypoint actual waypoint

param vehicle_transform transform of the target vehicle

vector(*location_1*, *location_2*)

Returns the unit vector from location_1 to location_2

param location_1, location_2 carla.Location objects

compute_distance(*location_1*, *location_2*)

Euclidean distance between 3D points

param location_1, location_2 3D points

positive(*num*)

Return the given number if positive, else 0

param num value to check

scenic.simulators.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

Module Attributes

roadDirection	Vector field representing the nominal traffic direction at a point on the road
road	Region representing the roads in the GTA map.
curb	Region representing the curbs in the GTA map.
workspace	Workspace over the *road* Region.

Functions

createPlatoonAt	Create a platoon starting from the given car.

Classes

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

Member Details

roadDirection
Vector field representing the nominal traffic direction at a point on the road

road
Region representing the roads in the GTA map.

curb
Region representing the curbs in the GTA map.

workspace
Workspace over the *road* Region.

class Car (*<specifiers>*)
Bases: *scenic.core.object_types.Object*

Scenic class for cars.

Properties

- **position** – The default position is uniformly random over the *road*.
- **heading** – The default heading is aligned with *roadDirection*, plus an offset given by `roadDeviation`.
- **roadDeviation** (*float*) – Relative heading with respect to the road direction at the *Car*'s position. Used by the default value for `heading`.
- **model** (*CarModel*) – Model of the car.
- **color** (`Color` or RGB tuple) – Color of the car.

class EgoCar (*<specifiers>*)
Bases: *scenic.simulators.gta.model.Car*

Convenience subclass with defaults for ego cars.

class Bus (*<specifiers>*)
Bases: *scenic.simulators.gta.model.Car*

Convenience subclass for buses.

class Compact (*<specifiers>*)
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

`CarModel`	A model of car in GTA.
`GTA`	
`Map`	Represents roads and obstacles in GTA, extracted from a map image.
`MapWorkspace`	Workspace whose rendering is handled by a Map

Member Details

class Map(*imagePath*, *Ax*, *Ay*, *Bx*, *By*)

Represents roads and obstacles in GTA, extracted from a map image.

This code handles images from the GTA V Interactive Map, rendered with the "Road" setting.

Parameters

- **imagePath** (`str`) – path to image file
- **Ax** (`float`) – width of one pixel in GTA coordinates
- **Ay** (`float`) – height of one pixel in GTA coordinates
- **Bx** (`float`) – GTA X-coordinate of bottom-left corner of image
- **By** (`float`) – GTA Y-coordinate of bottom-left corner of image

class MapWorkspace(*mappy*, *region*)

Bases: `scenic.core.workspaces.Workspace`

Workspace whose rendering is handled by a Map

class CarModel(*name*, *width*, *length*, *viewAngle=1.5707963267948966*)

A model of car in GTA.

Attributes

- **name** (*str*) – name of model in GTA
- **width** (*float*) – width of this model of car
- **length** (*float*) – length of this model of car
- **viewAngle** (*float*) – view angle in radians (default is 90 degrees)

Class Attributes **models** – dict mapping model names to the corresponding `CarModel`

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

init_theta: float
Alias for field number 0

tangent: float
Alias for field number 1

opp_loc: Tuple[float, float]

Alias for field number 2

mid_loc: Tuple[float, float]
Alias for field number 3

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

classmethod _make(*iterable*)
Make a new EdgeData object from a sequence or iterable

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

scenic.simulators.gta.img_modf

This file has basic image modification functions

Summary of Module Members

Functions

`convert_black_white`
`get_edges`
`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
Stop
Vehicle

Member Details

scenic.simulators.lgsvl

Interface to the LGSVL driving simulator.

This interface has been tested with LGSVL version 2020.06. It supports dynamic scenarios involving vehicles and pedestrians.

The interface implements the *scenic.domains.driving* abstract domain, so any object types, behaviors, utility functions, etc. from that domain may be used freely.

model	Scenic world model for the LGSVL Simulator.
actions	Actions for agents in the LGSVL model.

continues on next page

Table 69 – continued from previous page

behaviors	Behaviors for dynamic agents in LGSVL.
simulator	Simulator interface for LGSVL.
utils	Common LGSVL interface.

scenic.simulators.lgsvl.model

Scenic world model for the LGSVL Simulator.

Summary of Module Members

Functions

`LGSVLSimulator`	

Classes

`ApolloCar`	
`Car`	alias of `scenic.simulators.lgsvl.model.EgoCar`
`EgoCar`	
`LGSVLObject`	
`NPCCar`	
`Pedestrian`	
`Vehicle`	
`Waypoint`	

Member Details

scenic.simulators.lgsvl.actions

Actions for agents in the LGSVL model.

Summary of Module Members

Classes

`CancelWaypointsAction`
`FollowWaypointsAction`
`SetDestinationAction`
`TrackWaypointsAction`

Member Details

scenic.simulators.lgsvl.behaviors

Behaviors for dynamic agents in LGSVL.

Summary of Module Members

Classes

`DriveTo`
`FollowWaypoints`

Member Details

scenic.simulators.lgsvl.simulator

Simulator interface for LGSVL.

Summary of Module Members

Classes

`LGSVLSimulation`
`LGSVLSimulator`

Member Details

scenic.simulators.lgsvl.utils

Common LGSVL interface.

Summary of Module Members

Functions

lgsvlToScenicAngularSpeed	
lgsvlToScenicElevation	Convert LGSVL positions to Scenic elevations.
lgsvlToScenicPosition	Convert LGSVL positions to Scenic positions.
lgsvlToScenicRotation	Convert LGSVL rotations to Scenic headings.
scenicToLGSVLPosition	
scenicToLGSVLRotation	

Member Details

lgsvlToScenicPosition(*pos*)
Convert LGSVL positions to Scenic positions.

lgsvlToScenicElevation(*pos*)
Convert LGSVL positions to Scenic elevations.

lgsvlToScenicRotation(*rot*)
Convert LGSVL rotations to Scenic headings.

Drops all but the Y component.

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(<*specifiers*>)

Bases: *scenic.core.object_types.Object*

Flag indicating the goal location.

class Rover(<*specifiers*>)

Bases: *scenic.core.object_types.Object*

Mars rover.

class Debris(<*specifiers*>)

Bases: *scenic.core.object_types.Object*

Abstract class for debris scattered randomly in the workspace.

class BigRock(<*specifiers*>)

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

Large rock.

class Rock(<*specifiers*>)

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

Small rock.

class Pipe(<*specifiers*>)

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

continues on next page

Table 80 – continued from previous page

`Truck`
`WebotsObject`
`WorkBarrier`

Member Details

scenic.simulators.webots.road.world

Stub to allow changing the Webots world without changing the model.

Summary of Module Members

Module Attributes

`worldPath`	Path to the WBT file to load the Webots world from

Functions

`setLocalWorld`	Select a WBT file relative to the given module.

Member Details

`worldPath = '../tests/simulators/webots/road/simple.wbt'`
Path to the WBT file to load the Webots world from

`setLocalWorld`(*module*, *relpath*)
Select a WBT file relative to the given module.

This function is intended to be used with `__file__` as the *module*.

scenic.simulators.webots.road.interface

Python library supporting the main Scenic module.

Summary of Module Members

Functions

polygonWithPoints	
regionWithPolygons	

Classes

Crossroad	OSM crossroads
OSMObject	Objects with OSM id tags
PedestrianCrossing	PedestrianCrossing nodes
Road	OSM roads
WebotsWorkspace	

Member Details

class OSMObject(*attrs*)

Objects with OSM id tags

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

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

OSM roads

class Crossroad(*attrs*)

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

OSM crossroads

class PedestrianCrossing(*attrs*)

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*, *length*)

Bases: `tuple`

_asdict()

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

classmethod _make(*iterable*)

Make a new CarModel object from a sequence or iterable

_replace(***kwds*)

Return a new CarModel object replacing specified fields with new values

length

Alias for field number 2

name

Alias for field number 0

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*)
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(<*specifiers*>)

Bases: `scenic.core.object_types.Object`

Placeholder object for the plane.

scenic.simulators.utils

Various utilities useful across multiple simulators.

`colors`	A basic color type.

scenic.simulators.utils.colors

A basic color type.

This used for example to represent car colors in the abstract driving domain, as well as in the interfaces to GTA and Webots.

Summary of Module Members

Classes

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

Member Details

class Color(*r*, *g*, *b*)
Bases: *scenic.simulators.utils.colors.Color*

A color as an RGB tuple.

static uniformColor()
Return a uniformly random color.

static defaultCarColor()
Default color distribution for cars.

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

class NoisyColorDistribution(**args*, ***kwargs*)
Bases: *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 ColorMutator
Bases: *scenic.core.object_types.Mutator*

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

1.9.5 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*)
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.
`findConstructorsIn`	Find all constructors (Scenic classes) defined in a namespace.
`functionForStatement`	
`gatherBehaviorNamespacesFrom`	Gather any global namespaces which could be referred to by behaviors.
`nameForStatement`	
`parseTranslatedSource`	
`partitionByImports`	Partition the tokens into blocks ending with import statements.
`peek`	
`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.

continues on next page

Table 101 – continued from previous page

`topLevelNamespace`	Creates an environment like that of a Python script being run directly.
`translateParseTree`	Modify the Python AST to produce the desired Scenic semantics.

Classes

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

Member Details

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

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

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

`scenarioFromFile`(*path*, *params={}*, *model=None*, *scenario=None*, *cacheImports=False*)

Compile a Scenic file into a `Scenario`.

Parameters

- **`path`** (`str`) – path to a Scenic file
- **`params`** (`dict`) – global parameters to override
- **`model`** (`str`) – Scenic module to use as world model
- **`scenario`** (`str`) – if there are multiple scenarios in the file, which one to use
- **`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*, *params={}*, *model=None*, *scenario=None*, *filename='<stream>'*, *path=None*, *cacheImports=False*)
Compile a stream of Scenic code into a `Scenario`.

topLevelNamespace(*path=None*)
Creates an environment like that of a Python script being run directly.

Specifically, __name__ is '__main__', __file__ is the path used to invoke the script (not necessarily its absolute path), and the parent directory is added to the path so that 'import blobbo' will import blobbo from that directory if it exists there.

compileStream(*stream*, *namespace*, *params={}*, *model=None*, *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*, *bases*)
Bases: `tuple`

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

classmethod _make(*iterable*)
Make a new Constructor object from a sequence or iterable

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

bases
Alias for field number 1

name
Alias for field number 0

class ModifierInfo(*name*, *terminators*, *contexts*)
Bases: `tuple`

name: str
Alias for field number 0

terminators: Tuple[str]
Alias for field number 1

contexts: Optional[Tuple[str]]
Alias for field number 2

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

classmethod _make(*iterable*)
Make a new ModifierInfo object from a sequence or iterable

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

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

syntax: str
Alias for field number 0

implementation: Optional[str]
Alias for field number 1

arity: int
Alias for field number 2

token: Tuple[int, str]
Alias for field number 3

node: _ast.AST
Alias for field number 4

contexts: Optional[Tuple[str]]
Alias for field number 5

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

classmethod _make(*iterable*)
Make a new InfixOp object from a sequence or iterable

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

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

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

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

class Peekable(*gen*)
Utility class to allow iterator lookahead.

class TokenTranslator(*constructors=()*, *filename='<unknown>'*)
Translates a Scenic token stream into valid Python syntax.

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

translate(*tokens*)
Do the actual translation of the token stream.

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

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

class LocalFinder

Bases: `ast.NodeVisitor`

Utility class for finding all local variables of a code block.

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

Modify the Python AST to produce the desired Scenic semantics.

executeCodeIn(*code*, *namespace*)

Execute the final translated Python code in the given namespace.

storeScenarioStateIn(*namespace*, *requirementSyntax*)

Post-process an executed Scenic module, extracting state from the veneer.

gatherBehaviorNamespacesFrom(*behaviors*)

Gather any global namespaces which could be referred to by behaviors.

We'll need to rebind any sampled values in them at runtime.

constructScenarioFrom(*namespace*, *scenarioName=None*)

Build a Scenario object from an executed Scenic module.

scenic.syntax.veneer

Python implementations of Scenic language constructs.

This module is automatically imported by all Scenic programs. In addition to defining the built-in functions, operators, specifiers, etc., it also stores global state such as the list of all created Scenic objects.

Summary of Module Members

Functions

Ahead	The 'ahead of X [by Y]' polymorphic specifier.
AngleFrom	The 'angle from <vector> to <vector>' operator.
AngleTo	The 'angle to <vector>' operator (using the position of ego as the reference).
ApparentHeading	The 'apparent heading of <oriented point> [from <vector>]' operator.
ApparentlyFacing	The 'apparently facing <heading> [from <vector>]' specifier.
At	The 'at <vector>' specifier.
Back	The 'back of <object>' operator.
BackLeft	The 'back left of <object>' operator.
BackRight	The 'back right of <object>' operator.
Behind	The 'behind X [by Y]' polymorphic specifier.
Beyond	The 'beyond X by Y [from Z]' polymorphic specifier.
CanSee	The 'X can see Y' polymorphic operator.
DistanceFrom	The `distance from {X} to {Y}` polymorphic operator.
Facing	The 'facing X' polymorphic specifier.
FacingToward	The 'facing toward <vector>' specifier.
FieldAt	The '<VectorField> at <vector>' operator.

continues on next page

Table 103 – 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.
NotVisible	The 'not visible <region>' operator.
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.
Visible	The 'visible <region>' operator.
VisibleFrom	The 'visible from <Point>' specifier.
VisibleSpec	The 'visible' specifier (equivalent to 'visible from ego').
With	The 'with <property> <value>' specifier.
activate	Activate the veneer when beginning to compile a Scenic module.
alwaysProvidesOrientation	Whether a Region or distribution over Regions always provides an orientation.
beginSimulation	
callWithStarArgs	
deactivate	Deactivate the veneer after compiling a Scenic module.
ego	Function implementing loads and stores to the 'ego' pseudo-variable.
endScenario	
endSimulation	
executeInBehavior	
executeInGuard	
executeInRequirement	
executeInScenario	
filter	
finishScenarioSetup	

continues on next page

Table 103 – continued from previous page

globalParameters	
in_initial_scenario	
instantiateSimulator	
isActive	Are we in the middle of compiling a Scenic module?
leftSpecHelper	
localPath	
makeRequirement	
model	
mutate	Function implementing the mutate statement.
param	Function implementing the param statement.
prepareScenario	
registerDynamicScenarioClass	
registerExternalParameter	Register a parameter whose value is given by an external sampler.
registerObject	Add a Scenic object to the global list of created objects.
require	Function implementing the require statement.
require_always	Function implementing the 'require always' statement.
resample	The built-in resample function.
simulation	
simulationInProgress	
simulator	
startScenario	
terminate_after	
terminate_simulation_when	Function implementing the 'terminate simulation when' statement.
terminate_when	Function implementing the 'terminate when' statement.
verbosePrint	Built-in function printing a message when the verbosity is >0.
wrapStarredValue	

Classes

Modifier
ParameterTableProxy

Member Details

ego (*obj=None*)
Function implementing loads and stores to the 'ego' pseudo-variable.

The translator calls this with no arguments for loads, and with the source value for stores.

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

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

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

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

verbosePrint (*msg*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *level=1*)
Built-in function printing a message when the verbosity is >0.

(Or when the verbosity exceeds the specified level.)

require_always (*reqID*, *req*, *line*)
Function implementing the 'require always' statement.

terminate_when (*reqID*, *req*, *line*)
Function implementing the 'terminate when' statement.

terminate_simulation_when (*reqID*, *req*, *line*)
Function implementing the 'terminate simulation when' statement.

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

NotVisible (*region*)
The 'not visible <region>' operator.

Front (*X*)
The 'front of <object>' operator.

Back (*X*)
The 'back of <object>' operator.

Left (*X*)
The 'left of <object>' operator.

Right (*X*)
The 'right of <object>' operator.

FrontLeft (*X*)
The 'front left of <object>' operator.

FrontRight(*X*)
The ‘front right of <object>’ operator.

BackLeft(*X*)
The ‘back left of <object>’ operator.

BackRight(*X*)
The ‘back right of <object>’ operator.

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 {X} to {Y}` polymorphic operator.

Allowed forms:

- `distance from` <vector> [`to` <vector>]
- `distance from` <region> [`to` <vector>]
- `distance from` <vector> `to` <region>

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

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>

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.

Return type *scenic.core.vectors.Vector*

angleWith(*other*)

Compute the signed angle between self and other.

The angle is positive if other is counterclockwise of self (considering the smallest possible rotation to align them).

Return type float

class VectorField(*name*, *value*, *minSteps=4*, *defaultStepSize=5*)

A vector field, providing a heading at every point.

Parameters

- **name** (`str`) – name for debugging.
- **value** – function computing the heading at the given `Vector`.
- **minSteps** (`int`) – Minimum number of steps for `followFrom`; default 4.
- **defaultStepSize** (`float`) – Default step size for `followFrom`; default 5.

followFrom(*pos*, *dist*, *steps=None*, *stepSize=None*)

Follow the field from a point for a given distance.

Uses the forward Euler approximation, covering the given distance with equal-size steps. The number of steps can be given manually, or computed automatically from a desired step size.

Parameters

- **pos** (`Vector`) – point to start from.
- **dist** (`float`) – distance to travel.
- **steps** (`int`) – number of steps to take, or `None` to compute the number of steps based on the distance (default `None`).
- **stepSize** (`float`) – length used to compute how many steps to take, or `None` to use the field's default step size.

static forUnionOf(*regions*)

Creates a `PiecewiseVectorField` from the union of the given regions.

If none of the regions have an orientation, returns `None` instead.

class PolygonalVectorField(*name*, *cells*, *headingFunction=None*, *defaultHeading=None*)

Bases: `scenic.core.vectors.VectorField`

A piecewise-constant vector field defined over polygonal cells.

Parameters

- **name** (`str`) – name for debugging.
- **cells** – a sequence of cells, with each cell being a pair consisting of a Shapely geometry and a heading. If the heading is `None`, we call the given **headingFunction** for points in the cell instead.

- **headingFunction** – function computing the heading for points in cells without specified headings, if any (default None).
- **defaultHeading** – heading for points not contained in any cell (default None, meaning reject such points).

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

Abstract class for regions.

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

intersects(*other*)
Check if this *Region* intersects another.

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

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

Not supported by all region types.

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*, *name=None*)
Bases: *scenic.core.regions.Region*

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

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

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

signedDistanceTo (*point*)
Compute the signed distance of the PolylineRegion to a point.

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

class Workspace (*region=<AllRegion everywhere>*)
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=1*)
Zoom the schematic around the specified objects

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

class Mutator
An object controlling how the `mutate` statement affects an *Object*.

A *Mutator* can be assigned to the `mutator` property of an *Object* to control the effect of the `mutate` statement. When mutation is enabled for such an object using that statement, the mutator's *appliedTo* method is called to compute a mutated version.

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

class Range (**args*, ***kwargs*)
Bases: *scenic.core.distributions.Distribution*

Uniform distribution over a range

class DiscreteRange (**args*, ***kwargs*)
Bases: *scenic.core.distributions.Distribution*

Distribution over a range of integers.

class Options (**args*, ***kwargs*)
Bases: *scenic.core.distributions.MultiplexerDistribution*

Distribution over a finite list of options.

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

Uniform(**opts*)

Uniform distribution over a finite list of options.

Implemented as an instance of `Options` when the set of options is known statically, and an instance of `UniformDistribution` otherwise.

Discrete

alias of `scenic.core.distributions.Options`

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

Bases: `scenic.core.distributions.Distribution`

Normal distribution

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

Bases: `scenic.core.distributions.Normal`

Truncated normal distribution.

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

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

Bases: `scenic.core.external_params.VerifaiParameter`

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

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

Bases: `scenic.core.external_params.VerifaiParameter`

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

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

Bases: `scenic.core.distributions.Options`

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

class Point(*<specifiers>*)

Implementation of the Scenic base class `Point`.

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

Properties

- **position** (`Vector`; dynamic) – Position of the point. Default value is the origin.
- **visibleDistance** (*float*) – Distance for `can see` operator. Default value 50.
- **width** (*float*) – Default value zero (only provided for compatibility with operators that expect an `Object`).
- **length** (*float*) – Default value zero.

Note: If you're looking into Scenic's internals, note that *Point* is actually a subclass of the internal Python class *_Constructible*.

class OrientedPoint(<*specifiers*>)

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*.

Properties

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

class Object(<*specifiers*>)

Bases: *scenic.core.object_types.OrientedPoint*

Implementation of the Scenic class `Object`.

This is the default base class for Scenic classes.

Properties

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

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 ‘length’. See other dependencies below.

Allowed forms:

- `ahead of` <oriented point> [`by` <scalar/vector>] – optionally specifies ‘heading’;
- `ahead of` <vector> [`by` <scalar/vector>] – depends on ‘heading’.

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

Behind (*pos*, *dist=0*)
The ‘behind X [by Y]’ polymorphic specifier.

Specifies ‘position’, depending on ‘length’. See other dependencies below.

Allowed forms: behind <oriented point> [by <scalar/vector>] – optionally specifies ‘heading’; behind <vector> [by <scalar/vector>] – depends on ‘heading’.

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

Following (*field*, *dist*, *fromPt=None*)
The ‘following F [from X] for D’ specifier.

Specifies ‘position’, and optionally ‘heading’, with no dependencies.

Allowed forms: following <field> [from <vector>] for <number>

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

exception GuardViolation (*behavior*, *lineno*)
Bases: `Exception`

Abstract exception raised when a guard of a behavior is violated.

This will never be raised directly; either of the subclasses *PreconditionViolation* or *InvariantViolation* will be used, as appropriate.

exception PreconditionViolation (*behavior*, *lineno*)
Bases: `scenic.core.dynamics.GuardViolation`

Raised when a precondition is violated when invoking a behavior.

exception InvariantViolation (*behavior*, *lineno*)
Bases: `scenic.core.dynamics.GuardViolation`

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

class PropertyDefault (*requiredProperties*, *attributes*, *value*)
A default value, possibly with dependencies.

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

class BlockConclusion(*value*)

Bases: `enum.Enum`

An enumeration.

class Modifier(*name*, *value*, *terminator*)

Bases: `tuple`

name: str

Alias for field number 0

value: Any

Alias for field number 1

terminator: Optional[str]

Alias for field number 2

_asdict()

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

classmethod _make(*iterable*)

Make a new Modifier object from a sequence or iterable

_replace(***kwds*)

Return a new Modifier object replacing specified fields with new values

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

scenarioFromFile(*path*, *params={}*, *model=None*, *scenario=None*, *cacheImports=False*)

Compile a Scenic file into a *Scenario*.

Parameters

- **path** (*str*) – path to a Scenic file
- **params** (*dict*) – global parameters to override
- **model** (*str*) – Scenic module to use as world model
- **scenario** (*str*) – if there are multiple scenarios in the file, which one to use
- **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*, *params={}*, *model=None*, *scenario=None*, *filename='<string>'*, *cacheImports=False*)

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

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

1.10 Scenic Libraries

One of the strengths of Scenic is its ability to reuse functions, classes, and behaviors across many scenarios, simplifying the process of writing complex scenarios. This page describes the libraries built into Scenic to facilitate scenario writing by end users.

1.10.1 Simulator Interfaces

Many of the simulator interfaces provide utility functions which are useful when writing scenarios for particular simulators. See the documentation for each simulator on the *Supported Simulators* page, as well as the corresponding module under `scenic.simulators`.

1.10.2 Abstract Domains

To enable cross-platform scenarios which are not specific to one simulator, Scenic defines *abstract domains* which provide APIs for particular application domains like driving scenarios. An abstract domain defines a protocol which can be implemented by various simulator interfaces so that scenarios written for that domain can be executed in those simulators. For example, a scenario written for our *driving domain* can be run in both LGSVL and CARLA.

A domain provides a Scenic world model which defines Scenic classes for the various types of objects that occur in its scenarios. The model also provides a simulator-agnostic way to access the geometry of the simulated world, by defining regions, vector fields, and other objects as appropriate (for example, the driving domain provides a `Network` class abstracting a road network). For domains which support dynamic scenarios, the model will also define a set of simulator-agnostic actions for dynamic agents to use.

Driving Domain

The driving domain, `scenic.domains.driving`, is designed to support scenarios taking place on or near roads. It defines generic classes for cars and pedestrians, and provides a representation of a road network that can be loaded from standard map formats (e.g. OpenDRIVE). The domain supports dynamic scenarios, providing actions for agents which can drive and walk as well as implementations of common behaviors like lane following and collision avoidance. See the documentation of the `scenic.domains.driving` module for further details.

1.11 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:

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

1.11.1 CARLA

Our interface to the CARLA simulator enables using Scenic to describe autonomous driving scenarios. The interface supports dynamic scenarios written using the CARLA world model (*scenic.simulators.carla.model*) as well as scenarios using the cross-platform *Driving Domain*. To use the interface, please follow these instructions:

1. Install the latest version of CARLA (we've tested versions 0.9.9, 0.9.10, and 0.9.11) from the CARLA Release Page.
2. Install Scenic in your Python virtual environment as instructed in *Getting Started with Scenic*.
3. Within the same virtual environment, install CARLA's Python API by executing the following command:

```
$ easy_install /PATH_TO_CARLA_FOLDER/PythonAPI/carla/dist/carla-0.9.9-py3.7-linux-x86_
↪64.egg
```

The exact name of the `.egg` file may vary depending on the version of CARLA you installed; make sure to use the file for Python 3, not 2. You may get an error message saying `Could not find suitable distribution`, which you can ignore. Instead, check that the `carla` package was correctly installed by running **`pip show carla`**.

To start CARLA, run the command **`./CarlaUE4.sh`** in your CARLA folder. Once CARLA is running, you can run dynamic Scenic scenarios following the instructions in *the dynamics tutorial*.

Note: If you are using Scenic 1.x, there is an older CARLA interface which works with static Scenic scenarios and so requires agent behaviors to be written in plain Python. This interface is part of the VerifAI toolkit; documentation and examples can be found in the VerifAI repository.

1.11.2 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.

1.11.3 LGSVL

We have developed an interface to the LGSVL simulator for autonomous driving, used in our ITSC 2020 paper. The interface supports dynamic scenarios written using the LGSVL world model (*scenic.simulators.lgsvl.model*) as well as scenarios using the cross-platform *Driving Domain*.

To use the interface, first install the simulator from the LGSVL Simulator website. Then, within the Python virtual environment where you installed Scenic, install LGSVL's Python API package from source.

An example of how to run a dynamic Scenic scenario in LGSVL is given in *Dynamic Scenarios*.

1.11.4 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!

1.11.5 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.

1.12 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.

1.12.1 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).

Supporting dynamic scenarios requires additionally implementing a subclass of *Simulator* which communicates periodically with your simulator to implement the actions taken by dynamic agents and read back the state of the simulation. See the *scenic.simulators.carla.simulator* and *scenic.simulators.lgsvl.simulator* modules for examples.

1.12.2 Defining a World Model

To make writing scenarios for your simulator easier, you should write a Scenic library specifying all the relevant information about the simulated world. This "world model" could include:

- Scenic classes (subclasses of `Object`) corresponding to types of objects in the simulator;
- instances of `Region` corresponding to locations of interest (e.g. one for each road);
- a `Workspace` specifying legal locations for objects (and optionally providing methods for schematically rendering scenes);
- a set of actions (subclasses of `Action`) which can be taken by dynamic agents during simulations;
- any other information or utility functions that might be useful in scenarios.

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

Each of the simulators natively supported by Scenic has a corresponding `model.scenic` file containing its world model. See the *Supported Simulators* page for links to the module under `scenic.simulators` for each simulator, where the world model can be found. The `scenic.simulators.webots.mars` model is particularly simple and would be a good place to start.

1.13 What's New in Scenic

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

1.13.1 Scenic 2.x

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

Scenic 2.0.0

Backwards-incompatible syntax changes:

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

Major new features:

- under construction…

Minor new features:

- Operators and specifiers which take vectors as arguments will now accept tuples and lists of length 2; for example, you can write `Object at (1, 2)`. The old syntax `Object at 1@2` is still supported.

1.14 Publications Using Scenic

1.14.1 Main Papers

The main paper on Scenic is:

Scenic: A Language for Scenario Specification and Scene Generation.
Fremont, Dreossi, Ghosh, Yue, Sangiovanni-Vincentelli, and Seshia.
PLDI 2019. [full version]
(see also the extended preprint on Scenic 2.0)

An expanded version of this paper appears as Chapters 5 and 8 of this thesis:

Algorithmic Improvisation. [thesis]
Daniel J. Fremont.
Ph.D. dissertation, 2019 (University of California, Berkeley; Group in Logic and the Methodology of Science).

Scenic is also integrated into the VerifAI toolkit, which is described in another paper:

VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems.
Dreossi*, Fremont*, Ghosh*, Kim, Ravanbakhsh, Vazquez-Chanlatte, and Seshia.
CAV 2019.

* Equal contribution.

1.14.2 Case Studies

We have also used Scenic in several industrial case studies:

Formal Analysis and Redesign of a Neural Network-Based Aircraft Taxiing System with VerifAI.
Fremont, Chiu, Margineantu, Osipychev, and Seshia.
CAV 2020.

Formal Scenario-Based Testing of Autonomous Vehicles: From Simulation to the Real World.
Fremont, Kim, Pant, Seshia, Acharya, Bruso, Wells, Lemke, Lu, and Mehta.
ITSC 2020.
[See also this white paper and associated blog post]

1.14.3 Other Papers Building on Scenic

A Programmatic and Semantic Approach to Explaining and Debugging Neural Network Based Object Detectors.
Kim, Gopinath, Pasareanu, and Seshia.
CVPR 2020.

1.15 Credits

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

Scenic is primarily maintained by Daniel J. Fremont.

The Scenic project was started at UC Berkeley in Sanjit Seshia's research group.

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

Edward Kim assisted in developing the library for dynamic driving scenarios and putting together this documentation.

The Scenic tool and example scenarios have benefitted from code contributions from:

- Johnathan Chiu
- Greg Crow
- Francis Indaheng
- Ellen Kalvan
- Martin Jansa (LG Electronics, Inc.)
- Kevin Li
- Guillermo López
- Shalin Mehta
- Joel Moriana
- Gaurav Rao
- Matthew Rhea
- Jay Shenoy
- Wilson Wu

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

- Ankush Desai
- Alastair Donaldson
- Andrew Gordon
- Steve Lemke
- Jonathan Ragan-Kelley
- Sriram Rajamani
- German Ros
- Marcell Vazquez-Chanlatte

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- glossary

CHAPTER THREE

LICENSE

Scenic is distributed under the 3-Clause BSD License.

BIBLIOGRAPHY

[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

D

E

F

G

H

I

K

L

M

N

O

P

R

T

U

V

W