vivarium Documentation

Release 0.9.3

The vivarium developers

Dec 08, 2019

Contents

Vivarium is a framework for building complex simulations using standard scientific Python tools.

CHAPTER 1

Installing Vivarium

1.1 Overview

Vivarium is written in Python and supports Python 3.6+.

1.2 Installation from PyPI

Vivarium packages are published on the Python Package Index. The preferred tool for installing packages from *PyPI* is **`pip`**. This tool is provided with all modern versions of Python

On Linux or MacOS, you should open your terminal and run the following command.

```
$ pip install -U vivarium
```

On Windows, you should open *Command Prompt* and run the same command.

```
C:\> pip install -U vivarium
```

After installation, type **`simulate test`**. This will run a test simulation packaged with the framework and validate that everything is installed correctly.

1.3 Installation from source

You can install Vivarium directly from a clone of the Git repository. You can clone the repository locally and install from the local clone:

```
$ git clone https://github.com/ihmeuw/vivarium.git
$ cd vivarium
$ pip install .
```

You can also install directly from the git repository with pip:

```
$ pip install git+https://github.com/ihmeuw/vivarium.git
```

Additionally, you can download a snapshot of the Git repository in either tar.gz or zip format. Once downloaded and extracted, these can be installed with **`pip`** as above.

CHAPTER 2

Tutorials

Here you'll find a set of tutorials to introduce concepts and tools available in the `vivarium` framework.

Before digging into the tutorials, please walk through the *getting started* guide. It will help you set up a Python package where you'll build your model components.

2.1 Artifact

A data artifact is a bundle of input data associated with a particular model. It is typically stored as an `hdf` file on disk with very particular formatting. This file is then used by the `vivarium` simulations to fill in all the relevant parameter data.

It is frequently useful to be able to view or modify this data outside the simulation. The `vivarium.Artifact` provides a high level interface to do just that. In this tutorial we'll go through how to view, delete, and write data to an artifact using the tools provided by the `Artifact`. You'll access data in the artifact through keys, mirroring the underlying hdf storage of artifacts.

2.1.1 Creating an artifact

To view an existing hdf file via the `Artifact` tools, we'll create a new artifact. We can print the resulting artifact to view the tree structure of the keys in our artifact. We'll use our test artifact to illustrate:

```
from vivarium import Artifact

art = Artifact('test_artifact.hdf')
print(art)
```

```
Artifact containing the following keys:
metadata
        keyspace
        locations
        versions
population
        age_bins
        structure
        theoretical_minimum_risk_life_expectancy
```

Now we have an `Artifact` object, which we can use to interact with the data stored in the hdf file with which we created it.

Filter Terms

The data stored in artifacts may be large, potentially on the order of millions of rows for a single dataset, and loading a full dataset requires time and memory, both of which may be limited. If you are only interested in certain subsets of the data you may want to read only the portion you need. This is the idea behind filter terms.

Filter terms are built into an `Artifact` on its creation and apply to all data loaded from that Artifact. You can think of filter terms as somewhat similar to the `pandas.DataFrame.query()` method, although the key difference is that filter terms apply to what data is actually read off disk. This means that they can reduce the time and memory required to load a single dataset from an Artifact.

Filter terms should be specified as a list of strings, with each item in the list corresponding to a single filter. This allows multiple filters to be applied to a single Artifact. These terms are combined logically using 'AND', so filter terms of `['draw == 0', 'year_start > 2010', 'age_start < 5']` would mean only return rows with `draw == 0 AND year_start > 2010 AND age_start < 5`. Note that if some data stored in your Artifact does not contain the column or columns included in your filter terms, the non-applicable filter terms will be skipped for that data. So if a dataset in an Artifact created with the draw, year_start, and age_start filter terms only included a draw column, only `draw == 0` would be applied to that data.

Here's how we would construct an Artifact with the draw, year_start, and age_start filters we just described:

```
from vivarium_public_health.dataset_manager import Artifact

art = Artifact('test_artifact.hdf', filter_terms=['draw == 0', 'year_start > 2005',
↪'age_start <= 5'])
print(art)
```

```
Artifact containing the following keys:
metadata
        keyspace
        locations
        versions
population
        age_bins
        structure
        theoretical_minimum_risk_life_expectancy
```

Note that the keys in the artifact are unchanged. The filter terms only affect data when it is loaded out of the artifact.

Keys

Artifacts store data under keys. Each key is of the form `<type>.<name>.<measure>`, e.g., "cause.all_causes.restrictions" or `<type>.<measure>`, e.g., "population.structure." To view all keys in an artifact, use the `keys` attribute of the artifact:

```
art.keys
```

```
['metadata.keyspace', 'metadata.locations', 'metadata.versions', 'population.age_bins
↪',
 'population.structure', 'population.theoretical_minimum_risk_life_expectancy']
```

2.1.2 Reading data

Now that we've seen how to create an `Artifact` object and view the underlying storage structure, let's cover how to actually retrieve data from that artifact. We'll use the `load()` method.

We saw the key names in our artifact in the previous step, and we'll use those names to load data. For example, if we want to load the population structure data from our Artifact we do:

```
art = Artifact('test_artifact.hdf')
pop = art.load('population.structure')
print(pop.head())
```

```
                                                          value
age_end  age_start location sex    year_end year_start
0.019178 0.0       Ethiopia Female 2007     2006        25610.50
                            Male   2012     2011        29136.66
                                   2009     2008        27492.91
                            Female 2000     1999        22157.50
                                   1993     1992        19066.45
```

Notice that if we construct our artifact with filter terms as discussed above, we'll filter the data that gets loaded out of it:

```
art = Artifact('test_artifact.hdf', filter_terms=['age_start > 5'])
pop = art.load('population.structure')
print(pop.head())
```

```
                                                            value
age_end age_start location sex    year_end year_start
15.0    10.0      Ethiopia Male   2011     2010        6009393.00
                                   2003     2002        4489336.99
                           Female 2016     2015        6424674.99
                           Male   2017     2016        6610845.00
                           Female 2006     2005        4922733.99
```

We can only load keys that already exist in the Artifact, however. If we try to load a key not present in our Artifact, we will get an error:

```
art.load('a.fake.key')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(continues on next page)

(continued from previous page)

```
  File "/home/kate/code/vivarium/vivarium/src/vivarium/framework/artifact/artifact.py
↪", line 75, in load
    raise ArtifactException(f"{entity_key} should be in {self.path}.")
vivarium.framework.artifact.ArtifactException: a.fake.key should be in tests/dataset_
↪manager/artifact.hdf.
```

2.1.3 Writing data

To write new data to an artifact, use the `write()` method, passing the full key (in the string representation we saw above of `type.name.measure` or `type.measure`) and the data you wish to store.

```
new_data = ['United States', 'Washington', 'California']

art.write('locations.names', new_data)

if 'locations.names' in art:
    print('Successfully Added!')
```

```
Successfully Added!
```

What if the key we wish to write to is already present in the data? Let's see what happens if we try to write again to the `locations.names` key we just wrote to. We get an error:

```
art.write('locations.names', ['New York', 'Florida'])
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/kate/code/vivarium/vivarium/src/vivarium/framework/artifact/artifact.py
↪", line 105, in write
    raise ArtifactException(f'{entity_key} already in artifact.')
vivarium.framework.artifact.ArtifactException: locations.names already in artifact.
```

If the key you want to write to is already in the artifact, you'll want to use the `replace()` method instead of `write()`. This allows you to replace the data in the artifact at the given key with the passed data.

```
updated_data = ['Texas', 'Oregon']
art.replace('locations.names', updated_data)
print(art.load('locations.names'))
```

```
['Texas', 'Oregon']
```

2.1.4 Removing data

Like `load()` and `write()`, `remove()` is based on keys. Pass the name of the key you wish to remove, and it will be deleted from the artifact and the underlying hdf file.

```
art.remove('locations.names')

if not 'locations.names' in art:
    print('Successfully Deleted!')
```

```
Successfully Deleted!
```

2.2 Boids

To get started with agent-based modelling, we'll recreate the classic Boids simulation of flocking behavior. This is a relatively simple example but it produces very pleasing visualizations.

2.2.1 Setup

I'm assuming you've read through the material in *getting started* and are working in your `vivarium_examples` package. If not, you should go there first.

Todo: package setup with __init__ and stuff

2.2.2 Building a population

In many ways, this is a bad place to start. The population component is one of the more complicated components in a simulation as it typically is responsible for bootstrapping some of the more interesting features in Vivarium. What we'll do is start with a simple population and revisit this component as we wish to add more complexity.

Listing 1: **File**: `~/code/vivarium_examples/boids/population.py`

```
import numpy as np
import pandas as pd


class Population:

```

(continues on next page)

(continued from previous page)

```
    configuration_defaults = {
        'population': {
            'colors': ['red', 'blue'],
        }
    }

    def __init__(self):
        self.name = 'population'

    def setup(self, builder):
        self.colors = builder.configuration.population.colors

        columns_created = ['color', 'entrance_time']
        builder.population.initializes_simulants(self.on_initialize_simulants,␣
↪columns_created)
        self.population_view = builder.population.get_view(columns_created)

    def on_initialize_simulants(self, pop_data):
        new_population = pd.DataFrame({
            'color': np.random.choice(self.colors, len(pop_data.index)),
            'entrance_time': pop_data.creation_time,
        }, index=pop_data.index)
        self.population_view.update(new_population)
```

Here we're defining a component that generates a population of 1000 birds. Those birds are then randomly chosen to be either red or blue.

Let's examine what's going on in detail, as you'll see many of the same patterns repeated in later components.

Imports

```
import numpy as np
import pandas as pd
```

NumPy is a library for doing high performance numerical computing in Python. pandas is a set of tools built on top of numpy that allow for fast database-style querying and aggregating of data. Vivarium uses `pandas.DataFrame` objects as it's underlying representation of the population and for many other data storage and manipulation tasks. By convention, most people abbreviate these packages as `np` and `pd` respectively, and we'll follow that convention here.

Population class

Vivarium components are expressed as Python classes. You can find many resources on classes and object-oriented programming with a simple google search. We'll assume some fluency with this style of programming, but you should be able to follow along with most bits even if you're unfamiliar.

Configuration defaults

In most simulations, we want to have an easily tunable set up knobs to adjust various parameters. vivarium accomplishes this by pulling those knobs out as configuration information. Components typically expose the values they use in the `configuration_defaults` class attribute.

```
class Population:

    configuration_defaults = {
        'population': {
            'colors': ['red', 'blue'],
        }
    }
```

We'll talk more about configuration information later. For now observe that we're exposing the size of the population that we want to generate and a set of possible colors for our birds.

The `setup` method

Almost every component in vivarium will have a setup method. The setup method gives the component access to an instance of the *`Builder`* which exposes a handful of tools to help build components. The simulation framework is responsible for calling the setup method on components and providing the builder to them. We'll explore these tools that the builder provides in detail as we go.

```
1     self.name = 'population'
2
3 def setup(self, builder):
4     self.colors = builder.configuration.population.colors
5
6     columns_created = ['color', 'entrance_time']
```

Our setup method is doing three things.

First, it's accessing the subsection of the configuration that it cares about (line). The full simulation configuration is available from the builder as `builder.configuration`. You can treat the configuration object just like a nested python **'dictionary'__** that's been extended to support dot-style attribute access. Our access here mirrors what's in the `configuration_defaults` at the top of the class definition.

Next, we interact with the vivarium's *population management system*.

Note: The Population Table

When we talk about columns in the context of Vivarium, we are typically talking about the simulant *attributes*. Vivarium represents the population of simulants as a single **'pandas DataFrame'__**. We think of each simulant as a row in this table and each column as an attribute of the simulants.

In line 4 we create a variable to hold the names of the columns we want to create and in line 5 we tell the simulation that any time new people get added to the simulation from any component the framework should call the `on_initialize_simulants` function in this component to set the `'entrance_time'` and `'color'` columns for each new simulant.

We'll see a third argument for this function soon and discuss the population management system in more detail.

Next in line 6 we get a view into the population table. *`Population views`* are used both to query the current state of simulants and to update that state information. When you request a population view from the builder, you must tell it which columns in the population table you want to see, and so here we pass along the same set of columns we've said we're creating.

The `on_initialize_simulants` method

Finally we look at the `on_initialize_simulants` method. You can name this whatever you like in practice, but I have a tendency to give methods that the framework is calling names that describe where in the simulation life-cycle they occur. This helps me think more clearly about what's going on and helps debugging.

```
    self.population_view = builder.population.get_view(columns_created)

def on_initialize_simulants(self, pop_data):
    new_population = pd.DataFrame({
        'color': np.random.choice(self.colors, len(pop_data.index)),
        'entrance_time': pop_data.creation_time,
```

We see that like the `setup` method, `on_initialize_simulants` takes in a special argument that we don't provide. This argument, `pop_data` is an instance of *`SimulantData`* containing a handful of information useful when initializing simulants.

The only two bits of information we need for now are the `pop_data.index`, which supplies the index of the simulants to be initialized, and the `pop_data.creation_time` which gives us a representation (typically an `int` or **'pandas Timestamp'__**) of the simulation time when the simulant was generated.

Note: The Population Index

The population table we described before has an index that uniquely identifies each simulant. This index is used in several places in the simulation to look up information, calculate simulant-specific values, and update information about the simulants' state.

Using the population index, we generate a `pandas.DataFrame` on lines 2-5 and fill it with the initial values of 'entrance_time' and 'color' for each new simulant. Right now, this is just a table with data hanging out in our simulation. To actually do something, we have to tell the population management system to update the underlying population table, which we do on line 6.

Putting it together

Vivarium supports both a command line interface and an interactive one. We'll look at how to run simulations from the command line later. For now, we can set up our simulation with the following code:

```
from vivarium import InteractiveContext
from vivarium_examples.boids.population import Population

sim = InteractiveContext(components=[Population()])

# Peek at the population table
print(sim.get_population().head())
```

```
   tracked entrance_time color
0     True    2005-07-01  blue
1     True    2005-07-01   red
2     True    2005-07-01   red
3     True    2005-07-01   red
4     True    2005-07-01   red
```

2.2.3 Position

The classic Boids model introduces three *steering* behaviors into a population of birds and simulates their resulting behavior. For this to work, we need to track the position and velocity of our birds, so let's start there.

Listing 2: **File**: `~/code/vivarium_examples/boids/location.py`

```
class Location:

    configuration_defaults = {
        'location': {
            'width': 1000,   # Width of our field
            'height': 1000,  # Height of our field
        }
    }

    def __init__(self):
        self.name = 'location'

    def setup(self, builder):
        self.width = builder.configuration.location.width
        self.height = builder.configuration.location.height

        columns_created = ['x', 'vx', 'y', 'vy']
        builder.population.initializes_simulants(self.on_create_simulants, columns_
↪created)
        self.population_view = builder.population.get_view(columns_created)

    def on_create_simulants(self, pop_data):
        count = len(pop_data.index)
        # Start clustered in the center with small random velocities
        new_population = pd.DataFrame({
            'x': self.width * (0.4 + 0.2 * np.random.random(count)),
            'y': self.height * (0.4 + 0.2 * np.random.random(count)),
            'vx': -0.5 + np.random.random(count),
```

You'll notice that this looks very similar to our initial population model. Indeed, we can split up the responsibilities of initializing simulants over many different components. In Vivarium we tend to think of components as being responsible for individual behaviors or *attributes*. This makes it very easy to build very complex models while only having to think about local pieces of it.

Let's add this component to our model and look again at the population table.

```
from vivarium import InteractiveContext
from vivarium_examples.boids.population import Population
from vivarium_examples.boids.location import Location

sim = InteractiveContext(components=[Population(), Location()])

# Peek at the population table
print(sim.get_population().head())
```

```
      tracked           x           y        vx        vy entrance_time color
0     True   458.281179  463.086940 -0.473012  0.355904    2005-07-01  blue
1     True   480.864694  596.290448 -0.058006 -0.241146    2005-07-01   red
2     True   406.092503  533.870307  0.299711 -0.041151    2005-07-01  blue
```

(continues on next page)

(continued from previous page)

```
3     True  444.028917  497.491363 -0.005976 -0.491665    2005-07-01   red
4     True  487.670224  412.832049 -0.145613 -0.123138    2005-07-01  blue
```

Our population now has initial position and velocity!

2.2.4 Visualizing our population

Now is also a good time to come up with a way to plot our birds. We'll later use this to generate animations of our birds flying around. We'll use matplotlib for this.

Making good visualizations is hard, and beyond the scope of this tutorial, but the `matplotlib` documentation has a large number of examples and **'tutorials <https://matplotlib.org/tutorials/index.html'__** that should be useful.

For our purposes, we really just want to be able to plot the positions of our birds and maybe some arrows to indicated their velocity.

Listing 3: **File**: `~/code/vivarium_examples/boids/visualization.py`

```
import matplotlib.pyplot as plt


def plot_birds(simulation, plot_velocity=False):
    width = simulation.configuration.location.width
    height = simulation.configuration.location.height
    pop = simulation.get_population()

    plt.figure(figsize=[12, 12])
    plt.scatter(pop.x, pop.y, color=pop.color)
    if plot_velocity:
        plt.quiver(pop.x, pop.y, pop.vx, pop.vy, color=pop.color, width=0.002)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.axis([0, width, 0, height])
    plt.show()
```

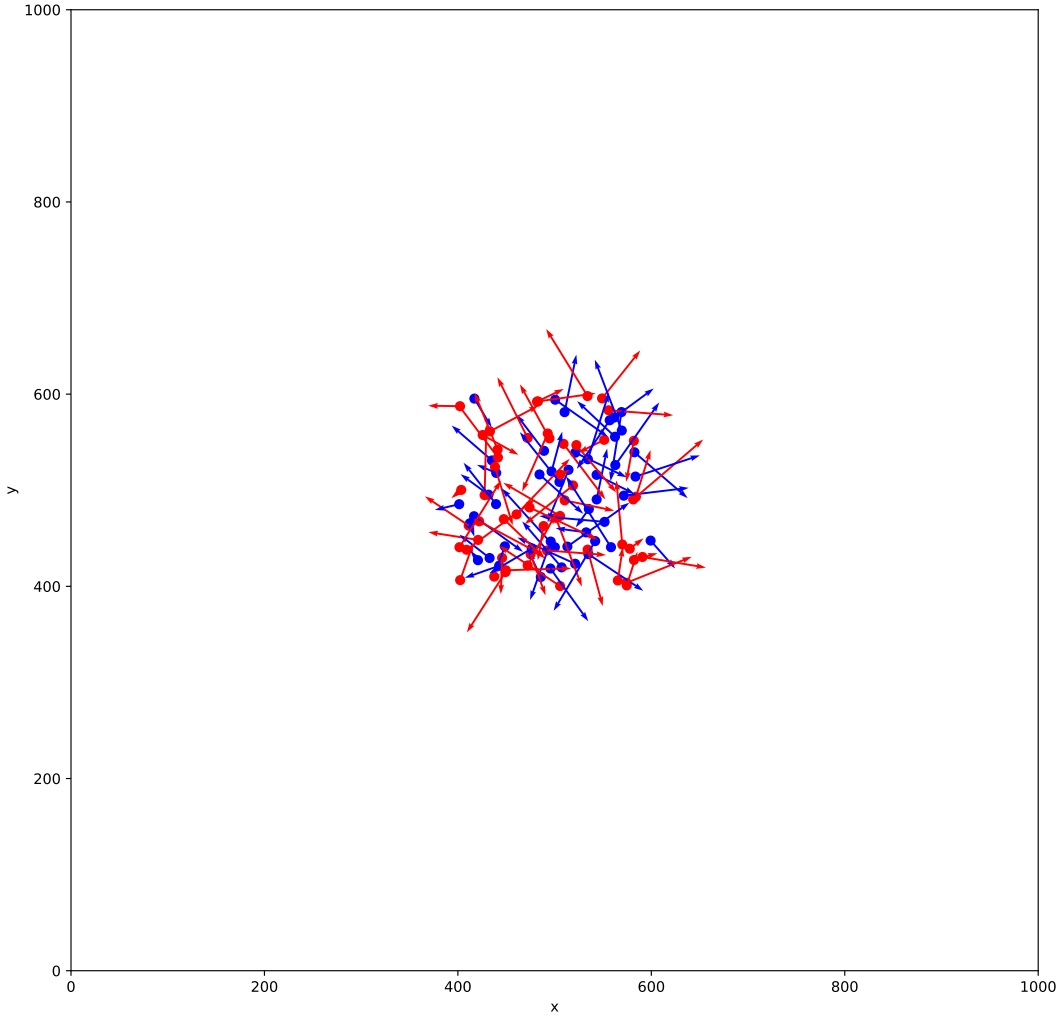
We can then visualize our flock with

```
from vivarium import InteractiveContext
from vivarium_examples.boids.population import Population
from vivarium_examples.boids.location import Location
from vivarium_examples.boids.visualization import plot_birds

sim = InteractiveContext(components=[Population(), Location()])

plot_birds(sim, plot_velocity=True)
```

2.2.5 Calculating Neighbors

The steering behavior in the Boids model is dictated by interactions of each bird with its nearby neighbors. A naive implementation of this can be very expensive. Luckily, Python has a ton of great libraries that have solved most of the hard problems.

Here, we'll pull in a **'KDTree'__** from SciPy and use it to build a component that tells us about the neighbor relationships of each bird.


1000
800
600
400
200
0
y
0
200
400
600
800
1000
x

Listing 4: **File**: ~/code/vivarium_examples/boids/neighbors.py

```
import pandas as pd
from scipy import spatial


class Neighbors:

    configuration_defaults = {
        'neighbors': {
            'radius': 10
        }
    }

    def __init__(self):
        self.name = 'Neighbors'

    def setup(self, builder):
        self.radius = builder.configuration.neighbors.radius

        self.neighbors_calculated = False
        self._neighbors = pd.Series()
        self.neighbors = builder.value.register_value_producer('neighbors',␣
↪source=self.get_neighbors)

        builder.population.initializes_simulants(self.on_create_simulants)
        self.population_view = builder.population.get_view(['x', 'y'])

        builder.event.register_listener('time_step', self.on_time_step)

    def on_create_simulants(self, pop_data):
        self._neighbors = pd.Series([[]] * len(pop_data.index), index=pop_data.index)

    def on_time_step(self, event):
        self.neighbors_calculated = False

    def get_neighbors(self, index):
        if not self.neighbors_calculated:
            self.calculate_neighbors()
        return self._neighbors[index]

    def calculate_neighbors(self):
        # Reset our list of neighbors
        pop = self.population_view.get(self._neighbors.index)
        self._neighbors = pd.Series([[]] * len(pop), index=pop.index)

        tree = spatial.KDTree(pop)

        # Iterate over each pair of simulates that are close together.
        for boid_1, boid_2 in tree.query_pairs(self.radius):
            # .iloc is used because query_pairs uses 0,1,... indexing instead of␣
↪pandas.index
            self._neighbors.iloc[boid_1].append(self._neighbors.index[boid_2])
            self._neighbors.iloc[boid_2].append(self._neighbors.index[boid_1])
```

Todo:

- Describe rationale for neighbors component
- Start building behavior components
- Build animation component

2.3 Disease Model

Todo: Motivate the development of the disease model. We're trying to understand the impact of interventions.

Here we'll produce a data-free disease model focusing on core Vivarium concepts. You can find more complicated versions of the *components* built here in the vivarium_public_health library. Those components must additionally deal with manipulating complex data which makes understanding what's going on more complicated.

After this tutorial, you should be well poised to begin working with and examining those components.

2.3.1 Setup

I'm assuming you've read through the material in *getting started* and are working in your `vivarium_examples` package. If not, you should go there first.

Todo: package setup with __init__ and stuff

2.3.2 Building a population

In many ways, this is a bad place to start. The population component is one of the more complicated components in the simulation as it typically is responsible for bootstrapping some of the more interesting features in vivarium.

We need a population though. So we'll start with one here and defer explanation of some of the more complex pieces/systems until later.

Listing 5: **File**: `~/code/vivarium_examples/disease_model/population.py`

```
import pandas as pd

from vivarium.framework.engine import Builder
from vivarium.framework.population import SimulantData
from vivarium.framework.event import Event


class BasePopulation:
    """Generates a base population with a uniform distribution of age and sex.

    Attributes
    ----------
    configuration_defaults :
        A set of default configuration values for this component. These can be
        overwritten in the simulation model specification or by providing
        override values when constructing an interactive simulation.
    """

    configuration_defaults = {
        'population': {
            # The range of ages to be generated in the initial population
            'age_start': 0,
            'age_end': 100,
            # Note: There is also a 'population_size' key.
        },
    }

    def __init__(self):
        self.name = 'base_population'

    def setup(self, builder: Builder):
        """Performs this component's simulation setup.

        The ``setup`` method is automatically called by the simulation
        framework. The framework passes in a ``builder`` object which
        provides access to a variety of framework subsystems and metadata.
```

(continues on next page)

(continued from previous page)

```
        Parameters
        ----------
        builder :
            Access to simulation tools and subsystems.
        """
        self.config = builder.configuration

        self.with_common_random_numbers = bool(self.config.randomness.key_columns)
        self.register = builder.randomness.register_simulants
        if (self.with_common_random_numbers
                and not ['entrance_time', 'age'] == self.config.randomness.key_
↪columns):
            raise ValueError("If running with CRN, you must specify ['entrance_time',
↪'age'] as"
                             "the randomness key columns.")

        self.age_randomness = builder.randomness.get_stream('age_initialization',
                                                            for_initialization=self.
↪with_common_random_numbers)
        self.sex_randomness = builder.randomness.get_stream('sex_initialization')

        columns_created = ['age', 'sex', 'alive', 'entrance_time']
        builder.population.initializes_simulants(self.on_initialize_simulants,
                                                 creates_columns=columns_created)

        self.population_view = builder.population.get_view(columns_created)

        builder.event.register_listener('time_step', self.age_simulants)

    def on_initialize_simulants(self, pop_data: SimulantData):
        """Called by the simulation whenever new simulants are added.

        This component is responsible for creating and filling four columns
        in the population state table:

        'age' :
            The age of the simulant in fractional years.
        'sex' :
            The sex of the simulant. One of {'Male', 'Female'}
        'alive' :
            Whether or not the simulant is alive. One of {'alive', 'dead'}
        'entrance_time' :
            The time that the simulant entered the simulation. The 'birthday'
            for simulants that enter as newborns. A `pandas.Timestamp`.

        Parameters
        ----------
        pop_data :
            A record containing the index of the new simulants, the
            start of the time step the simulants are added on, the width
            of the time step, and the age boundaries for the simulants to
            generate.

        """

        age_start = self.config.population.age_start
        age_end = self.config.population.age_end
```

(continues on next page)

(continued from previous page)

```
        if age_start == age_end:
            age_window = pop_data.creation_window / pd.Timedelta(days=365)
        else:
            age_window = age_end - age_start

        age_draw = self.age_randomness.get_draw(pop_data.index)
        age = age_start + age_draw * age_window

        if self.with_common_random_numbers:
            population = pd.DataFrame({'entrance_time': pop_data.creation_time,
                                       'age': age.values}, index=pop_data.index)
            self.register(population)
            population['sex'] = self.sex_randomness.choice(pop_data.index, ['Male',
↪'Female'])
            population['alive'] = 'alive'
        else:
            population = pd.DataFrame(
                {'age': age.values,
                 'sex': self.sex_randomness.choice(pop_data.index, ['Male', 'Female
↪']),
                 'alive': pd.Series('alive', index=pop_data.index),
                 'entrance_time': pop_data.creation_time},
                index=pop_data.index)

        self.population_view.update(population)

    def age_simulants(self, event: Event):
        """Updates simulant age on every time step.

        Parameters
        ----------
        event :
            An event object emitted by the simulation containing an index
            representing the simulants affected by the event and timing
            information.
        """
        population = self.population_view.get(event.index, query="alive == 'alive'")
        population['age'] += event.step_size / pd.Timedelta(days=365)
        self.population_view.update(population)
```

There are a lot of things here. Let's take them piece by piece. (*Note*: I'll be leaving out the docstrings in the code snippets below).

Imports

```
import pandas as pd

from vivarium.framework.engine import Builder
from vivarium.framework.population import SimulantData
from vivarium.framework.event import Event
```

Aside from `pandas`, we also import three classes from the core Vivarium framework here. We'll use them to provide typing information in method signatures.

Note: Providing type hints in Python totally optional, but if you're using a modern python IDE or plugins for

traditional text editors, they can offer you completion options and easy access to interface documentation. It also enables the use of other static analysis tools like mypy.

Default Configuration

```
class BasePopulation:

    configuration_defaults = {
        'population': {
            # The range of ages to be generated in the initial population
            'age_start': 0,
            'age_end': 100,
            # Note: There is also a 'population_size' key.
        },
    }
```

You'll see this sort of pattern repeated in many, many Vivarium components.

We declare a configuration block as a class attribute for components. Vivarium has a *cascading configuration system* that aggregates configuration data from many locations. The configuration is essentially a declaration of the parameter space for the simulation.

The most important thing to understand is that configuration values are given default values provided by the components and that they can be overriden with a higher level system like a command line argument later.

In this component in particular declares defaults for the age range for the initial population of simulants. It also notes that there is a *'population_size'* key. This key has a default value set by Vivarium's population management system.

The `__init__()` method

Though Vivarium components are represented are represented by Python classes you'll notice that many of the classes have very sparse `__init__` methods. Due to the way the simulation bootstraps itself, the `__init__` method is usually only used to assign names to generic components and muck with the `configuration_defaults` a bit. We'll see more of this later.

The `setup` method

Instead of the `__init__` method, most of the component initialization takes place in the `setup` method.

```
1  def setup(self, builder: Builder):
2      self.config = builder.configuration
3
4      self.with_common_random_numbers = bool(self.config.randomness.key_columns)
5      self.register = builder.randomness.register_simulants
6      if (self.with_common_random_numbers
7              and not ['entrance_time', 'age'] == self.config.randomness.key_columns):
8          raise ValueError("If running with CRN, you must specify ['entrance_time', 'age
   ↪'] as"
9                           "the randomness key columns.")
10
11     self.age_randomness = builder.randomness.get_stream('age_initialization',
12                                                         for_initialization=self.with_
   ↪common_random_numbers)
13     self.sex_randomness = builder.randomness.get_stream('sex_initialization')
```

(continues on next page)

(continued from previous page)

```
14
15      columns_created = ['age', 'sex', 'alive', 'entrance_time']
16      builder.population.initializes_simulants(self.on_initialize_simulants,
17                                               creates_columns=columns_created)
18
19      self.population_view = builder.population.get_view(columns_created)
20
21      builder.event.register_listener('time_step', self.age_simulants)
```

The signature for the `setup` method is the same in every component. When the framework is constructing the simulation it looks for a `setup` method on each component and calls that method with a *`Builder`* instance.

Note: The Builder

The `builder` object is essentially the simulation toolbox. It provides access to several simulation subsystems:

- `builder.configuration` : A dictionary-like representation of all of the parameters in the simulation.
- `builder.lookup` : A service for generating interpolated lookup tables. We won't use these in this tutorial.
- `builder.value` : The value pipeline system. In many ways this is the heart of any Vivarium simulation. We'll discuss this in great detail as we go.
- `builder.event` : Access to Vivarium's event system. The primary use is to register listeners for `'time_step'` events.
- `builder.population` : The population management system. Registers population initializers (functions that fill in initial state information about simulants), give access to views of the simulation state, and mediates updates to the simulation state. It also provides access to functionality for generating new simulants (e.g. via birth or migration), though we won't use that feature in this tutorial.
- `builder.randomness` : Vivarium uses a variance reduction technique called Common Random Numbers to perform counterfactual analysis. In order for this to work, the simulation provides a centralized source of randomness.
- `builder.time` : The simulation clock.
- `builder.components` : The component management system. Primarily used for registering subcomponents for setup.

Let's step through the `setup` method and examine what's happening.

Line 2 simply grabs a copy of the simulation *`configuration`*. This is essentially a dictionary that supports `.`-access notation.

```
2       self.config = builder.configuration
```

Lines 4-13 interact with Vivarium's *`randomness system`*. Several things are happening here.

Lines 4-9 deal with the topic of *Common Random Numbers*, a variance reduction technique employed by the Vivarium framework to make it easier to perform counterfactual analysis. It's not important to have a full grasp of this system at this point.

```
4       self.with_common_random_numbers = bool(self.config.randomness.key_columns)
5       self.register = builder.randomness.register_simulants
6       if (self.with_common_random_numbers
7               and not ['entrance_time', 'age'] == self.config.randomness.key_columns):
```

(continues on next page)

(continued from previous page)

```
8           raise ValueError("If running with CRN, you must specify ['entrance_time', 'age
↪'] as"
9                            "the randomness key columns.")
```

Note: Common Random Numbers

The idea behind Common Random Numbers (or CRN) is to enable comparison between two simulations running under slightly different conditions. Conceptually, we achieve this by guaranteeing that the same events occur to the same people at the same time across simulations with the same random seed.

For example, suppose we have two simulations of the world. We model the world as it is in the first simulation and we introduce a vaccine for the flu in the second simulation. Unless my model explicitly encodes the causal relationship between flu vaccination and vehicle traffic patterns, the person who died in a vehicle accident on the 43rd time step in the first simulation will also die in a vehicle accident on the 43rd time step in the second simulation.

In practice, what the CRN system requires is a way to uniquely identify simulants across simulations. We need to randomly generate some simulant characteristics in a repeatable fashion and then use those characteristics to identify the simulants in the randomness system later. This is **only** handled by the population component typically. It's vitally important to get right when doing counterfactual analysis, but it's not especially important that you understand the mechanics of the implementation.

In this component we're using some information about the configuration of the randomness system to let us know whether or not we care about using CRN. We'll explore this much later when we're looking at running simulations with interventions.

The next thing we do is grab actual *randomness streams* from the framework.

```
11      self.age_randomness = builder.randomness.get_stream('age_initialization',
12                                                           for_initialization=self.with_
↪common_random_numbers)
13      self.sex_randomness = builder.randomness.get_stream('sex_initialization')
```

`get_stream` is the only call most components make to the randomness system. The best way to think about randomness streams is as decision points in your simulation. Any time you need to answer a question that requires a random number, you should be using a randomness stream linked to that question.

Here we have the questions "What age are my simulants when they enter the simulation?" and "What sex are my simulants?" and streams to go along with them.

The `for_initialization` argument tells the stream that the simulants you're asking this question about won't already be registered with the randomness system. This is the bootstrapping part. Here we're using the `'entrance_time'` and `'age'` to identify a simulant and so we need a stream to initialize ages with. There is should really only be one of these initialization streams in a simulation.

The `'sex_randomness'` is a much more typical example of how to interact with the randomness system.

Next we register the `on_initialize_simulants` method of our `BasePopulation` object as a population initializer and let the *population management system* know that it is responsible for generating the `'age'`, `'sex'`, `'alive'`, and `'entrance_time'` columns in the population state table.

```
15      columns_created = ['age', 'sex', 'alive', 'entrance_time']
16      builder.population.initializes_simulants(self.on_initialize_simulants,
17                                               creates_columns=columns_created)
```

Note: The Population Table

When we talk about columns in the context of Vivarium, we are typically talking about the simulant *attributes*. Vivarium represents the population of simulants as a single **'pandas DataFrame'__**. We think of each simulant as a row in this table and each column as an attribute of the simulants.

Next we get a view into the population table.

```
19      self.population_view = builder.population.get_view(columns_created)
```

Population views are used both to query the current state of simulants and to update that state information. When you request a population view from the builder, you must tell it which columns in the population table you want to see, and so here we pass along the same set of columns we've said we're creating.

Finally, we register the `age_simulants` method as a listener to the `'time_step'` event using the *event system*. Vivarium emits several *events* over the course of the simulation. Any time the `'time_step'` event is called, the `age_simulants` method will be called as well.

```
21      builder.event.register_listener('time_step', self.age_simulants)
```

That was a lot of stuff

As I mentioned at the top the population component is one of the more complicated pieces of any simulation. It's not important to grasp everything right now. We'll see many of the same patterns repeated in the `setup` method of other components later. The unique things here are worth coming back to at a later point once you have more familiarity with the framework conventions.

The `on_initialize_simulants` method

During `setup`, we registered this method with the framework as a simulant initializer. You can name this whatever you like in practice, but I have a tendency to give methods that the framework is calling names that describe where in the simulation life-cycle they occur. This helps me think more clearly about what's going on and helps debugging.

```
1   def on_initialize_simulants(self, pop_data: SimulantData):
2       age_start = self.config.population.age_start
3       age_end = self.config.population.age_end
4       if age_start == age_end:
5           age_window = pop_data.creation_window / pd.Timedelta(days=365)
6       else:
7           age_window = age_end - age_start
8
9       age_draw = self.age_randomness.get_draw(pop_data.index)
10      age = age_start + age_draw * age_window
11
12      if self.with_common_random_numbers:
13          population = pd.DataFrame({'entrance_time': pop_data.creation_time,
14                                     'age': age.values}, index=pop_data.index)
15          self.register(population)
16          population['sex'] = self.sex_randomness.choice(pop_data.index, ['Male',
    ↪'Female'])
17          population['alive'] = 'alive'
18      else:
19          population = pd.DataFrame(
20              {'age': age.values,
21               'sex': self.sex_randomness.choice(pop_data.index, ['Male', 'Female']),
22               'alive': pd.Series('alive', index=pop_data.index),
23               'entrance_time': pop_data.creation_time},
24              index=pop_data.index)
```

(continues on next page)

(continued from previous page)

```
25
26      self.population_view.update(population)
```

Every initializer is called by the population management whenever simulants are created. For our purposes, this happens only once at the very beginning of the simulation. Typically, we'd task another component with responsibility for managing other ways simulants might enter (we might, for instance, have a `Migration` component that knows about how and when people enter and exit our location of interest).

The population management system uses information about what columns are created by which components in order to determine what order to call initializers defined in separate classes. We'll see what this means in practice later.

We see that like the `setup` method, `on_initialize_simulants` takes in a special argument that we don't provide. This argument, `pop_data` is an instance of *`SimulantData`* containing a handful of information useful when initializing simulants.

Note: SimulantData

This simple structure only has four attributes (used here in the generic Python sense of the word).

- `index` : The population table index of the simulants being initialized.
- `user_data` : A (potentially empty) dictionary generated by the user in components that directly create simulants.
- `creation_time` : The current simulation time. A `pandas.Timestamp`.
- `creation_window` : The size of the time step over which the simulants are created. A `pandas.Timedelta`.

We'll take this method line by line as we did with `setup`.

The most interesting thing that that the `BasePopulation` component does is manage the age of our simulants. Back in the `configuration_defaults` we specified an `'age_start'` and `'age_end'`. Here we use these to generate the age distribution of our initial population.

```
2       age_start = self.config.population.age_start
3       age_end = self.config.population.age_end
4       if age_start == age_end:
5           age_window = pop_data.creation_window / pd.Timedelta(days=365)
6       else:
7           age_window = age_end - age_start
8
9       age_draw = self.age_randomness.get_draw(pop_data.index)
10      age = age_start + age_draw * age_window
```

We've built in support for two different kinds of populations based on the `'age_start'` and `'age_end'` specified in the configuration. If we get the same `'age_start'` and `'age_end'`, we have a cohort, and so we smear out ages within the width of a single time step (the `creation_window`). Otherwise, we assume our population is uniformly distributed within the age window bounded by `'age_start'` and `'age_end'`. You can use demographic data here to generate arbitrarily complex starting populations.

The only thing really of note here is the call to `self.age_randomness.get_draw`. If we recall from the `setup` method, `self.age_randomness` is an instance of a *`RandomnessStream`* which supports several convenience methods for interacting with random numbers. `get_draw` takes in an `index` representing particular simulants and returns a `pandas.Series` with a uniformly drawn random number for each simulant in the index.

Note: The Population Index

The population table we described before has an index that uniquely identifies each simulant. This index is used in several places in the simulation to look up information, calculate simulant-specific values, and update information about the simulants' state.

We then come back to the question of whether or not we're using common random numbers in our system. In the `setup` method, our criteria for using common random numbers was that `'entrance_time'` and `'age'` were specified as the randomness `key_columns` in the configuration. These `key_columns` are what the randomness system uses to uniquely identify simulants across simulations.

```
    if self.with_common_random_numbers:
        population = pd.DataFrame({'entrance_time': pop_data.creation_time,
                                   'age': age.values}, index=pop_data.index)
        self.register(population)
        population['sex'] = self.sex_randomness.choice(pop_data.index, ['Male',
↪'Female'])
        population['alive'] = 'alive'
```

If we are using CRN, we must generate these columns before any other calls are made to the randomness system with the population index. We then register these simulants with the randomness system using `self.register`, a reference to `register_simulants` method in the randomness management system. This is responsible for mapping the attributes of interest (here `'entrance_time'` and `'age'`) to a particular set of random numbers that will be used across simulations with the same random seed.

Once registered, we can generate the remaining attributes of our simulants with guarantees around reproducibility.

If we're not using CRN, we can just generate the full set of simulant attributes straightaway.

```
    else:
        population = pd.DataFrame(
            {'age': age.values,
             'sex': self.sex_randomness.choice(pop_data.index, ['Male', 'Female']),
             'alive': pd.Series('alive', index=pop_data.index),
             'entrance_time': pop_data.creation_time},
            index=pop_data.index)
```

In either case, we are hanging on to a table representing some attributes of our new simulants. However, this table does not matter yet because the simulation's population system doesn't know anything about it. We must first inform the simulation by passing in the `DataFrame` to our *`population view's`* `update` method. This method is the only way to modify the underlying population table.

Warning: The data generated and passed into the population view's `update` method must have the same index that was passed in with the `pop_data`. You can potentially cause yourself a great deal of headache otherwise.

Aging our simulants

The last piece of our population component is the `'time_step'` listener method `age_simulants`.

```
def age_simulants(self, event: Event):
    population = self.population_view.get(event.index, query="alive == 'alive'")
    population['age'] += event.step_size / pd.Timedelta(days=365)
    self.population_view.update(population)
```

This method takes in an *`Event`* argument provided by the simulation. This is very similar to the `SimulantData` argument provided to `on_initialize_simulants`. It carries around some information about what's happening in the event.

Note: Event

The event also has four attributes.

- `index` : The population table index of the simulants responding to the event.
- `user_data` : A (potentially empty) dictionary generated by the user in components that directly events.
- `time` : The current simulation time. A `pandas.Timestamp`.
- `step_size` : The size of the time step we're about to take. A `pandas.Timedelta`.

It also supports some method for generating new events that we don't care about here.

In order to age our simulants, we first acquire a copy of the current population state from our population view. In addition to the `update` method, population views also support a `get` method that takes in an index and an optional `query` used to filter down the returned population. Here, we only want to increase the age of people still living. The `query` argument needs to be consistent with the **'query'__** method of a `pandas.DataFrame`.

What we get back is another `pandas.DataFrame` containing the filtered rows corresponding to the index we passed in. The columns of the returned `DataFrame` are precisely the columns we specified when we created the view.

We next update the age of our simulants by adding on the width of the time step to their current age and passing the update table to the `update` method of our population view as we did in `on_initialize_simulants`

Examining our work

Now that we've done all this hard work, let's see what it gives us.

```
from vivarium import InteractiveContext
from vivarium_examples.disease_model.population import BasePopulation

config = {'randomness': {'key_columns': ['entrance_time', 'age']}}

sim = InteractiveContext(components=[BasePopulation()], configuration=config)

print(sim.get_population().head())
```

```
   tracked  alive     sex        age entrance_time
0     True  alive    Male  78.088109    2005-07-01
1     True  alive    Male  44.072665    2005-07-01
2     True  alive  Female  48.346571    2005-07-01
3     True  alive  Female  91.002147    2005-07-01
4     True  alive  Female  63.641191    2005-07-01
```

Great! We generate a population with a non-trivial age and sex distribution. Let's see what happens when our simulation takes a time step.

```
sim.step()
print(sim.get_population().head())
```

```
      tracked  alive     sex        age entrance_time
0     True  alive    Male  78.090849    2005-07-01
1     True  alive    Male  44.075405    2005-07-01
2     True  alive  Female  48.349311    2005-07-01
3     True  alive  Female  91.004887    2005-07-01
4     True  alive  Female  63.643931    2005-07-01
```

Everyone get's older! Right now though, we could just keep taking steps in our simulation and people would continue getting older. This, of course, does not reflect how the world goes. Time to introduce the grim reaper.

2.3.3 Mortality

Now that we have population generation and aging working, the next step is introducing mortality into our simulation.

Listing 6: **File**: `~/code/vivarium_examples/disease_model/mortality.py`

```
import numpy as np
import pandas as pd

from vivarium.framework.engine import Builder
from vivarium.framework.event import Event


class Mortality:
    """Introduces death into the simulation.

    Attributes
    ----------
    configuration_defaults :
        A set of default configuration values for this component. These can be
        overwritten in the simulation model specification or by providing
        override values when constructing an interactive simulation.
    """

    configuration_defaults = {
        'mortality': {
            'mortality_rate': 0.01,
        }
    }

    def __init__(self):
        self.name = 'mortality'

    def setup(self, builder: Builder):
        """Performs this component's simulation setup.

        The ``setup`` method is automatically called by the simulation
        framework. The framework passes in a ``builder`` object which
        provides access to a variety of framework subsystems and metadata.

        Parameters
        ----------
        builder :
            Access to simulation tools and subsystems.
        """
        self.config = builder.configuration.mortality
        self.population_view = builder.population.get_view(['alive'], query="alive ==
↪'alive'")
        self.randomness = builder.randomness.get_stream('mortality')

        self.mortality_rate = builder.value.register_rate_producer('mortality_rate',␣
↪source=self.base_mortality_rate)
```

(continues on next page)

(continued from previous page)

```
        builder.event.register_listener('time_step', self.determine_deaths)

    def base_mortality_rate(self, index: pd.Index) -> pd.Series:
        """Computes the base mortality rate for every individual.

        Parameters
        ----------
        index :
            A representation of the simulants to compute the base mortality
            rate for.

        Returns
        -------
            The base mortality rate for all simulants in the index.
        """
        return pd.Series(self.config.mortality_rate, index=index)

    def determine_deaths(self, event: Event):
        """Determines who dies each time step.

        Parameters
        ----------
        event :
            An event object emitted by the simulation containing an index
            representing the simulants affected by the event and timing
            information.
        """
        effective_rate = self.mortality_rate(event.index)
        effective_probability = 1 - np.exp(-effective_rate)
        draw = self.randomness.get_draw(event.index)
        affected_simulants = draw < effective_probability
        self.population_view.update(pd.Series('dead', index=event.index[affected_
↪simulants]))
```

The purpose of this component is to determine who dies every time step based on a mortality rate. You'll see many of the same framework features we used in the `BasePopulation` component used again here and a few new things.

Let's dive in.

What's new in the configuration?

Since we're building our disease model without data to inform it, we'll expose all the important bits of the model as parameters in the configuration.

```
class Mortality:
    configuration_defaults = {
        'mortality': {
            'mortality_rate': 0.01,
        }
    }
```

Here we're specifying the overall mortality rate in our simulation. Rates have units! We'll phrase our model with rates specified in terms of events per person-year. So here we're specifying a uniform mortality rate of 0.01 deaths per person-year. This is obviously not realistic. Using toy data like this is often extremely useful in validating a model though.

Setting up the mortality component

Many of the tools we explored in the `BasePopulation` component are used again here. There are two new things to look at.

```
1  def setup(self, builder: Builder):
2      self.config = builder.configuration.mortality
3      self.population_view = builder.population.get_view(['alive'], query="alive ==
   ↪'alive'")
4      self.randomness = builder.randomness.get_stream('mortality')
5
6      self.mortality_rate = builder.value.register_rate_producer('mortality_rate',␣
   ↪source=self.base_mortality_rate)
7
8      builder.event.register_listener('time_step', self.determine_deaths)
```

The first comes in line 3. Previously, we'd acquired a population view from the builder and then supplied a query to filter out dead people when we were requesting the population table from the view. We can also provide a default query when we construct the view and bypas the query argument when requesting the population table from the view later. In line 3 we're saying we want a view of the `'alive'` column of the population table, but only for those people who are actually alive in the current time step.

The other feature of note is is the introduction of the *`values system`* in line 6. The values system provides a way of distributing the computation of a value over multiple components. This is a bit difficult to get used to, but is vital to the way we think about components in Vivarium. The best way to understand this system is by *example.*

In our current context we introduce a named value "pipeline" into the simulation called `'mortality_rate'`. The source for a value is always a callable function or method. It typically takes in a `pandas.Index` as its only argument. Other things are possible, but not necessary for our current use case.

The `'mortality_rate'` source is then responsible for returning a `pandas.Series` containing a base mortality rate for each simulant in the index to the values system. Other components may register themselves as modifiers to this base rate. We'll see more of this once we get to the disease modelling portion of the tutorial.

The value system will coordinate how the base value is modified behind the scenes and return the results of all computations wherever the pipeline is called from (here, in the soon to be discussed `determine_deaths` method.

Supplying a base mortality rate

As just discussed, the `base_mortality_rate` method is the source for the `'mortality_rate'` value. Here we take in an index and build a `pandas.Series` that assigns each individual the mortality rate specified in the configuration.

```
1  def base_mortality_rate(self, index: pd.Index) -> pd.Series:
2      return pd.Series(self.config.mortality_rate, index=index)
```

In an actual simulation, we'd inform the base mortality rate with data specific to the age, sex, location, year (and potentially other demographic factors) that represent each simulant. We might disaggregate or interpolate our data here as well. Which is all to say, the source of a data pipeline can do some pretty complicated stuff.

Determining who dies

Like our aging method in the population component, our `determine_deaths` method responds to `'time_step'` events.

```
1 def determine_deaths(self, event: Event):
2     effective_rate = self.mortality_rate(event.index)
3     effective_probability = 1 - np.exp(-effective_rate)
4     draw = self.randomness.get_draw(event.index)
5     affected_simulants = draw < effective_probability
6     self.population_view.update(pd.Series('dead', index=event.index[affected_
  ↪simulants]))
```

Line 2 is where we actually call the pipeline we constructed during setup. It will return the effective mortality rate for each person in the simulation. Right now this will just be the base mortality rate, but we'll see how this changes once we bring in a disease. Importantly for now though, the pipeline is automatically rescaling the rate down to the size of the time steps we're taking.

In lines 3-5, we determine who died this time step. We turn our mortality rate into a probability of death in the given time step by assuming deaths are exponentially distributed and using the inverse distribution function. We then draw a uniformly distributed random number for each person and determine who died by comparing that number to the computed probability of death for the individual.

Finally, in line 6, we update the state table with the newly dead simulants.

Did it work?

It's a good time to check and make sure that what we did works. We've got a mortality rate of 0.01 deaths per person-year and we're taking 1 day time steps, so we give ourselves a relatively large population this time so we can see the impact of our mortality component without taking too many steps.

```
from vivarium InteractiveContext
from vivarium_examples.disease_model.population import BasePopulation
from vivarium_examples.disease_model.mortality import Mortality

config = {
    'population': {
        'population_size': 100_000
    },
    'randomness': {
        'key_columns': ['entrance_time', 'age']
    }
}

sim = InteractiveContext(components=[BasePopulation(), Mortality()],␣
  ↪configuration=config)
print(sim.get_population().head())
```

```
   tracked  alive     sex        age entrance_time
0     True  alive    Male  78.088109    2005-07-01
1     True  alive    Male  44.072665    2005-07-01
2     True  alive  Female  48.346571    2005-07-01
3     True  alive  Female  91.002147    2005-07-01
4     True  alive  Female  63.641191    2005-07-01
```

This looks (exactly!) the same as last time. Good.

```
sim.get_population().alive.value_counts()
```

```
alive    100000
Name: alive, dtype: int64
```

Just checking that everyone is alive. Let's run our simulation for a while and see what happens.

```
sim.take_steps(365)  # Run for one year with one day time steps
sim.get_population().alive.value_counts()
```

```
alive    99037
dead       963
Name: alive, dtype: int64
```

We simulated somewhere between 99,037 (if everyone died in the first time step) and 100,000 (if everyone died in the last time step) living person-years and saw 963 deaths. This means our empirical mortality rate is somewhere close to 0.0097 deaths per person-year, very close to the 0.01 rate we provided.

2.3.4 Observer

In a real simulation, we typically want to record sophisticated output. We also frequently work in non-interactive (or even distributed) environments where we don't have easy access to the simulation object.

2.3.5 Disease

2.3.6 Risk

2.3.7 Intervention

2.3.8 Running from the command line

2.3.9 Exploring some results

2.4 Exploring a Simulation in an Interactive Setting

In other tutorials *[1] [2]* we've walked through how to build components for simulations. We've also shown how to run those simulations from the *command line* and in an *interactive setting*.

In this tutorial we'll focus on exploring simulations in an interactive setting. The only prerequisite is that you've set up your programming environment (See *the getting started section*). We'll look at how to examine the *population table*, how to print and interpret the simulation *configuration*, and how to get values from the *value pipeline* system.

We'll work through all this with a few case studies using the simulations built in the other tutorials.

2.4.1 What Are We Looking At?

Simulations are complicated things. It's beyond the scope of this tutorial in particular to talk about what they are and how they work and when they make sense as models of the world. Luckily, once you have one in hand, you can start figuring out the answers to many of those questions yourself.

In the case studies that follow, we'll start simply. We'll get our simulations *setup* in an interactive environment. We'll then examine various aspects of the simulation state at the beginning of the simulation. We'll then run them for a while and see how that state changes over time. After we have a handle on examining different aspects of the simulation, we'll take a step back to talk about what our expectations should be about how the simulation should work and look at some examples of how to test those expectations. Finally, we'll setup a comparison across two simulations to examine how changing our *configuration parameters* alters what happens in a simulation.

2.4.2 Case Study #1: Population Epidemiology

In this case study, we're going to put together and examine an individual-based epidemiology model from a bunch of pre-constructed parts. We'll start out rather mechanically, just showing how to set up and run a simulation and pull out interesting data. As we go on, we'll talk about what sort of results we should expect from the structure of the model and how we can verify those expectations.

Getting Things Set Up

Before we can start exploring properties of the simulation, we need to get our hands on a simulation *context*. This is the object we'll use to examine and run our simulation model. You can check out our tutorial on *setting up a simulation* to see the tools that `vivarium` provides for building your own simulation context objects. For this tutorial on exploring simulations, however, we've provided a convenience function to get you started. In a Jupyter notebook or python interpreter, you can run the following

```
from vivarium.examples.disease_model import get_disease_model_simulation

sim = get_disease_model_simulation()
```

The `sim` object returned here is our simulation context. With it, we're ready to begin examining various aspects of the simulation state.

Checking Out the Configuration

One of the things we might want to look at is the simulation *configuration*. Typically, a *model specification* encodes some configuration information, but leaves many things set to defaults. We can see what's in the configuration by simply printing it.

```
print(sim.configuration)
```

```
randomness:
    key_columns:
        model_override: ['entrance_time', 'age']
    map_size:
        component_configs: 1000000
    random_seed:
        component_configs: 0
    additional_seed:
        component_configs: None
time:
```

(continues on next page)

(continued from previous page)

```
    start:
        year:
            model_override: 2005
        month:
            model_override: 7
        day:
            model_override: 1
    end:
        year:
            model_override: 2006
        month:
            model_override: 7
        day:
            model_override: 1
    step_size:
        model_override: 3
population:
    population_size:
        model_override: 10000
    age_start:
        model_override: 0
    age_end:
        model_override: 30
mortality:
    mortality_rate:
        model_override: 0.05
    life_expectancy:
        model_override: 80
diarrhea:
    incidence_rate:
        model_override: 2.5
    remission_rate:
        model_override: 42
    excess_mortality_rate:
        model_override: 12
child_growth_failure:
    proportion_exposed:
        model_override: 0.5
effect_of_child_growth_failure_on_infected_with_diarrhea.incidence_rate:
    relative_risk:
        model_override: 5
effect_of_child_growth_failure_on_infected_with_diarrhea.excess_mortality_rate:
    relative_risk:
        model_override: 5
breastfeeding_promotion:
    effect_size:
        model_override: 0.5
interpolation:
    order:
        component_configs: 0
    extrapolate:
        component_configs: True
input_data:
    artifact_path:
        component_configs: None
    artifact_filter_term:
        component_configs: None
```

(continues on next page)

(continued from previous page)

```
    input_draw_number:
        component_configs: None
    location:
        component_configs: None
```

What do we see here? The configuration is *hierarchical*. There are a set of top level *keys* that define named subsets of configuration data. We can access just those subsets if we like.

```
print(sim.configuration.randomness)
```

```
key_columns:
    model_override: ['entrance_time', 'age']
map_size:
    component_configs: 1000000
random_seed:
    component_configs: 0
additional_seed:
    component_configs: None
```

This subset of configuration data contains more keys. All of the keys in our example here (key_columns, map_size, random_seed, and additional_seed) point directly to values. We can access these values from the simulation as well.

```
print(sim.configuration.randomness.key_columns)
print(sim.configuration.randomness.map_size)
print(sim.configuration.randomness.random_seed)
print(sim.configuration.randomness.additional_seed)
```

```
['entrance_time', 'age']
1000000
0
None
```

However, we can no longer modify the configuration since the simulation has already been setup.

```
from vivarium.config_tree import ConfigurationError

try:
    sim.configuration.randomness.update({'random_seed': 5})
except ConfigurationError:
    print("Can't update configuration after setup")
```

```
Can't update configuration after setup
```

If we look again at the randomness configuration, it appears that there should be one more layer of keys.

```
key_columns:
    model_override: ['entrance_time', 'age']
map_size:
    component_configs: 1000000
random_seed:
    component_configs: 0
additional_seed:
    component_configs: None
```

This last layer reflects a priority level in the way simulation configuration is managed. The `component_configs` under `map_size`, `random_seed`, and `additional_seed` tells us that the value was set by a simulation com-

ponent's `configuration_defaults`. The `model_override` under `key_columns` tells us that a model specification file set the value. If you're trying to debug issues, you may want more information than this. You can also type `repr(sim.configuration)` (this is the equivalent of evaluating `sim.configuration` in a jupyter notebook or ipython cell). This will give you considerable information about where each configuration value was set and at what priority level. You can read more about how the configuration works in the *configuration concept section*

Looking at the Simulation Population

Another interesting thing to look at at the beginning of the simulation is your starting population.

```
pop = sim.get_population()
print(pop.head())
```

```
   tracked     sex  alive        age entrance_time  child_growth_failure_propensity
↪             diarrhea
0     True  Female  alive   3.452598    2005-06-28                         0.552276
↪susceptible_to_diarrhea
1     True  Female  alive   4.773249    2005-06-28                         0.019633
↪susceptible_to_diarrhea
2     True    Male  alive  23.423383    2005-06-28                         0.578892
↪susceptible_to_diarrhea
3     True  Female  alive  13.792463    2005-06-28                         0.988650
↪susceptible_to_diarrhea
4     True    Male  alive   0.449368    2005-06-28                         0.407759
↪susceptible_to_diarrhea
```

This gives you a `pandas.DataFrame` representing your starting population. You can use it to check all sorts of characteristics about individuals or the population as a whole.

```
count    10000.000000
mean        15.076739
std          8.707947
min          0.000118
25%          7.526312
50%         15.004701
75%         22.687441
max         29.998082
Name: age, dtype: float64
alive    10000
Name: alive, dtype: int64
count    10000.000000
mean         0.503233
std          0.288712
min          0.000002
25%          0.255108
50%          0.504047
75%          0.752366
max          0.999943
Name: child_growth_failure_propensity, dtype: float64
susceptible_to_diarrhea    10000
Name: diarrhea, dtype: int64
2005-06-28    10000
Name: entrance_time, dtype: int64
Female    5015
Male      4985
Name: sex, dtype: int64
```

(continues on next page)

(continued from previous page)

```
True    10000
Name: tracked, dtype: int64
```

Understanding the Simulation Data

Our model starts with a bunch of people with uniformly distributed ages and sexes. They march through time 3 days at a time (we'll vary this later) in discrete steps. On each step for each person, the simulation will ask and answer several questions: Did they die? Did they get sick? If they were sick, did they recover? Are they exposed to any risks? At the end we'll examine how many people died and compare that with a theoretical life expectancy. Later, we'll consider two simulations that differ only by the presence of a new intervention and examine how effective that intervention is.

Todo: Show how to understand the starting population from both the configuration and the population state table. Show how to understand the simulation time and how the clock progresses based on configuration parameters.

2.4.3 Case Study #2: Boids

Todo: Everything

2.5 Getting Started

We're about to walk through building several sets of components for simulations. `vivarium` supports interactive use, so we could do all of that in a single script or in an interactive environment like a jupyter notebook. This makes sharing code or working collaboratively very difficult, however.

Instead, we'll first walk through some tips and tools that will help us keep things organized and make collaboration easy. It will also enable us to run simulations from the command line. This is vital for any serious simulation work.

2.5.1 Organizing our work

Instead of storing our work in one or several python files, we'll store it in a python package. A python package is a bundle of structured python code that a user can install. This is the key to making python's `import` statement work.

To start off, pick a place for your work on your computer. I'll pretend we're working in `~/code/`.

Note: `~/` is general shorthand for the user's home directory. It would be something like `C:\Users\<YourUserName>\` on Windows, `/home/<YourUserName>/` on Linux, or `/Users/<YourUserName>/` on Mac.

In this directory, make a subdirectory called `vivarium_examples`. This will be our package directory. We then want to generate the following directory structure

```
~/code/vivarium_examples/
    src/
        vivarium_examples/
            __init__.py
    setup.py
```

The `__init__.py` file can be blank. It's the file that tells python that `vivarium_examples` is a package. We need to fill out the `setup.py` file though.

Listing 7: **File**: `~/code/vivarium_examples/setup.py`

```
from setuptools import setup, find_packages


if __name__ == "__main__":

    setup(
        name='vivarium_examples',
        version='1.0',
        description="Examples of simulations built with vivarium",
        author=''  # YOUR NAME HERE,

        package_dir={'': 'src'},
        packages=find_packages(where='src'),
        include_package_data=True,

        install_requires=['vivarium'],
)
```

This is the file that lets us install your package and import it from anywhere. We'll use it shortly.

2.5.2 Version control

2.5.3 Making an environment

The next thing we'll do is set up a programming environment. This is like a clean room for your code and all the code it depends on. It helps

2.5.4 Installing your library

2.5.5 Next steps

2.6 Running a Simulation

Once you've built a *model specification* that describes the *plugins*, *components*, and *configuration* for the model you want to run, it's time to run the simulation defined by that model specification using `vivarium`. There are two ways to run a single simulation.

2.6.1 Running from the Command Line

To run from the command line, we'll use the `vivarium.interface.cli.simulate()` command. This command is actually a group containing three sub-commands: `run`, `test`, and `profile`. We will focus on `run` here.

The basic use of `simulate run` requires no more than a *model specification* yaml file. With this, you can do the following to run the model defined by that specification:

```
simulate run /path/to/your/model/specification.yaml
```

By default, `simulate run` will output whatever results your model produces to the `~/vivarium_results` directory.

Note: `~` is used as a shortcut to represent the user's home directory on a file system. If you're on Windows, this probably looks something like `C:\Users\YOUR_NAME` while on linux it would be `/home/YOUR_NAME`.

If you navigate to that directory, you should see a subdirectory with the name of your model specification. Inside the model specification directory, there will be another subdirectory named for the start time of the run. In here, you should see two hdf files: `final_state.hdf`, which is the population *state table* at the end of the simulation, and `output.hdf`, which is the results of the *metrics* generated by the simulation.

For example, say we've run a simulation for a model specification called `potatoes.yaml` (maybe we're really into gardening). Our directory tree will look like:

```
~/vivarium_results/
    potatoes/
        2019_04_20_15_44_20/
            final_state.hdf
            output.hdf
```

If we decide we don't like our results, or want to rerun the simulation with a different set of *configuration parameters*, we'll add new time stamped sub-directories to our `potatoes` model results directory:

```
~/vivarium_results/
    potatoes/
        2019_04_20_15_44_20/
            final_state.hdf
            output.hdf
        2019_04_20_16_34_12/
            final_state.hdf
            output.hdf
```

`simulate run` also provides various flags which you can use to configure options for the run. These are:

Table 1: **Available** `simulate run` **options**

Option	Description
–results-directory or **-o**	The top-level directory in which to write results. Within this directory, a subdirectory named to match the model-specification file will be created. Within this, a further subdirectory named for the time at which the run was started will be created.
–verbose or **-v**	Report each time step as it occurs during the run.
–log	A path at which a log file should be created.
–pdb	If an error occurs, drop into the python debugger.

Let's illustrate how to use them. Say we run the following:

```
simulate run /path/to/your/model/specification -o /path/to/output/directory --log /
↪path/to/log/file --pdb -v
```

Let's walk through how each of these flags will change the behavior from our initial plain `simulate run`. First, we have specified an output directory via the **-o** flag. In our first example, outputs went to `~/vivarium_results`. Now they will go to our specified directory. Second, we have also provided a path to a log file via **–log** at which we can find the log outputs of our simulation run. Next, we have provided the **–pdb** flag so that if something goes wrong in our run, we will drop into the python debugger where we can investigate. Finally, we have turned on the verbose option via the **-v** flag. Whereas before, we saw nothing printed to the console while our simulation was running, we will now see something like the following:

```
DEBUG:vivarium.framework.values:Registering PopulationManager.metrics as modifier to␣
↪metrics
DEBUG:vivarium.framework.values:Registering value pipeline mortality_rate
DEBUG:vivarium.framework.values:Registering value pipeline metrics
DEBUG:vivarium.framework.values:Unsourced pipelines: []
DEBUG:vivarium.framework.engine:2005-07-01 00:00:00
DEBUG:vivarium.framework.engine:2005-07-04 00:00:00
DEBUG:vivarium.framework.engine:2005-07-07 00:00:00
DEBUG:vivarium.framework.engine:2005-07-10 00:00:00
DEBUG:vivarium.framework.engine:2005-07-13 00:00:00
DEBUG:vivarium.framework.engine:{'simulation_run_time': 0.7717499732971191,
 'total_population': 10000,
 'total_population_tracked': 10000,
 'total_population_untracked': 0}
DEBUG:vivarium.framework.engine:Some configuration keys not used during run: {'input_
↪data.cache_data', 'output_data.results_directory', 'input_data.intermediary_data_
↪cache_path'}
```

The specifics of these messages will depend on your model specification, but you should see a series of timestamps

that correspond to the time steps the simulation takes as it runs your model.

2.6.2 Running a Simulation Interactively

In this tutorial, we'll walk through getting a simulation up and running in an interactive setting such as a Python interpreter or Jupyter notebook.

Running a simulation in this way is useful for a variety of reasons, foremost for debugging and validation work. It allows for changing simulation *configuration* programmatically, stepping through a simulation in a controlled fashion, and examining the *state* of the simulation itself.

For the following tutorial, we will assume you have set up an environment and installed `vivarium`. If you have not, please see the *Getting Started* section. We'll be using the *disease model* constructed in a separate tutorial here, though no background knowledge of population health is necessary to follow along. The *components* constructed in that tutorial are available in the `vivarium` package, so you don't need to build them yourself before starting this tutorial.

Setting up a Simulation

To run a simulation interactively, we will need to create a *`simulation context`*. At a bare minimum, we need to provide the context with a set of *components* that encode all the behavior of the simulation model. Frequently, we'll also provide some *configuration* data that is used to parameterize those components.

Note: We can also optionally provide a set of *plugins* to the simulation framework. Plugins are special components that add new functionality to the framework itself. This is an advanced feature for building tools to adapt `vivarium` to models in a particular problem domain and not important for most users.

The combination of components, configuration, and plugins forms a *model specification*, a complete description of a `vivarium` model.

The *`InteractiveContext`* can be generated from several different kinds of data and may be generated at two separate *lifecycle* stages. We'll explore several examples of generating simulation objects here.

With a Model Specification File - The Automatic Way

A *model specification* file contains all the information needed to prepare and run a simulation, so to get up and running quickly, we need only provide this file. You typically find yourself in this use case if you already have a well-developed model and you're looking to explore its behavior in more detail than you'd be able to using the command line utility *simulate*.

In this example, we will use the model specification from our *disease model* tutorial:

Listing 8: **File**: `disease_model.yaml`

```
components:
    vivarium.examples.disease_model:
        population:
            - BasePopulation()
        mortality:
            - Mortality()
        observer:
            - Observer()
        disease:
            - SIS_DiseaseModel('diarrhea')
        risk:
            - Risk('child_growth_failure')
            - DirectEffect('child_growth_failure', 'infected_with_diarrhea.incidence_
↪rate')
            - DirectEffect('child_growth_failure', 'infected_with_diarrhea.excess_
↪mortality_rate')
        intervention:
            - MagicWandIntervention('breastfeeding_promotion', 'child_growth_failure.
↪proportion_exposed')

configuration:
    randomness:
        key_columns: ['entrance_time', 'age']
    time:
        start:
            year: 2005
            month: 7
            day: 1
        end:
            year: 2006
            month: 7
            day: 1
        step_size: 3  # Days
    population:
        population_size: 10_000
        age_start: 0
        age_end: 30
    mortality:
        mortality_rate: 0.05
        life_expectancy: 80
    diarrhea:
        incidence_rate: 2.5  # Approximately 2.5 cases per person per year.
        remission_rate: 42  # Approximately 6 day median recovery time
        excess_mortality_rate: 12  # Approximately 22 % of cases result in death
    child_growth_failure:
        proportion_exposed: 0.5
    effect_of_child_growth_failure_on_infected_with_diarrhea.incidence_rate:
        relative_risk: 5
    effect_of_child_growth_failure_on_infected_with_diarrhea.excess_mortality_rate:
        relative_risk: 5
    breastfeeding_promotion:
        effect_size: 0.5
```

Generating a simulation from a model specification is very straightforward, as it is the primary use case.

```
from vivarium import InteractiveContext
p = "/path/to/disease_model.yaml"
sim = InteractiveContext(p)
```

In order to make it easier to follow along with this tutorial, we've provided a convenience function to get the path to the disease model specification distributed with `vivarium`.

```
from vivarium import InteractiveContext
from vivarium.examples.disease_model import get_model_specification_path

p = get_model_specification_path()
sim = InteractiveContext(p)
```

The `sim` object produced here is all set up and ready to run if you want to jump directly to the *running the simulation* section.

Without a Model Specification File - The Manual Way

It is possible to prepare a simulation by explicitly passing in the instantiated objects you wish to use rather than getting them from a *model specification* file. This method requires initializing all the model components and building the simulation configuration by hand. This requires a lot of boilerplate code but is frequently very useful during model development and debugging.

To demonstrate this, we will recreate the simulation from the *disease_model.yaml* specification without using the actual file itself.

Components

We will first instantiate the *components* necessary for the simulation. In this case, we will get them directly from the disease model example and we will place them in a normal Python list.

```
from vivarium.examples.disease_model import (BasePopulation, Mortality, Observer,
                                             SIS_DiseaseModel, Risk, DirectEffect,
                                             MagicWandIntervention)

components = [BasePopulation(),
              Mortality(),
              SIS_DiseaseModel('diarrhea'),
              Risk('child_growth_failure'),
              DirectEffect('child_growth_failure', 'infected_with_diarrhea.incidence_
↪rate'),
              DirectEffect('child_growth_failure', 'infected_with_diarrhea.excess_
↪mortality_rate'),
              MagicWandIntervention('breastfeeding_promotion', 'child_growth_failure.
↪proportion_exposed'), ]
```

Configurations

We also need to create a dictionary for the *configuration* information for the simulation. Components will typically have defaults for many of these parameters, so this dictionary will contain all the parameters we want to change (or that we want to show are available to change).

```
config = {
    'randomness': {
        'key_columns': ['entrance_time', 'age'],
    },
    'population': {
        'population_size': 10_000,
    },
    'diarrhea': {
        'incidence_rate': 2.5,         # Approximately 2.5 cases per person per year.
        'remission_rate': 42,          # Approximately 6 day median recovery time
        'excess_mortality_rate': 12,   # Approximately 22 % of cases result in death
    },
    'child_growth_failure': {
        'proportion_exposed': 0.5,
    },
    'effect_of_child_growth_failure_on_infected_with_diarrhea.incidence_rate': {
        'relative_risk': 5,
    },
    'effect_of_child_growth_failure_on_infected_with_diarrhea.excess_mortality_rate': {
        'relative_risk': 5,
    },
    'breastfeeding_promotion': {
        'effect_size': 0.5,
    },
}
```

Setting up

With our components and configuration in hand, we can then set up the simulation in a very similar manner as before.

```
from vivarium import InteractiveContext
sim = InteractiveContext(components=components, configuration=config)
```

Typically when you're working this way, you're not trying to load in and parameterize so many components, so it's usually not this bad. You typically only want to do this if you're building a new simulation from scratch.

With this final step, you can proceed directly to *running the simulation*, or stick around to see one last way to set up the simulation in an interactive setting.

Modifying an Existing Simulation

Another frequent use case is when you're trying to add on to an already existing simulation. Here you'll want to grab a prebuilt simulation before the *setup phase* so you can add extra components or modify the configuration data. You then have to call setup on the simulation yourself.

To do this we'll set the `setup` flag in the `InteractiveContext` to `False`.

```
from vivarium import InteractiveContext
from vivarium.examples.disease_model import get_model_specification_path

p = get_model_specification_path()
sim = InteractiveContext(p, setup=False)
```

This function returns a simulation object that has not been setup yet so we can alter the configuration programmatically, if we wish. Let's alter the population size to be smaller so the simulation takes less time to run.

```
sim.configuration.update({'population': {'population_size': 1_000}})
```

We then need to call the `setup()` method on the simulation context to prepare it to run.

```
sim.setup()
```

After this step, we are ready to *run the simulation*.

Note: While this is a kind of trivial example, this last use case is extremely important. Practically speaking, the utility of initializing the simulation without setting it up is that it allows you to alter the configuration data and components in the simulation before it is run or examined. This is frequently useful for setting up specific configurations for validating the simulation from a notebook or for reproducing a particular set of configuration parameters that produce unexpected outputs.

```
from vivarium import InteractiveContext
from vivarium.examples.disease_model import get_model_specification_path

p = get_model_specification_path()
sim = InteractiveContext(p, setup=False)
sim.configuration.update({'population': {'population_size': 1_000}})
sim.setup()
```

Bonus: Adding Additional Components

Another use case for creating the `InteractiveContext` in its pre-setup state is to extend existing models.

For example, say we wanted to add another risk for unsafe water sources into our disease model. We could do the following.

```
from vivarium import InteractiveContext
from vivarium.examples.disease_model import get_model_specification_path, Risk,␣
↪DirectEffect

p = get_model_specification_path()
sim = InteractiveContext(p, setup=False)

sim.add_components([Risk('unsafe_water_source'),
                    DirectEffect('unsafe_water_source', 'infected_with_diarrhea.
↪incidence_rate')])

sim.configuration.update({
    'unsafe_water_source': {
        'proportion_exposed': 0.3
    },
    'effect_of_unsafe_water_source_on_infected_with_diarrhea.incidence_rate': {
        'relative_risk': 8,
    },
})

sim.setup()
```

This is an easy way to take an old model and toy around with new components to immediately see their effects.

Running the Simulation

A simulation can be run in several ways once it is set up. The simplest way to advance a simulation is to call *sim.run()* on it, which will advance it from its current time to the end time specified in the simulation *configuration*. If you need finer control, there are a set of methods on the context that allow you to run the simulation for specified spans of time or numbers of simulation steps. Below is a table of the functions that can be called on an InteractiveContext <vivarium.interface.interactive.InteractiveContext to advance a simulation in different ways.

Table 2: **InteractiveContext methods for advancing simulations**

Method	Description
run	Run the simulation for its entire duration, from its current time to its end time. The start time and end time are specified in the time block of the configuration.
step	Advance the simulation one step. The step size is taken from the time block of the configuration.
take_steps	Advance the simulation n steps.
run_until	Advance the simulation to a specific time. This time should make sense given the simulation's clock type.
run_for	Advance the simulation for a duration. This duration should make sense given the simulation's clock type.

CHAPTER 3

Concepts

Here we cover several core conceptual topics related to modeling with the Vivarium framework.

3.1 The Builder

3.1.1 Outline

- The setup method
- The builder wraps up services
- Detail of services

3.2 What is a Component?

3.2.1 Outline

- Components represent aspects and behaviors of simulants
- Components are classes
- `configuration_defaults`

- An `__init__` trick for generic components.
- The `setup` method.
  - Load data here
  - No simulants exist yet
- Write a good `__repr__`

3.3 The Configuration System

3.3.1 Outline

- Configuration is hierarchical
- Configuration sources and priorities
- Specifying defaults.

3.4 Common Random Numbers

3.4.1 Outline

- Variance reduction
- Counterfactual Analysis

3.5 Data in the Simulation

3.5.1 Outline

- The 3 different kinds of data.
- Model inputs and the value system
- State information and the population system
- Model parameters and the configuration system.

3.6 Simulation Entry Points

`vivarium` provides a single main entry point, the *`SimulationContext`*, that is then wrapped for use on the command line and in interactive settings. This document describes the main entry point and the wrappers and gives an indication about how you might parallelize the simulations to run on multiple CPUs. The purpose here is to describe architecture and guarantees. For tutorials on running simulations, see the *tutorials section*.

3.6.1 The Vivarium Engine

The *`engine`* houses the *`SimulationContext`* – the key `vivarium` object for running and interacting with simulations. It is the top-level manager for all state information in `vivarium`. All simulations are created by a call to the `__init__` of the *`SimulationContext`* at some level and wrappers around the context should try to be as thin as possible around simulation creation.

The context accepts four arguments:

model_specification The *model specification* is a complete representation of a `vivarium` simulation formatted as a yaml file. As an argument ot the *`SimulationContext`*, it can be provided as a path to a file (either as a `str` or a `pathlib.Path`) or as a *`ConfigTree`*, the internal representation of configuration information used by `vivarium`. The model specification contains three pieces, each represented by the next three arguments. For more information about model specifications and their formatting, see the associated *concept note*.

components *Components* provide the logical structure of a `vivarium` simulation. They are python classes that interact with the framework via the *builder*. Components may be provided to the context as a list of instantiated objects, as a dictionary representation of their import paths, or as a *`ConfigTree`* representation of their import paths. The latter two representations are treated as prefix trees when they are parsed into objects. This behavior is controlled by the `ComponentConfigurationParser`. More information about components is available in the component *concept note*.

configuration The *configuration* is the set of variable model parameters in a `vivarium` simulation. It may be provided as a dictionary or *`ConfigTree`* representation. See the *concept note* for more information.

plugins *Plugins* represent core functionality and subsystems of a `vivarium` simulation. Users may wish to extend the functionality of the framework by writing their own plugins. The framework then needs to be notified of their names and where they are located. Plugins may be specified as either a dictionary or *`ConfigTree`* and are parsed into objects by the *`PluginManager`*. This is an advanced feature and almost never necessary.

The `configuration` and `plugins` arguments are treated as overrides for anything provided in the `model_specification`. This allows easy modification of a simulation defined in a model specification file.

> **Warning:** If you provide `components` as a `dict` or *`ConfigTree`*, these will also be treated as overrides, though this is almost never the intended use case, so tread cautiously.

By intention, the context exposes a very simple interface for managing the *simulation lifecycle*. The combination of initializing and running the simulation is encapsulated in the *`run_simulation`* command also available in the *`engine`*.

The simulation *`Builder`* is also part of the engine. It is the main interface that components use to interact with the simulation framework. You can read more about how the builder works and what services it exposes *here*.

3.6.2 Public Interfaces

Functionality in the the *vivarium.framework.engine* serves as the lowest level entry point into the simulation, but common use cases demand more usability. In the vivarium.interface subpackage we have two public interfaces for interacting with the simulation.

The *vivarium.interface.cli* module provides the simulate command and sub-commands for running and profiling simulations from the command line. A complete tutorial is available *here*. simulate restricts the user to work only with *model specification* files and so is primarily useful in a workflow where the user is modifying that file directly to run simulations. Results are deposited in the ~/vivarium_results folder by default, though a command line flag allows the user to specify different output directories.

During model development and debugging, it is frequently more useful to work in an interactive setting like a jupyter notebook or a Python REPL. For this sort of work, the *vivarium.interface.interactive* module provides the InteractiveContext <vivarium.interface.interactive.InteractiveContext (also available as a top-level import from vivarium). Details about the many ways to initialize and run a simulation using the interactive context are available in the *interactive tutorial*.

vivarium itself does not provide tools for running simulations in a distributed system, mostly because each cluster is unique. However, many common simulation tasks will require running many variations of the same simulation (parameter searches, intervention analysis, uncertainty analysis, etc.). For an example of a distributed system built on top of vivarium, see the vivarium_cluster_tools package and its associated documentation.

3.7 The Event System

vivarium constructs and manages the flow of *time* through the emission of regularly scheduled events. This event system provides a means of coordinating across various components in a simulation.

3.7.1 What is an Event?

An *Event* is a simple container for a group of attributes that provide all the necessary information to respond to the event. Events have the following attributes:

Table 1: **Event Attributes**

Name	Description
index	An index into the population table that contains all individuals that may respond to the event.
time	The time at which the event will resolve. The current simulation time plus the size of the current time step.
step_size	A `Timedelta` or `int` representing the size of the next simulation time step.
user_data	A `dict` that allows an event emitter to pass arbitrary information to event listeners.

`vivarium` manages these events with a publication-subscriber system. The framework maintains several named event channels where it and user *components* can emit (or publish) events. User components may then register methods or functions as *listeners* (or subscribers) to events on a particular channel. The *listeners* will then be called any time an event is emitted on the channel and will receive the event as an argument.

User components can also create new event channels simply by requesting an *emitter* or *listener* from the event system. This behavior is discouraged, however, as it makes understanding the structure of a model much more difficult.

3.7.2 Lifecycle Events

The simulation engine itself is responsible for the emission of a dedicated set of events that determine the progression of time in the simulation. Each time step in the simulation corresponds to the emission of a set of `time_step` events. *Components* can register themselves as listeners to these events and thus take action on each time step or other key simulation phase.

The following table depicts the events emitted by the simulation engine itself and the lifecycle phase when they are emitted. See the *lifecycle concept note* for more information about these phases.

Table 2: **Events Emitted by the Simulation Engine**

Lifecycle Phase	Events
Post-setup	`post_setup`

Main Event Loop	`time_step_prepare`
	`time_step`
	`time_step__cleanup`
	`collect_metrics`

Finalization	`simulation_end`

3.7.3 Interacting with Events

The *`EventInterface`* is available off the *Builder* and provides two options for interacting with the event system:

1. `register_listener` to add a listener to a given event to be called on emission

2. *`get_emitter`* to retrieve a callable emitter for a given event

Although methods for both getting emitters and registering listeners are provided, it is strongly encouraged that only the registering listeners aspect is used.

Registering Listeners

In order to register a listener to an event to respond when that event is emitted, we can use the `register_listener`. The listener itself should be a callable function that takes as its only argument the *`Event`* that is emitted.

Suppose we wish to track how many simulants are affected each time step. We could do this by creating an observer component with an `on_time_step` method that we will register as a listener for the `time_step` event. Our component would look something like the following:

```
class AffectedObserver:

    def setup(self, builder):
        self.affected_counts = pd.DataFrame(columns=['time_step', 'number_affected])
        builder.event.register_listener('time_step', self.on_time_step)

    def on_time_step(self, event):
        self.affected_counts.append(pd.DataFrame({'time_step': event.time, 'number_
↪affected': len(event.index)}))
```

On each time step, our `on_time_step` method will be called and we will add another row to our dataframe tracking the number of affected simulants.

Note: Listeners are stored in priority levels when registered to an event. These levels (0-9) indicate which order listeners should be called when the event is emitted; listeners in lower priority levels will be called earlier. Within a priority level, there is no guarantee of order.

This feature should be avoided if possible. Components should strive to obey the Markov property as they transform the state table: the state of the simulation at the beginning of the next time step should only depend on the current state of the system.

Emitting Events

The *get_emitter* provides a way to get a callable emitter for a given named event. It can be used as follows:

```
emitter = builder.event.get_emitter('my_event')
```

Danger: Do not emit any of the simulation lifecyle events described in the table above. These are events that correspond to particular phases in the simulation and should only be emitted by the engine itself.

Caution: While users may provide their own named events by requesting an emitter, this is not advised. Adding additional events beyond those emitted by the simulation engine essentially creates arbitrary `GOTO` statements in the simulation flow and makes time much more difficult to think about.

3.8 Interpolating Data

Todo: Everything here.

3.9 The Simulation Lifecycle

The life cycle of a `vivarium` simulation is a representation of the different execution **states** and their transitions. An execution state is a clearly delineated execution period during the simulation around which we build and enforce concrete programmatic contracts. These states can be grouped into five important **phases**. The phases are closely related groups of execution states. Contracts are not enforced directly around phases, but they are a useful tool for thinking about the execution flow during a simulation.

Table 3: **Life Cycle Phases**

Phase	Description
Bootstrap	Arguments passed from the simulation *entry point* are parsed and core framework systems are initialized.
Initialization	Simulation managers and *components* are initialized and all *configuration* information is loaded.
Setup	Components register themselves with simulation services and request access to resources by interacting with the simulation *builder*; the initial population is created.
Main Loop	The core logic (as encoded in the simulation components) is executed.
Simulation End	The population state is finalized and results are tabulated.

The simulation itself maintains a formal representation of its internal execution state using the tools in the `lifecycle` module. The tools allow the simulation to make concrete contracts about flow of execution in simulations and to constrain the availability of certain framework services to particular life cycle states. This makes error handling much more robust and allows users to more easily reason about complex simulation models.

Todo: Make a graph of service availability in the simulation.

3.9.1 The Bootstrap Phase

The bootstrap and initialization phases look like an atomic operation to an external user. Bootstrap only exists as a separate phase because certain operations must take place before the internal representation of the simulation life cycle exists.

During bootstrap, all user input arguments are parsed into an internal representation of the simulation *plugins*, *components*, and *configuration*. The internal plugin representation is then parsed into the simulation managers, the set of private and public services used to build and run simulations. Finally, the formal representation of the simulation lifecycle is constructed and the initialization phase begins.

3.9.2 The Initialization Phase

The initialization phase of a `vivarium` simulation starts when the *`LifeCycle`* is fully constructed and ends when the `__init__` method of the *`vivarium.framework.engine.SimulationContext`* completes.

Two important things happen here:

- The internal representation of the simulation *components* is parsed into python import paths and **all** components are instantiated and registered with the component manager.
- The internal representation of the *configuration* is updated with all component configuration defaults.

At this point, all input arguments have been parsed, all components have been instantiated and registered with the framework, and the configuration is effectively complete. In an interactive setting, this is a useful phase in the simulation life cycle because you can add locally created components and modify the configuration.

3.9.3 The Setup Phase

The setup phase is broken down into three life cycle states.

Setup

The first state is named the same as the phase and is where the bulk of the phases work is done. During the setup state, the simulation managers and then the simulation components will have their `setup` method called with the simulation *builder* as an argument. The builder allows the components to request services like *randomness* or views into the *population state table* or to register themselves with various simulation subsystems. Setting up components may also involve loading data, registering or getting *pipelines*, creating *lookup tables*, and registering *population initializers*, among other things. The specifics of this are determined by the `setup` method on each component - the framework itself simply calls that method with a *`vivarium.framework.engine.Builder`* object.

Post-setup

This is a short state that exists in the simulation mainly so that framework *managers* can coordinate shared state and do any necessary cleanup. This is the first actual *event* emitted by the simulation framework. Normal `vivarium` *components* should never listen for this event. This may be enforced at a later date.

Population Initialization

It's not until this stage that the framework actually generates the base *population* for the simulation. Here, the framework rewinds the simulation *clock* one time step and generates the population. This time step fence-posting ensures that *simulants* enter the simulation on the correct start date. Note that this rewinding of the clock is purely what it sounds like - there is no concept of a time step being taken here. Instead, the clock is literally reset back the duration

of one time step. Once the simulant population is generated, the clock is reset to the simulation start time, again by changing the clock time only without any time step being taken.

3.9.4 The Main Event Loop

At this stage, all the preparation work has been completed and the framework begins to move through the simulation. This occurs as an *event loop*. Like the the setup phase, the main loop phase is broken into a series of simulation states. The framework signals the state transitions by emitting a series of events for each *time step*:

1. *time_step__prepare* A state in which simulation *components* can do any work necessary to prepare for the time step.
2. *time_step* The phase in which the bulk of the simulation work is done. Simulation state is updated.
3. *time_step__cleanup* A phase for simulation components to do any post time step cleanup.
4. *collect_metrics* A life-cycle phase specifically reserved for computing and recording simulation outputs.

By listening for these events, individual components can perform actions, including manipulating the *state table*. This sequence of events is repeated until the simulation clock passes the simulation end time.

Note: We have multiple sources of time during this process. The `vivarium.framework.engine.SimulationContext` itself holds onto a clock. This simulation clock is the actual time in the simulation. Events (including e.g., *time_step*) come with a time as well. This time is the time at the start of the next time step, that is, the time when any changes made during the loop will happen.

3.9.5 The Simulation End Phase

The final phase in the simulation life cycle is fittingly enough, simulation end. It is split into two states. During the first, the *simulation_end* *event* is emitted to signal that the event loop has finished and the *state table* is final. At this point, final simulation outputs are safe to compute. The second state is *report* in which the simulation will accumulate all final outputs and return them.

3.10 Lookup Tables

Simulations tend to require a large quantity of data to run. A completely reasonable way to look at a simulation is to think of it as a task of getting the right data and the right random numbers in the appropriate place at the appropriate time. To address the first concern, `vivarium` provides the `Lookup Table` abstraction to ensure that the right data can be retrieved when it's needed. In particular, it attempts to wrap different strategies for constructing interpolations or distributions on data such that a user simply needs to request values for a set of simulants when they're needed. This idea is extended to compositions of of several data-based values by `vivarium`'s *values system*.

- *Extrapolation*
- *Specifying Options in the Model Configuration*

3.10.1 The Lookup Table

A `Lookup Table` for a quantity is a callable object that is built from a scalar value or a `pandas.DataFrame` of data points that describes how the quantity varies with other variable(s). It is called with a `pandas.Index` as a function parameter which represents a simulated population as discussed in the *population concept*. When called, the *`Lookup Table`* simply returns appropriate values of the quantity for the population it was passed, interpolating if necessary or extrapolating if configured to. This behavior represents the standard interface for asking for data about a population in a simulation.

The lookup table system is built in layers. At the top is the *`Lookup Table`* object which is responsible for providing a uniform interface to the user regardless of the underlying implementation. From the user's perspective, it takes in a data set or scalar value on initialization and then lets them query against that data with a population index.

The next layer is selected at initialization time based on the type of data provided. The *`Lookup Table`* picks either a *`ScalarTable`* if a single value is provided as the data or a *`InterpolatedTable`* if a `pandas.DataFrame` is provided as the data.

Note: The *`InterpolatedTable`* is a misnomer here. It confuses the data handling strategy with the underlying data representation. A better name would be `BinnedDataTable` to indicate that it wraps data where the continuous parameters are represented by bin edges in the provided data. This would allow us to easily think about and extend the lookup system to wrap data where the continuous parameters are represented by points and to tables where all parameters are categorical.

If the underlying data is a single value, this is the last layer of abstraction. The *`ScalarTable`* has only one reasonable strategy which is to broadcast the value over the population index. If we have a `pandas.DataFrame` and therefore an *`InterpolatedTable`*, there are additional layers to the lookup system to allow the user to control the strategy for turning the population index into values based on the data. The *`InterpolatedTable`* is then responsible for turning the population index into a set of attributes relevant to the value production based on the structure of the input data and then providing those attributes to the value production strategy.

Note: I'm being careful with language here. We have objects named `Interpolation` and `InterpolatedTable` though the operation they perform is actually disaggregation. If we extend the system to work with point estimates for the continuous parameters, then interpolation would appropriately describe what we do. Both are value production strategies based on the structure of the input data.

More information about the value production strategies can be found in *here*.

Construction Parameters

A lookup table is defined for a set of categorical variables, continuous variables, and the values that depend on those variables. The lookup table calls these variables keys, parameters, and values, respectively.

key A categorical variable, such as sex, that a quantity depends on.

parameter A continuous variable, such as age, that a quantity depends on. This data frequently represents bins for which values are defined.

value Known values of the quantity of interest, which vary with the keys and parameters.

Along with data about these variables, A lookup table is instantiated with the corresponding column names which are used to query an internal *`population view`* when the table itself is called. This means the lookup table only needs to be called with a population index – it gathers the population information it needs itself. It also means the data must be available in the *population state table* with the same column name.

In the table below is an example of (unrealistic) data that could be used to create a lookup table for a quantity of interest about a population, in this case, Body Mass Index (BMI). We may find ourselves in a situation where we want to know the BMI of a simulant in order to make a treatment decision. If we construct a lookup table with these data, we can cleanly get the information we want and go on implementing our treatment. When called, the lookup table will return values of BMI for the simulants defined by the population index.

Key	Parameter		Value
sex	age_start	age_end	BMI
Male	0	20	20
Male	20	40	25
Male	40	60	30
Male	60	100	27
Female	0	20	20
Female	20	40	25
Female	40	60	30
Female	60	100	27

Example Usage

The following is an example of creating and calling a lookup table in an *interactive setting* using the data above. The interface and process are the same when integrating a lookup table into a *component*, which is primarily how they are used. Assuming you have a valid simulation object named `sim` and the data from the above table in a `pandas.DataFrame` named `data`, you can construct a lookup table in the following way, using the interface from the builder.

```
  # value_columns implicitly set to remaining columns
> bmi = sim.builder.lookup.build_table(data, key_columns=['sex'], parameter_columns=[
↪'age'])
> population = sim.get_population()
> bmi(population.index).head()  # returns BMI values for the population

  0     20.0
  1     20.0
  2     30.0
  3     27.0
  4     25.0
  Name: BMI, dtype: float64
```

Note: Constructing a lookup table currently requires your data meet specific conditions. These are a consequence of the method the lookup table uses to arrive at the correct data. Specifically, your parameter columns must represent bins and they must overlap.

3.10.2 Estimating Unknown Values

Interpolation

If a lookup table was constructed with a scalar value or values, the lookup call trivially returns the same scalar(s) back for any population passed in. However, if the lookup table was instead created with a `pandas.DataFrame` of varying data the lookup will perform interpolation which is an important feature. Interpolation is the process of estimating values for unspecified parameters within the bounds of the parameters we have defined in the lookup table. Currently, the most common case arises when the values are binned by the parameters. Then, the interpolation simply finds the correct bin a value belongs to. Please see the *interpolation concept note* for more in-depth information about the kinds of interpolation performed by the lookup table.

Extrapolation

Previously, we discussed interpolation as the process of estimating data within the bounds defined by our lookup table. What would happen if we wanted data outside of this range? Estimating such data is called extrapolation, and it can be performed using a lookup table as well. Extrapolation is a configurable option that, when enabled, allows a lookup data to provide values outside of the range it was created with. This is done by extending the edge points outwards to encompass outside points. This is a dumb but useful strategy and is primarily used to run simulations beyond the time bounds included in the data under the assumption that parameters do not change in the future.

Specifying Options in the Model Configuration

Configuring interpolation and extrapolation in a model specification is straightforward. Currently, the only acceptable value for order is *0*. Extrapolation can be turned on and off.

```
configuration:
    interpolation:
        order: 0
        extrapolate: True
```

3.11 Population Management

Since `Vivarium` is an agent-based simulation framework, managing a group of simulants and their attributes is a critical task. Fundamentally, to run a simulation we need to be able to create new simulates, update their state attributes, and facilitate access to their state so that components in the simulation can do interesting things based on it. The tooling to support working with our simulant population is called the population management system.

3.11.1 The State Table

The core representation of simulants and their state information in `Vivarium` is a `pandas.DataFrame` known as the state table. Under this representation rows represent simulants while columns correspond to state attributes like age, sex or systolic blood pressure. These columns represent one of several important resources within `Vivarium` that other components can draw on. Each of the actions we need to be able to take correspond to a manipulation of this

state table. The addition of new simulants is the creation of rows, the creation of new state attributes is the creation of columns, and the reading and updating of state is reading and updating the dataframe itself.

<<TODO: image of state table w/ expanding rows and columns>>

3.11.2 Population Views

The population manager holds the state table directly and tightly controls read and write access to it through a structure it provides known as a population view. A population view itself represents a subset of columns and rows from the state table. Through a view, components can read, update, or, under the right circumstances, create new state in the state table.

Views are created on-demand for components in a simulation by specifying a set of columns and an optional query string to the population manager interface. The columns dictate the subset of the state table that is viewable and modifiable while the query string is a filter on the simulants returned. The view itself is callable and accepts an index, which is the simulants to be viewed. It also provides an update method that accepts a dataframe and will replace values in the state table according to column and index. Only the columns that the view was created with can be updated in this way. The only exception is at simulant initialization time, when initial state must be created.

Population views can themselves create subviews through the subview method. This generates a new population view that is constrained by it's parents columns and query string in addition to whatever arguments it is passed, with the requirement that it's columns must be a subset of its parent view's.

<<TODO: Potentially a view to subview picture showing additional constraining>>

3.11.3 Creating Simulants

The population view pattern also underlies the creation of simulants, the only difference being that when simulations are being initialized for the first time, it is acceptable to create columns in the state table via update that don't already exist.

The Simulant Creator Function

Simulants are are introduced to the simulation using a function that takes the number of new simulants as its parameter. This function, known as the simulant creator, is provided by the population manager interface and is used by the simulation entrypoint to initialize the population. It can also be used by components that want to introduce new simulants over the course of a simulation, such as a fertility component that models births. This means there are two distinct execution states in which simulants can be created: The population initialization state during the setup phase, and the main event loop.

The simulant creator function first adds rows to the state table. It then loops through a set of functions that have been registered to it as population intitializers via *initializes_simulants*, passing in the index of the newly created simulants. These functions generally proceed by using population views to dictate the state of the newly created simulants they are responsible for. It is the only time creating columns in the state table is acceptable.

3.12 Resource Management

Todo: Everything here.

3.13 Thinking about Time in the Simulation

3.13.1 Outline

- The simulation clock
- Start, stop, and step size.
- Fundamental assumptions about discrete time simulation.
- builder.time.clock().clock() vs. event.time

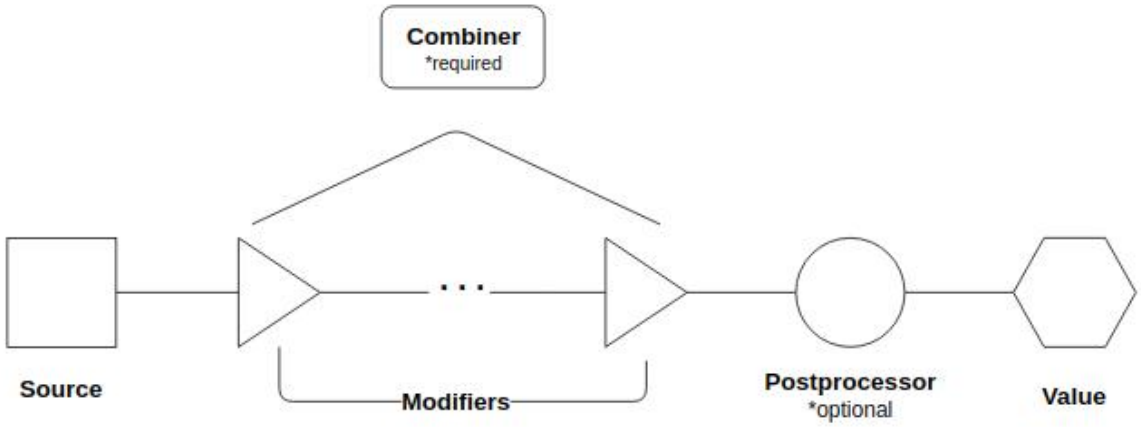
3.14 The Values System

The values system provides an interface to an alternative representation of *state* in the simulation: pipelines. `Pipelines` are dynamically calculated values that can be constructed across multiple *components*. This ability for multiple components to together compose a single value is the biggest advantage pipelines provide over the standard state representation of the population state table.

Note: You should use the values system when you have a value that must be composed across multiple components.

3.14.1 What are pipelines?

We can visualize a pipeline as the following:



At the left, we have the original **source** of the pipeline. This is a callable registered by a single component that returns a dataframe. To this source, additional components can register **modifiers**. These modifiers are also callables that return dataframes.

The source and modifiers are composed into a single value by the **combiner** with which the pipeline is registered. The combiner is also a callable that returns a dataframe - it is the function that dictates how the dataframe produced by the source and the dataframes produced by the modifiers will be combined into a single dataframe. The combiner also determines the required signatures of modifiers in relation to the source. The values system provides three options for combiners, detailed in the following table.

Table 4: **Pipeline Combiners**

Combiner	Description	Modifier Signature
`Replace`	Replaces the output of the source or modifier with the output of the next modifier. This is the default combiner if none is specified on pipeline registration.	Arguments for the modifiers should be the same as the source with an additional last argument of the results of the previous modifier.
`List`	The output of the source should be a list to which the results of the modifiers are appended.	Modifiers should have the same signature as the source.

Pipelines may also optionally be registered with a **postprocessor**. This is a callable that returns a dataframe that will be called on the output of the combiner to do some postprocessing.

Table 5: **Pipeline Post-processors**

Post-processor	Description
`Rescale`	Used for pipelines that produce rates. Rescales the rates to the size of the time step. Rates provided by source and modifiers are presumed to be annual.
`Union`	Used for pipelines that produce independent proportions or probabilities. Combines values in a way that is consistent with a union of the underlying sample space

The values system also inverts the direction of control from information that is stored in the state table. Components that update columns in the state table can be seen as "pushing" that information out. Pipelines, however, are "pulled" on by components, often components that did not play any part in the construction of the pipeline value.

3.14.2 How to use pipelines

The values system provides four interface methods, available off the *builder* during setup.

Table 6: **Values System Interface Methods**

Method	Description
register_value_producer	Register a new pipeline with the values system. Provide a name for the pipeline and a source. Optionally provide a combiner (defaults to the replace combiner) and a postprocessor. Provide dependencies (see note).
register_rate_producer	A special case of `register_value_producer()` for rates specifically. Provide a name for the pipeline and a source and the values system will automatically use the rescale postprocessor. Provide dependencies (see note).
register_value_modifier	Register a modifier to a pipeline. Provide a name for the pipeline to modify and a modifier callable. Provide dependencies (see note).
get_value	Retrieve a reference to the pipeline with the given name.

Note: The registration methods for the values system require dependencies be specified in order for the *resource manager* to properly order and manage dependencies. These dependencies are the state table columns, other pipelines,

and randomness streams that the source or modifier callable uses in producing the dataframe it returns.

For a view of the values system in action, see the *disease model tutorial*, specifically the mortality component.

3.15 The Model Specification

A *model specification* is a complete representation of a `vivarium` simulation formatted as a yaml file.

A model specification file contains three distinct blocks:

1. Plugins
2. Components
3. Configuration

Each of these blocks is delineated by a top-level key in the yaml file: `plugins`, `components`, or `configuration`, respectively.

You can find a short intro to yaml basics *here*.

3.15.1 The Components Block

The components block of the model specification file contains the information necessary to identify the components that should be included in the model. Each *component* in this block maps to an object that will be managed by the simulation to add some functionality.

In the model specification, these components should be specified in either a list or a hierarchical format, as the following examples illustrate:

A flat list:

```
components:
    vivarium_examples.disease_model.population.BasePopulation()
    vivarium_examples.disease_model.population.Mortality()
    vivarium_examples.disease_model.disease.SIS_DiseaseModel("diarrhea")
```

and a hierchical format:

```
components:
    vivarium_examples:
        disease_model:
            population:
                - BasePopulation()
                - Mortality()
            disease:
                - SIS_DiseaseModel("diarrhea")
```

When the model specification is loaded in, we need some way of transforming the string representation of the components into the actual component objects that the simulation can use. The exact process of that mapping between the model specification item and the fully instantiated object is the domain of the *ComponentConfigurationParser*.

The *ComponentConfigurationParser* is responsible for taking a list or hierarchical ConfigTree of components derived from a model specification file and turning it into a list of instantiated component objects. The *get_components* method of the parser is used anytime a simulation is initialized from a model specification file. This method is responsible for the following three steps that comprise the transformation process:

1. Parsing the model specification's components
2. Validating the arguments and prepping each component
3. Importing and instantiating the actual components

To illustrate this process, the result of a *get_components* call on either of the above yaml components block examples would be a list containing three instantiated objects: a population object, a mortality object, and a diarrhea disease model.

YAML Basics

YAML is a simple, human-readable data serialization format that is used for vivarium *model specification* files. The extensions of a file can be **.yaml** or **.yml**, both of which are accepted throughout the vivarium framework. The following are general rules to keep in mind when writing and interpreting YAML files. Examples use snippets from vivarium model specifications but do not go in-depth about that topic. For more information about model specifications, please see the relevant *concept note*.

You can find way more information than you wanted about YAML on their website.

Structure

YAML files are structured by lines and space indentations. Indentation levels should be either 2 or 4 spaces, and **tabs are not valid**. For example, a specification file that sets parameters for a BMI drug treatment component looks like the following:

```
configuration:
    bmi_treatment:
        age_cutoff: 20
        bmi_cutoff: 30
        adherence_proportion: 0.92
        treatment_proportion: 1.0
        treatment_available:
            year: 2019
            month: 7
            day: 15
```

Comments

YAML comments are denoted with the pound symbol #, and can be placed anywhere, but must be separated from the preceding token by a space. For example, adding a comment to the configuration from above looks like this:

```
configuration:
    bmi_treatment:
        age_cutoff: 20
        bmi_cutoff: 30
        adherence_proportion: 0.92  #  Proportion of population selected who continue␣
↪treatment
        treatment_proportion: 1.0  # Proportion of population selected to be treated
        treatment_available:
            year: 2019
            month: 7
            day: 15
```

Mappings

A mapping, or key-value pairing, is formed using a colon :. This corresponds to an entry from the `dictionary` data structure from Python, and there is no notion of ordering. Mappings can be specified in block format or inline, however we recommend block format so that is what we will show an example of here. In block format, mappings are separated onto new lines, and indentation forms a parent-child relationship. For example, below is a snippet from a configuration that specifies configuration parameters for a simulation population as mappings. Each colon below begins a mapping.

```
configuration:
    population:
        population_size: 1000
        age_start: 0
        age_end: 30
```

The interpretation of this configuration into Python is shown below . You may have noticed that the above example contains nested mappings, this is valid YAML syntax and it relies on whitespace indentation. Also, the inner most block (population_size, age_start, age_end) is unordered.

```
{configuration: {
    population: {
        population_size: 1000,
        age_start: 0,
        age_end: 30
        }
    }
}
```

Lists

An in-line list in YAML is formed by a comma-separated set of items inside square brackets, similar to a python list. For example, below is a YAML configuration snippet that defines a list of years in which a hypothetical drug treatment is available in a simulation.

```
configuration:
    drug_treatment:
        available_years: [2015, 2016, 2017]
```

This will be interpreted in python as

```
{configuration:
    drug_treatment: {
        available_years: [2015, 2016, 2017]
    }
}
```

You may sometimes see a list in block format, which is also valid YAML syntax. Such a list is formed using a hyphen - and with each entry appearing on a new line with the same indentation level. The YAML example below is interpreted equivalently in python to the previous YAML example.

```
configuration:
    drug_treatment:
        available_years:
            - 2015
            - 2016
            - 2017
```

Composite Data

Lists and Mappings can be nested together to make more complicated structures. In fact, the previous mapping and list examples were taken from Vivarium model specifications and included nested mappings and lists. Vivarium model specifications will generally always take the form of these nested mappings, where some values are lists.

CHAPTER 4

API Reference

4.1 The Config Tree

A configuration structure which supports cascading layers.

In `vivarium` it allows base configurations to be overridden by component level configurations which are in turn overridden by model level configuration which can be overridden by user supplied overrides. From the perspective of normal client code the cascading is hidden and configuration values are presented as attributes of the configuration object the values of which are the value of that key in the outermost layer of configuration where it appears.

For example:

```
>>> config = ConfigTree(layers=['inner_layer', 'middle_layer', 'outer_layer', 'user_
↪overrides'])
>>> config.update({'section_a': {'item1': 'value1', 'item2': 'value2'}, 'section_b': {
↪'item1': 'value3'}}, layer='inner_layer')
>>> config.update({'section_a': {'item1': 'value4'}, 'section_b': {'item1': 'value5'}}
↪, layer='middle_layer')
>>> config.update({'section_b': {'item1': 'value6'}}, layer='outer_layer')
>>> config.section_a.item1
'value4'
>>> config.section_a.item2
'value2'
>>> config.section_b.item1
'value6'
```

exception vivarium.config_tree.**ConfigurationError**(*message*, *value_name*)
Base class for configuration errors.

exception vivarium.config_tree.**ConfigurationKeyError**(*message*, *value_name*)
Error raised when a configuration lookup fails.

exception vivarium.config_tree.**DuplicatedConfigurationError**(*message*, *name*, *layer*, *source*, *value*)
Error raised when a configuration value is set more than once.

layer
The configuration layer at which the value is being set.

source
The original source of the configuration value.

value
The original configuration value.

class vivarium.config_tree.**ConfigNode**(*layers*, *name*)
A priority based configuration value.

A *ConfigNode* represents a single configuration value with priority-based layers. The intent is to allow a value to be set from sources with different priorities and to record what the value was set to and from where.

For example, a simulation may need certain values to always exist, and so it will set them up at a "base" layer. Components in the simulation may have a different set of priorities and so override the "base" value at a "component" level. Finally a user may want to override the simulation and component defaults with values at the command line or interactively, and so those values will be set in a final "user" layer.

A *ConfigNode* may only have a value set at each layer once. Attempts to set a value at the same layer multiple times will result in a *DuplicatedConfigurationError*.

The *ConfigNode* will record all values set and the source they are set from. This sort of provenance with configuration data greatly eases debugging and analysis of simulation code.

This class should not be instantiated directly. All interaction should take place by manipulating a *ConfigTree* object.

name
The name of this configuration value.

Return type str

accessed
Returns whether this node has been accessed.

Return type bool

metadata
Returns all values and associated metadata for this node.

Return type List[Dict[str, Any]]

freeze()
Causes the *ConfigNode* node to become read only.

This can be used to create a contract around when the configuration is modifiable.

get_value(*layer*)
Returns the value at the specified layer.

If no layer is specified, the outermost (highest priority) layer at which a value has been set will be used.

Parameters **layer** (Optional[str]) – Name of the layer to retrieve the value from.

Raises KeyError – If no value has been set at any layer.

Return type Any

update(*value*, *layer*, *source*)
Set a value for a layer with optional metadata about source.

Parameters

- **value** (Any) – Data to store in the node.

- **layer** (Optional[str]) – Name of the layer to use. If no layer is provided, the value will be set in the outermost (highest priority) layer.
- **source** (Optional[str]) – Metadata indicating the source of this value.

Raises

- *ConfigurationError* – If the node is frozen.
- *ConfigurationKeyError* – If the provided layer does not exist.
- *DuplicatedConfigurationError* – If a value has already been set at the provided layer or a value is already in the outermost layer and no layer has been provided.

class vivarium.config_tree.**ConfigTree**(*data=None*, *layers=None*, *name=''*)

A container for configuration information.

Each configuration value is exposed as an attribute the value of which is determined by the outermost layer which has the key defined.

freeze()

Causes the ConfigTree to become read only.

This is useful for loading and then freezing configurations that should not be modified at runtime.

items()

Return an iterable of all (child_name, child) pairs.

Return type Iterable[Tuple[str, Union[*ConfigTree*, *ConfigNode*]]]

keys()

Return an Iterable of all child names.

Return type Iterable[str]

values()

Return an Iterable of all children.

Return type Iterable[+T_co]

unused_keys()

Lists all values in the ConfigTree that haven't been accessed.

Return type List[str]

to_dict()

Converts the ConfigTree into a nested dictionary.

All metadata is lost in this conversion.

Return type Dict[~KT, ~VT]

get_from_layer(*name*, *layer=None*)

Get a configuration value from the provided layer.

If no layer is specified, the outermost (highest priority) layer at which a value has been set will be used.

Parameters

- **name** (str) – The name of the value to retrieve
- **layer** (Optional[str]) – The name of the layer to retrieve the value from.

Return type Any

update(*data*, *layer=None*, *source=None*)

Adds additional data into the *ConfigTree*.

Parameters

- **data** (Union[Dict[~KT, ~VT], str, Path, *ConfigTree*, None]) – *update()* accepts many types of data.
  - dict : Flat or nested dictionaries may be provided. Keys of dictionaries at all levels must be strings.
  - *ConfigTree* : Another *ConfigTree* can be used. All source information will be ignored and the provided layer and source will be used to set the metadata.
  - str : Strings provided can be yaml formatted strings, which will be parsed into a dictionary using standard yaml parsing. Alternatively, a path to a yaml file may be provided and the file will be read in and parsed.
  - pathlib.Path : A path object to a yaml file will be interpreted the same as a string representation.
- **layer** (Optional[str]) – The name of the layer to store the value in. If no layer is provided, the value will be set in the outermost (highest priority) layer.
- **source** (Optional[str]) – The source to attribute the value to.

Raises

- *ConfigurationError* – If the *ConfigTree* is frozen or attempting to assign an invalid value.
- *ConfigurationKeyError* – If the provided layer does not exist.
- *DuplicatedConfigurationError* – If a value has already been set at the provided layer or a value is already in the outermost layer and no layer has been provided.

metadata(*name*)

Return type List[Dict[str, Any]]

4.2 Exceptions

Module containing framework-wide exception definitions. Exceptions for particular subsystems are defined in their respective modules.

exception vivarium.exceptions.**VivariumError**

Generic exception raised for errors in vivarium simulations.

4.3 Interpolation

Provides interpolation algorithms across tabular data for vivarium simulations.

class vivarium.interpolation.**Interpolation**(*data*, *categorical_parameters*, *continuous_parameters*, *order*, *extrapolate*)

A callable that returns the result of an interpolation function over input data.

data

The data from which to build the interpolation. Contains cateogrical_parameters and continuous_parameters.

categorical_parameters
Column names to be used as categorical parameters in Interpolation to select between interpolation functions.

continuous_parameters
Column names to be used as continuous parameters in Interpolation. If bin edges, should be of the form (column name used in call, column name for left bin edge, column name for right bin edge).

order
Order of interpolation.

vivarium.interpolation.**validate_parameters**(*data*, *categorical_parameters*, *continuous_parameters*)

vivarium.interpolation.**validate_call_data**(*data*, *key_columns*, *parameter_columns*)

vivarium.interpolation.**check_data_complete**(*data*, *parameter_columns*)
For any parameters specified with edges, make sure edges don't overlap and don't have any gaps. Assumes that edges are specified with ends and starts overlapping (but one exclusive and the other inclusive) so can check that end of previous == start of current.

If multiple parameters, make sure all combinations of parameters are present in data.

Requires that bins of each parameter be standard across all values of other parameters, i.e., all bins for one parameter when de-duplicated should cover a continuous range of that parameter with no overlaps or gaps and the range covered should be the same for all combinations of other parameter values.

class vivarium.interpolation.**Order0Interp**(*data*, *parameter_columns*, *value_columns*, *extrapolate*)
A callable that returns the result of order 0 interpolation over input data.

data
The data from which to build the interpolation. Contains categorical_parameters and continuous_parameters.

parameter_columns
Column names to be used as parameters in Interpolation.

4.4 Vivarium Testing Utilities

Utility functions and classes to make testing `vivarium` components easier.

class vivarium.testing_utilities.**NonCRNTestPopulation**

configuration_defaults = {'population': {'age_end': 100, 'age_start': 0, 'exit_age'

name

setup(*builder*)

generate_test_population(*pop_data*)

age_simulants(*event*)

class vivarium.testing_utilities.**TestPopulation**

name

setup(*builder*)

generate_test_population(*pop_data*)

vivarium.testing_utilities.**build_table**(*value*, *year_start*, *year_end*, *columns=('age', 'year', 'sex', 'value')*)

vivarium.testing_utilities.**make_dummy_column**(*name*, *initial_value*)

vivarium.testing_utilities.**get_randomness**(*key='test'*, *clock=<function <lambda>>*, *seed=12345*, *for_initialization=False*)

vivarium.testing_utilities.**reset_mocks**(*mocks*)

vivarium.testing_utilities.**metadata**(*file_path*)

4.5 The Vivarium Framework

4.5.1 Configuration Utilities

A set of functions for turning model specification files into programmatic representations of *model specifications* and *configurations*.

vivarium.framework.configuration.**build_model_specification**(*model_specification=None*, *component_configuration=None*, *configuration=None*, *plugin_configuration=None*)

Return type *ConfigTree*

vivarium.framework.configuration.**validate_model_specification_file**(*file_path*)
Ensures the provided file is a yaml file

Return type None

vivarium.framework.configuration.**build_simulation_configuration**()

Return type *ConfigTree*

4.5.2 The Vivarium Engine

The engine houses the *SimulationContext* – the key `vivarium` object for running and interacting with simulations. It is the top level manager for all state information in `vivarium`. By intention, it exposes a very simple interface for managing the *simulation lifecycle*.

Also included here is the simulation *Builder*, which is the main interface that components use to interact with the simulation framework. You can read more about how the builder works and what services is exposes *here*.

Finally, there are a handful of wrapper methods that allow a user or user tools to easily setup and run a simulation.

class vivarium.framework.engine.**SimulationContext**(*model_specification=None*, *components=None*, *configuration=None*, *plugin_configuration=None*)

name

setup()

initialize_simulants()

step()

run()

finalize()

report()

add_components(*component_list*)
Adds new components to the simulation.

get_population(*untracked=True*)

class vivarium.framework.engine.**Builder**(*configuration*, *plugin_manager*)
Toolbox for constructing and configuring simulation components.

vivarium.framework.engine.**run_simulation**(*model_specification=None*, *components=None*, *configuration=None*, *plugin_configuration=None*)

4.5.3 The Vivarium Event Framework

vivarium constructs and manages the flow of *time* through the emission of regularly scheduled events. The tools in this module manage the relationships between event emitters and listeners and provide an interface for user *components* to register themselves as emitters or listeners to particular events.

The *EventManager* maintains a mapping between event types and channels. Each event type (and event types must be unique so event type is equivalent to event name, e.g., time_step_prepare) corresponds to an _EventChannel, which tracks listeners to that event in prioritized levels and passes on the event to those listeners when emitted.

The *EventInterface* is exposed off the *builder* and provides two methods: *get_emitter*, which returns a callable emitter for the given event type and *register_listener*, which adds the given listener to the event channel for the given event. This is the only part of the event framework with which client code should interact.

For more information, see the associated event *concept note*.

class vivarium.framework.event.**Event**
An Event object represents the context of an event.

Events themselves are just a bundle of data. They must be emitted along an _EventChannel in order for other simulation components to respond to them.

index
An index into the population table containing all simulants affected by this event.

user_data
Any additional data provided by the user about the event.

time
The simulation time at which this event will resolve. The current simulation size plus the current time step size.

step_size
The current step size at the time of the event.

index
Alias for field number 0

user_data
Alias for field number 1

time
Alias for field number 2

step_size
Alias for field number 3

split(*new_index*)
Create a copy of this event with a new index.

This function should be used to emit an event in a new _EventChannel in response to an event emitted from a different channel.

Parameters **new_index** (Index) – An index into the population table containing all simulants affected by this event.

Returns

Return type The new event.

class vivarium.framework.event.**EventManager**
The configuration for the event system.

Notes

Client code should never need to interact with this class except through the decorators in this module and the emitter function exposed on the builder during the setup phase.

name
The name of this component.

get_channel(*name*)

setup(*builder*)
Performs this component's simulation setup.

Parameters **builder** – Object giving access to core framework functionality.

on_post_setup(*event*)

get_emitter(*name*)
Get an emitter function for the named event.

Parameters **name** (str) – The name of the event.

Return type Callable[[Index, Optional[Dict[~KT, ~VT]]], *Event*]

Returns

- *A function that accepts an index and optional user data. This function*
- *creates and timestamps an Event and distributes it to all interested*
- *listeners*

register_listener(*name*, *listener*, *priority=5*)
Registers a new listener to the named event.

Parameters

- **name** (str) – The name of the event.
- **listener** (Callable) – The consumer of the named event.
- **priority** (int) – Number in range(10) used to assign the ordering in which listeners process the event.

get_listeners(*name*)
Get all listeners registered for the named event.

Parameters **name** (str) – The name of the event.

Return type Dict[int, List[Callable]]

Returns

- *A dictionary that maps each priority level of the named event's*
- *listeners to a list of listeners at that level.*

list_events()
List all event names known to the event system.

Returns

Return type A list of all known event names.

Notes

This value can change after setup if components dynamically create new event labels.

class vivarium.framework.event.**EventInterface**(*manager*)
The public interface for the event system.

get_emitter(*name*)
Gets an emitter for a named event.

Parameters **name** (str) – The name of the event the requested emitter will emit. Users may provide their own named events by requesting an emitter with this function, but should do so with caution as it makes time much more difficult to think about.

Return type Callable[[Index, Optional[Dict[~KT, ~VT]]], *Event*]

Returns

- *An emitter for the named event. The emitter should be called by*
- *the requesting component at the appropriate point in the simulation*
- *lifecycle.*

register_listener(*name*, *listener*, *priority=5*)
Registers a callable as a listener to a events with the given name.

The listening callable will be called with a named Event as its only argument any time the event emitter is invoked from somewhere in the simulation.

The framework creates the following events and emits them at different points in the simulation:

- At the end of the setup phase: post_setup
- Every time step: - time_step__prepare - time_step - time_step__cleanup - collect_metrics
- At simulation end: simulation_end

Parameters

- **name** (str) – The name of the event to listen for.
- **listener** (Callable[[*Event*], None]) – The callable to be invoked any time an Event with the given name is emitted.

- **priority** (*{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}*) – An indication of the order in which event listeners should be called. Listeners with smaller priority values will be called earlier. Listeners with the same priority have no guaranteed ordering. This feature should be avoided if possible. Components should strive to obey the Markov property as they transform the state table (the state of the simulation at the beginning of the next time step should only depend on the current state of the system).

Return type None

4.5.4 Life Cycle Management

The life cycle is a representation of the flow of execution states in a vivarium simulation. The tools in this model allow a simulation to formally represent its execution state and use the formal representation to enforce run-time contracts.

There are two flavors of contracts that this system enforces:

- **Constraints**: These are contracts around when certain methods, particularly those available off the *Builder*, can be used. For example, *simulants* should only be added to the simulation during initial population creation and during the main simulation loop, otherwise services necessary for initializing that population's attributes may not exist. By applying a constraint, we can provide very clear errors about what went wrong, rather than a deep and unintelligible stack trace.
- **Ordering Contracts**: The *SimulationContext* will construct the formal representation of the life cycle during its initialization. Once generated, the context declares as it transitions between different lifecycle states and the tools here ensure that only valid transitions occur. These kinds of contracts are particularly useful during interactive usage, as they prevent users from, for example, running a simulation whose population has not been created.

The tools here also allow for introspection of the simulation life cycle.

exception vivarium.framework.lifecycle.**LifeCycleError**
Generic error class for the life cycle management system.

exception vivarium.framework.lifecycle.**InvalidTransitionError**
Error raised when life cycle ordering contracts are violated.

exception vivarium.framework.lifecycle.**ConstraintError**
Error raised when life cycle constraint contracts are violated.

class vivarium.framework.lifecycle.**LifeCycleState**(*name*)
A representation of a simulation run state.

name
The name of the lifecycle state.

Return type str

entrance_count
The number of times this state has been entered.

Return type int

add_next(*next_state*, *loop=False*)
Link this state to the next state in the simulation life cycle.

States are linked together and used to ensure that the simulation life cycle proceeds in the proper order. A life cycle state can be bound to two next states to allow for loops in the life cycle and both are considered valid when checking for valid state transitions. The first represents the linear progression through the simulation, while the second represents a loop in the life cycle.

Parameters

- **next_state** (*LifeCycleState*) – The next state in the simulation life cycle.
- **loop** (bool) – Whether the provided state is the linear next state or a loop back to a previous state in the life cycle.

valid_next_state(*state*)
Check if the provided state is valid for a life cycle transition.

Parameters **state** (Optional[*LifeCycleState*]) – The state to check.

Returns

Return type Whether the state is valid for a transition.

enter()
Marks an entrance into this state.

add_handlers(*handlers*)
Registers a set of functions that will be executed during the state.

The primary use case here is for introspection and reporting. For setting constraints, see *LifeCycleInterface.add_constraint()*.

Parameters **handlers** (List[Callable]) – The set of functions that will be executed during this state.

class vivarium.framework.lifecycle.**LifeCyclePhase**(*name*, *states*, *loop*)
A representation of a distinct lifecycle phase in the simulation.

A lifecycle phase is composed of one or more unique lifecycle states. There is exactly one state within the phase which serves as a valid exit point from the phase. The states may operate in a loop.

name
The name of this life cycle phase.

Return type str

states
The states in this life cycle phase in order of execution.

Return type Tuple[*LifeCycleState*]

add_next(*phase*)
Link the provided phase as the next phase in the life cycle.

get_state(*state_name*)
Retrieve a life cycle state by name from the phase.

Return type *LifeCycleState*

class vivarium.framework.lifecycle.**LifeCycle**
A concrete representation of the flow of simulation execution states.

add_phase(*phase_name*, *states*, *loop*)
Add a new phase to the lifecycle.

Phases must be added in order.

Parameters

- **phase_name** (str) – The name of the phase to add. Phase names must be unique.
- **states** (List[str]) – The list of names (in order) of the states that make up the life cycle phase. State names must be unique across the entire life cycle.

- **loop** – Whether the life cycle phase states loop.

Raises *LifeCycleError* – If the phase or state names are non-unique.

get_state(*state_name*)
Retrieve a life cycle state from the life cycle.

Parameters **state_name** (str) – The name of the state to retrieve

Returns

Return type The requested state.

Raises *LifeCycleError* – If the requested state does not exist.

get_state_names(*phase_name*)
Retrieve the names of all states in the provided phase.

Parameters **phase_name** (str) – The name of the phase to retrieve the state names from.

Returns

Return type The state names in the provided phase.

Raises *LifeCycleError* – If the phase does not exist in the life cycle.

class vivarium.framework.lifecycle.**ConstraintMaker**(*lifecycle_manager*)
Factory for making state-based constraints on component methods.

check_valid_state(*method*, *permitted_states*)
Ensures a component method is being called during an allowed state.

Parameters

- **method** (method) – The method the constraint is applied to.
- **permitted_states** (List[str]) – The states in which the method is permitted to be called.

Raises *ConstraintError* – If the method is being called outside the permitted states.

constrain_normal_method(*method*, *permitted_states*)
Only permit a method to be called during the provided states.

Constraints are applied by dynamically wrapping and binding a method to an existing component at run time.

Parameters

- **method** (method) – The method to constrain.
- **permitted_states** (List[str]) – The life cycle states in which the method can be called.

Returns

Return type The constrained method.

static to_guid(*method*)
Convert a method on to a global id.

Because we dynamically rebind methods, the old ones will get garbage collected, making id() unreliable for checking if a method has been constrained before.

Return type str

class vivarium.framework.lifecycle.**LifeCycleManager**
Manages ordering- and constraint-based contracts in the simulation.

name
The name of this component.

Return type str

current_state
The name of the current life cycle state.

Return type str

add_phase(*phase_name*, *states*, *loop=False*)
Add a new phase to the lifecycle.

Phases must be added in order.

Parameters

- **phase_name** (str) – The name of the phase to add. Phase names must be unique.
- **states** (List[str]) – The list of names (in order) of the states that make up the life cycle phase. State names must be unique across the entire life cycle.
- **loop** (bool) – Whether the life cycle phase states loop.

Raises *LifeCycleError* – If the phase or state names are non-unique.

set_state(*state*)
Sets the current life cycle state to the provided state.

Parameters **state** (str) – The name of the state to set.

Raises

- *LifeCycleError* – If the requested state doesn't exist in the life cycle.
- *InvalidTransitionError* – If setting the provided state represents an invalid life cycle transition.

get_state_names(*phase*)
Gets all states in the phase in their order of execution.

Parameters **phase** (str) – The name of the phase to retrieve the states for.

Returns

Return type A list of state names in order of execution.

add_handlers(*state_name*, *handlers*)
Registers a set of functions to be called during a life cycle state.

This method does not apply any constraints, rather it is used to build up an execution order for introspection.

Parameters

- **state_name** (str) – The name of the state to register the handlers for.
- **handlers** (List[Callable]) – A list of functions that will execute during the state.

add_constraint(*method*, *allow_during=()*, *restrict_during=()*)
Constrains a function to be executable only during certain states.

Parameters

- **method** (method) – The method to add constraints to.
- **allow_during** (List[str]) – An optional list of life cycle states in which the provided method is allowed to be called.

- **restrict_during** (List[str]) – An optional list of life cycle states in which the provided method is restricted from being called.

Raises

- ValueError – If neither allow_during nor restrict_during are provided, or if both are provided.
- *LifeCycleError* – If states provided as arguments are not in the life cycle.
- *ConstraintError* – If a lifecycle constraint has already been applied to the provided method.

class vivarium.framework.lifecycle.**LifeCycleInterface**(*manager*)

Interface to the life cycle management system.

The life cycle management system allows components to constrain methods so that they're only available during certain simulation life cycle states.

add_handlers(*state*, *handlers*)

Registers a set of functions to be called during a life cycle state.

This method does not apply any constraints, rather it is used to build up an execution order for introspection.

Parameters

- **state** (str) – The name of the state to register the handlers for.
- **handlers** (List[Callable]) – A list of functions that will execute during the state.

add_constraint(*method*, *allow_during=()*, *restrict_during=()*)

Constrains a function to be executable only during certain states.

Parameters

- **method** (method) – The method to add constraints to.
- **allow_during** (List[str]) – An optional list of life cycle states in which the provided method is allowed to be called.
- **restrict_during** (List[str]) – An optional list of life cycle states in which the provided method is restricted from being called.

Raises

- ValueError – If neither allow_during nor restrict_during are provided, or if both are provided.
- *LifeCycleError* – If states provided as arguments are not in the life cycle.
- *ConstraintError* – If a life cycle constraint has already been applied to the provided method.

4.5.5 Lookup Tables

Simulations tend to require a large quantity of data to run. vivarium provides the *LookupTable* abstraction to ensure that accurate data can be retrieved when it's needed. It's a callable object that takes in a population index and returns data specific to the individuals represented by that index. See the *lookup concept note* for more.

class vivarium.framework.lookup.**InterpolatedTable**(*data*, *population_view*, *key_columns*, *parameter_columns*, *value_columns*, *interpolation_order*, *clock*, *extrapolate*)

A callable that interpolates data according to a given strategy.

data
The data from which to build the interpolation.

population_view
View of the population to be used when the table is called with an index.

key_columns
Column names to be used as categorical parameters in Interpolation to select between interpolation functions.

parameter_columns
Column names to be used as continuous parameters in Interpolation.

value_columns
Names of value columns to be interpolated over. All non parameter- and key- columns in data.

interpolation_order
Order of interpolation. Used to decide interpolation strategy.

clock
Callable for current time in simulation.

extrapolate
Whether or not to extrapolate beyond edges of given bins.

Notes

These should not be created directly. Use the *lookup* interface on the class:*Builder* during setup.

class vivarium.framework.lookup.**ScalarTable**(*values*, *value_columns*)

A callable that broadcasts a scalar or list of scalars over an index.

values
The scalar value(s) from which to build table columns.

value_columns
List of string names to be used to name the columns of the table built from values.

Notes

These should not be created directly. Use the *lookup* interface on the builder during setup.

class vivarium.framework.lookup.**LookupTable**(*table_number*, *data*, *population_view*, *key_columns*, *parameter_columns*, *value_columns*, *interpolation_order*, *clock*, *extrapolate*)

Wrapper for different strategies for looking up values for an index.

In vivarium simulations, the index is synonymous with the simulated population. The lookup system allows the user to provide different kinds of data and strategies for using that data. When the simulation is running, then, components can lookup parameter values based solely on the population index.

Notes

These should not be created directly. Use the *lookup* method on the builder during setup.

name
Tables are generically named after the order they were created.

vivarium.framework.lookup.**validate_parameters**(*data*, *key_columns*, *parameter_columns*, *value_columns*)
Makes sure the data format agrees with the provided column layout.

class vivarium.framework.lookup.**LookupTableManager**
Manages complex data in the simulation.

Notes

Client code should never access this class directly. Use `lookup` on the builder during setup to get references to LookupTable objects.

configuration_defaults = {'interpolation': {'extrapolate': True, 'order': 0}}

name

setup(*builder*)

build_table(*data*, *key_columns*, *parameter_columns*, *value_columns*)
Construct a lookup table from input data.

Return type *LookupTable*

class vivarium.framework.lookup.**LookupTableInterface**(*manager*)
The lookup table management system.

Simulations tend to require a large quantity of data to run. `vivarium` provides the *Lookup Table* abstraction to ensure that accurate data can be retrieved when it's needed.

For more information, see *here*.

build_table(*data*, *key_columns=None*, *parameter_columns=None*, *value_columns=None*)
Construct a LookupTable from input data.

If data is a `pandas.DataFrame`, an interpolation function of the order specified in the simulation *configuration* will be calculated for each permutation of the set of key_columns. The columns in parameter_columns will be used as parameters for the interpolation functions which will estimate all remaining columns in the table.

If data is a number, time, list, or tuple, a scalar table will be constructed with the values in data as the values in each column of the table, named according to value_columns.

Parameters

- **data** (Union[~ScalarValue, DataFrame, List[~ScalarValue], Tuple[~ScalarValue]]) – The source data which will be used to build the resulting *LookupTable*.
- **key_columns** (Union[List[str], Tuple[str], None]) – Columns used to select between interpolation functions. These should be the non-continuous variables in the data. For example 'sex' in data about a population.
- **parameter_columns** (Union[List[str], Tuple[str], None]) – The columns which contain the parameters to the interpolation functions. These should be the continuous variables. For example 'age' in data about a population.

- **value_columns** (Union[List[str], Tuple[str], None]) – The data columns that will be in the resulting LookupTable. Columns to be interpolated over if interpolation or the names of the columns in the scalar table.

Returns

Return type *LookupTable*

4.5.6 Output Metrics

Currently, `vivarium` uses the *values pipeline system* to produce the results, or output metrics, from a simulation. The metrics The component here is a normal `vivarium` component whose only purpose is to provide an empty `dict` as the source of the *"Metrics"* pipeline. It is included by default in all simulations.

class vivarium.framework.metrics.**Metrics**
This class declares a value pipeline that allows other components to store summary metrics.

name

setup(*builder*)

4.5.7 The Plugin Management System

Todo: This part will come in with the full description of the plugin system in the next PR. -J.C. 05/07/19

exception vivarium.framework.plugins.**PluginConfigurationError**
Error raised when plugin configuration is incorrectly specified.

class vivarium.framework.plugins.**PluginManager**(*plugin_configuration=None*)

get_plugin(*name*)

get_plugin_interface(*name*)

get_core_controllers()

get_core_interfaces()

get_optional_controllers()

get_optional_interfaces()

4.5.8 The Population Management System

This module provides tools for managing the *state table* in a `vivarium` simulation, which is the record of all simulants in a simulation and their state. It's main tasks are managing the creation of new simulants and providing the ability for components to view and update simulant state safely during runtime.

exception vivarium.framework.population.**PopulationError**
Error raised when the population is invalidly queried or updated.

class vivarium.framework.population.**PopulationView**(*manager*, *view_id*, *columns=()*, *query=None*)
A read/write manager for the simulation state table.

It can be used to both read and update the state of the population. A PopulationView can only read and write columns for which it is configured. Attempts to update non-existent columns are ignored except during simulant creation when new columns are allowed to be created.

Parameters

- **manager** (*PopulationManager*) – The population manager for the simulation.
- **columns** (Union[List[str], Tuple[str], None]) – The set of columns this view should have access too. If explicitly specified as None, this view will have access to the entire state table.
- **query** (Optional[str]) – A pandas-style filter that will be applied any time this view is read from.

Notes

By default, this view will filter out untracked simulants unless the tracked column is specified in the initialization arguments.

name

columns
The columns that the view can read and update.

If the view was created with None as the columns argument, then the view will have access to the full table by default. That case should be only be used in situations where the full state table is actually needed, like for some metrics collection applications.

Return type List[str]

query
A pandas style query to filter the rows of this view.

This query will be applied any time the view is read. This query may reference columns not in the view's columns.

Return type str

subview(*columns*)
Retrieves a new view with a subset of this view's columns.

Parameters **columns** (Union[List[str], Tuple[str]]) – The set of columns to provide access to in the subview. Must be a proper subset of this view's columns.

Returns

Return type A new view with access to the requested columns.

Raises *PopulationError* – If the requested columns are not a proper subset of this view's columns.

Notes

Subviews are useful during population initialization. The original view may contain both columns that a component needs to create and update as well as columns that the component needs to read. By requesting a subview, a component can read the sections it needs without running the risk of trying to access uncreated columns because the component itself has not created them.

get (*index*, *query=''*)
Select the rows represented by the given index from this view.

For the rows in `index` get the columns from the simulation's state table to which this view has access. The resulting rows may be further filtered by the view's query and only return a subset of the population represented by the index.

Parameters

- **index** (`Index`) – Index of the population to get.
- **query** (`str`) – Additional conditions used to filter the index. These conditions will be unioned with the default query of this view. The query provided may use columns that this view does not have access to.

Returns

Return type A table with the subset of the population requested.

Raises *PopulationError* – If this view has access to columns that have not yet been created and this method is called. If you see this error, you should request a subview with the columns you need read access to.

See also:

`subview <PopulationView.subview()`

update (*population_update*)
Updates the state table with the provided data.

Parameters **population_update** (`Union`[`DataFrame`, `Series`]) – The data which should be copied into the simulation's state. If the update is a `pandas.DataFrame`, it can contain a subset of the view's columns but no extra columns. If `pop` is a `pandas.Series` it must have a name that matches one of this view's columns unless the view only has one column in which case the Series will be assumed to refer to that regardless of its name.

Raises *PopulationError* – If the provided data name or columns does not match columns that this view manages or if the view is being updated with a data type inconsistent with the original population data.

class `vivarium.framework.population.`**SimulantData**
Data to help components initialize simulants.

Any time simulants are added to the simulation, each initializer is called with this structure containing information relevant to their initialization.

index
The index representing the new simulants being added to the simulation.

user_data
A dictionary of extra data passed in by the component creating the population.

creation_time
The time when the simulants enter the simulation.

creation_window
The span of time over which the simulants are created. Useful for, e.g., distributing ages over the window.

index
Alias for field number 0

user_data
Alias for field number 1

creation_time
Alias for field number 2

creation_window
Alias for field number 3

class vivarium.framework.population.**InitializerComponentSet**
Set of unique components with population initializers.

add(*initializer*, *columns_produced*)
Adds an initializer and columns to the set, enforcing uniqueness.

Parameters

- **initializer** (Callable) – The population initializer to add to the set.
- **columns_produced** (List[str]) – The columns the initializer produces.

Raises

- TypeError – If the initializer is not an object method.
- AttributeError – If the object bound to the method does not have a name attribute.
- *PopulationError* – If the component bound to the method already has an initializer registered or if the columns produced are duplicates of columns another initializer produces.

class vivarium.framework.population.**PopulationManager**
Manages the state of the simulated population.

configuration_defaults = {'population': {'population_size': 100}}

name
The name of this component.

setup(*builder*)
Registers the population manager with other vivarium systems.

on_initialize_simulants(*pop_data*)
Adds a `tracked` column to the state table for new simulants.

metrics(*index*, *metrics*)
Reports tracked and untracked population sizes at simulation end.

get_view(*columns*, *query=None*)
Get a time-varying view of the population state table.

The requested population view can be used to view the current state or to update the state with new values.

If the column 'tracked' is not specified in the `columns` argument, the query string 'tracked == True' will be added to the provided query argument. This allows components to ignore untracked simulants by default.

Parameters

- **columns** (Union[List[str], Tuple[str]]) – A subset of the state table columns that will be available in the returned view.
- **query** (Optional[str]) – A filter on the population state. This filters out particular simulants (rows in the state table) based on their current state. The query should be provided in a way that is understood by the `pandas.DataFrame.query()` method and may reference state table columns not requested in the `columns` argument.

Returns A filtered view of the requested columns of the population state table.

Return type *PopulationView*

register_simulant_initializer(*initializer*, *creates_columns=()*, *requires_columns=()*, *requires_values=()*, *requires_streams=()*)

Marks a source of initial state information for new simulants.

Parameters

- **initializer** (Callable) – A callable that adds or updates initial state information about new simulants.
- **creates_columns** (List[str]) – A list of the state table columns that the given initializer provides the initial state information for.
- **requires_columns** (List[str]) – A list of the state table columns that already need to be present and populated in the state table before the provided initializer is called.
- **requires_values** (List[str]) – A list of the value pipelines that need to be properly sourced before the provided initializer is called.
- **requires_streams** (List[str]) – A list of the randomness streams necessary to initialize the simulant attributes.

get_simulant_creator()

Gets a function that can generate new simulants.

Return type Callable

Returns

- *The simulant creator function. The creator function takes the*
- *number of simulants to be created as it's first argument and a dict*
- *population configuration that will be available to simulant*
- *initializers as it's second argument. It generates the new rows in*
- *the population state table and then calls each initializer*
- *registered with the population system with a data*
- *object containing the state table index of the new simulants, the*
- *configuration info passed to the creator, the current simulation*
- *time, and the size of the next time step.*

get_population(*untracked*)

Provides a copy of the full population state table.

Parameters **untracked** (bool) – Whether to include untracked simulants in the returned population.

Returns

Return type A copy of the population table.

class vivarium.framework.population.**PopulationInterface**(*manager*)

Provides access to the system for reading and updating the population.

The most important aspect of the simulation state is the `population table` or `state table`. It is a table with a row for every individual or cohort (referred to as a simulant) being simulated and a column for each of the attributes of the simulant being modeled. All access to the state table is mediated by *`population views`*, which may be requested from this system during setup time.

The population system itself manages a single attribute of simulants called `tracked`. This attribute allows global control of which simulants are available to read and update in the state table by default.

For example, in a simulation of childhood illness, we might not need information about individuals or cohorts once they reach five years of age, and so we can have them "age out" of the simulation at five years old by setting the `tracked` attribute to `False`.

get_view(*columns*, *query=None*)

Get a time-varying view of the population state table.

The requested population view can be used to view the current state or to update the state with new values.

If the column 'tracked' is not specified in the `columns` argument, the query string 'tracked == True' will be added to the provided query argument. This allows components to ignore untracked simulants by default.

Parameters

- **columns** (Union[List[str], Tuple[str]]) – A subset of the state table columns that will be available in the returned view.
- **query** (Optional[str]) – A filter on the population state. This filters out particular simulants (rows in the state table) based on their current state. The query should be provided in a way that is understood by the `pandas.DataFrame.query()` method and may reference state table columns not requested in the `columns` argument.

Returns A filtered view of the requested columns of the population state table.

Return type *PopulationView*

get_simulant_creator()

Gets a function that can generate new simulants.

Return type Callable[[int, Optional[Dict[str, Any]]], Index]

Returns

- *The simulant creator function. The creator function takes the*
- *number of simulants to be created as it's first argument and a dict*
- *population configuration that will be available to simulant*
- *initializers as it's second argument. It generates the new rows in*
- *the population state table and then calls each initializer*
- *registered with the population system with a data*
- *object containing the state table index of the new simulants, the*
- *configuration info passed to the creator, the current simulation*
- *time, and the size of the next time step.*

initializes_simulants(*initializer*, *creates_columns=()*, *requires_columns=()*, *requires_values=()*, *requires_streams=()*)

Marks a source of initial state information for new simulants.

Parameters

- **initializer** (Callable[[*SimulantData*], None]) – A callable that adds or updates initial state information about new simulants.
- **creates_columns** (List[str]) – A list of the state table columns that the given initializer provides the initial state information for.

- **requires_columns** (List[str]) – A list of the state table columns that already need to be present and populated in the state table before the provided initializer is called.
- **requires_values** (List[str]) – A list of the value pipelines that need to be properly sourced before the provided initializer is called.
- **requires_streams** (List[str]) – A list of the randomness streams necessary to initialize the simulant attributes.

4.5.9 Random Numbers in `vivarium`

This module contains classes and functions supporting common random numbers.

Vivarium has some peculiar needs around randomness. We need to be totally consistent between branches in a comparison. For example, if a simulant gets hit by a truck in the base case in must be hit by that same truck in the counter-factual at exactly the same moment unless the counter-factual explicitly deals with traffic accidents. That means that the system can't rely on standard global randomness sources because small changes to the number of bits consumed or the order in which randomness consuming operations occur will cause the system to diverge. The current approach is to generate hash-based seeds where the key is the simulation time, the simulant's id, the draw number and a unique id for the decision point which needs the randomness. These seeds are then used to generate *numpy.random.RandomState* objects that can be used to create pseudo-random numbers in a repeatable manner.

vivarium.framework.randomness.**RESIDUAL_CHOICE**
A probability placeholder to be used in an un-normalized array of weights to absorb leftover weight so that the array sums to unity. For example:

```
[0.2, 0.2, RESIDUAL_CHOICE] => [0.2, 0.2, 0.6]
```

Note: Currently this object is only used in the *choice* function of this module.

Type object

For mor information, see the Common Random Numbers *concept note*.

exception vivarium.framework.randomness.**RandomnessError**
Exception raised for inconsistencies in random number and choice generation.

class vivarium.framework.randomness.**IndexMap**(*map_size=1000000*)
A key-index mapping with a simple vectorized hash and vectorized lookups.

TEN_DIGIT_MODULUS = 10000000000

update(*new_keys*)
Adds the new keys to the mapping.

Parameters **new_keys** (Index) – The new index to hash.

hash_(*keys*, *salt=0*)
Hashes the given index into an integer index in the range [0, self.stride]

Parameters

- **keys** (Index) – The new index to hash.
- **salt** (int) – An integer used to perturb the hash in a deterministic way. Useful in dealing with collisions.

Returns A pandas series indexed by the given keys and whose values take on integers in the range [0, self.stride]. Duplicates may appear and should be dealt with by the calling code.

Return type pd.Series

convert_to_ten_digit_int(*column*)
Converts a column of datetimes, integers, or floats into a column of 10 digit integers.

Parameters **column** (Series) – A series of datetimes, integers, or floats.

Returns A series of ten digit integers based on the input data.

Return type pd.Series

Raises *RandomnessError :* – If the column contains data that is neither a datetime-like nor numeric.

static digit(*m*, *n*)
Returns the nth digit of each number in m.

Return type Union[int, Series]

static clip_to_seconds(*m*)
Clips UTC datetime in nanoseconds to seconds.

Return type Union[int, Series]

spread(*m*)
Spreads out integer values to give smaller values more weight.

Return type Union[int, Series]

shift(*m*)
Shifts floats so that the first 10 decimal digits are significant.

Return type Union[int, Series]

vivarium.framework.randomness.**random**(*key*, *index*, *index_map=None*)
Produces an indexed *pandas.Series* of uniformly distributed random numbers.

The index passed in typically corresponds to a subset of rows in a *pandas.DataFrame* for which a probabilistic draw needs to be made.

Parameters

- **key** (str) – A string used to create a seed for the random number generation.
- **index** (Index) – The index used for the returned series.
- **index_map** (Optional[*IndexMap*]) – A mapping between the provided index (which may contain ints, floats, datetimes or any arbitrary combination of them) and an integer index into the random number array.

Returns A series of random numbers indexed by the provided index.

Return type pd.Series

vivarium.framework.randomness.**get_hash**(*key*)
Gets a hash of the provided key.

Parameters **key** (str) – A string used to create a seed for the random number generator.

Returns A hash of the provided key.

Return type int

vivarium.framework.randomness.**choice**(*key*, *index*, *choices*, *p=None*, *index_map=None*)

Decides between a weighted or unweighted set of choices.

Given a a set of choices with or without corresponding weights, returns an indexed set of decisions from those choices. This is simply a vectorized way to make decisions with some book-keeping.

Parameters

- **key** (str) – A string used to create a seed for the random number generation.
- **index** (*pandas.Index*) – An index whose length is the number of random draws made and which indexes the returned *pandas.Series*.
- **choices** (Union[List[~T], Tuple, ndarray, Series]) – A set of options to choose from.
- **p** (Union[List[~T], Tuple, ndarray, Series, None]) – The relative weights of the choices. Can be either a 1-d array of the same length as *choices* or a 2-d array with *len(index)* rows and *len(choices)* columns. In the 1-d case, the same set of weights are used to decide among the choices for every item in the *index*. In the 2-d case, each row in *p* contains a separate set of weights for every item in the *index*.
- **index_map** (Optional[*IndexMap*]) – A mapping between the provided index (which may contain ints, floats, datetimes or any arbitrary combination of them) and an integer index into the random number array.

Returns An indexed set of decisions from among the available *choices*.

Return type pd.Series

Raises *RandomnessError* – If any row in *p* contains *RESIDUAL_CHOICE* and the remaining weights in the row are not normalized or any row of *p* contains more than one reference to *RESIDUAL_CHOICE*.

vivarium.framework.randomness.**filter_for_probability**(*key*, *population*, *probability*, *index_map=None*)

Decide an event outcome for each individual in a population from probabilities.

Given a population or its index and an array of associated probabilities for some event to happen, we create and return the sub-population for whom the event occurred.

Parameters

- **key** (str) – A string used to create a seed for the random number generation.
- **population** (Union[DataFrame, Series, Index]) – A view on the simulants for which we are determining the outcome of an event.
- **probability** (Union[List[~T], Tuple, ndarray, Series]) – A 1d list of probabilities of the event under consideration occurring which corresponds (i.e. *len(population) == len(probability)*) to the population array passed in.
- **index_map** (Optional[*IndexMap*]) – A mapping between the provided index (which may contain ints, floats, datetimes or any arbitrary combination of them) and an integer index into the random number array.

Returns The sub-population of the simulants for whom the event occurred. The return type will be the same as type(population)

Return type pd.core.generic.PandasObject

class vivarium.framework.randomness.**RandomnessStream**(*key*, *clock*, *seed*, *index_map=None*, *manager=None*, *for_initialization=False*)

A stream for producing common random numbers.

RandomnessStream objects provide an interface to Vivarium's common random number generation. They provide a number of methods for doing common simulation tasks that require random numbers like making decisions among a number of choices.

key
The name of the randomness stream.

clock
A way to get the current simulation time.

seed
An extra number used to seed the random number generation.

Notes

Should not be constructed by client code.

Simulation components get *RandomnessStream* objects by requesting them from the builder provided to them during the setup phase. I.E.:

```
class CeamComponent:
    def setup(self, builder):
        self.randomness_stream = builder.randomness.get_stream('stream_name')
```

See also:

engine.Builder

name

get_draw(*index*, *additional_key=None*)
Get an indexed sequence of floats pulled from a uniform distribution over [0.0, 1.0)

Parameters

- **index** (Index) – An index whose length is the number of random draws made and which indexes the returned *pandas.Series*.
- **additional_key** (Optional[Any]) – Any additional information used to seed random number generation.

Returns A series of random numbers indexed by the provided *pandas.Index*.

Return type pd.Series

get_seed(*additional_key=None*)
Get a randomly generated seed for use with external randomness tools.

Parameters **additional_key** (Optional[Any]) – Any additional information used to create the seed.

Returns A seed for a random number generation that is linked to Vivarium's common random number framework.

Return type int

filter_for_rate (*population*, *rate*, *additional_key=None*)
Decide an event outcome for each individual in a population from rates.

Given a population or its index and an array of associated rates for some event to happen, we create and return the sub-population for whom the event occurred.

Parameters

- **population** (Union[DataFrame, Series, Index]) – A view on the simulants for which we are determining the outcome of an event.
- **rate** (Union[List[~T], Tuple, ndarray, Series]) – A 1d list of rates of the event under consideration occurring which corresponds (i.e. *len(population) == len(probability))* to the population view passed in. The rates must be scaled to the simulation time-step size either manually or as a post-processing step in a rate pipeline.
- **additional_key** (Optional[Any]) – Any additional information used to create the seed.

Returns The index of the simulants for whom the event occurred.

Return type Index

See also:

framework.values() Value/rate pipeline management module.

filter_for_probability (*population*, *probability*, *additional_key=None*)
Decide an event outcome for each individual in a population from probabilities.

Given a population or its index and an array of associated probabilities for some event to happen, we create and return the sub-population for whom the event occurred.

Parameters

- **population** (Union[DataFrame, Series, Index]) – A view on the simulants for which we are determining the outcome of an event.
- **probability** (Union[List[~T], Tuple, ndarray, Series]) – A 1d list of probabilities of the event under consideration occurring which corresponds (i.e. *len(population) == len(probability)* to the population view passed in.
- **additional_key** (Optional[Any]) – Any additional information used to create the seed.

Returns The sub-population of the simulants for whom the event occurred. The return type will be the same as type(population)

Return type Index

choice (*index*, *choices*, *p=None*, *additional_key=None*)
Decides between a weighted or unweighted set of choices.

Given a a set of choices with or without corresponding weights, returns an indexed set of decisions from those choices. This is simply a vectorized way to make decisions with some book-keeping.

Parameters

- **index** (Index) – An index whose length is the number of random draws made and which indexes the returned *pandas.Series*.
- **choices** (Union[List[~T], Tuple, ndarray, Series]) – A set of options to choose from.

- **p** (Union[List[~T], Tuple, ndarray, Series, None]) – The relative weights of the choices. Can be either a 1-d array of the same length as *choices* or a 2-d array with *len(index)* rows and *len(choices)* columns. In the 1-d case, the same set of weights are used to decide among the choices for every item in the *index*. In the 2-d case, each row in *p* contains a separate set of weights for every item in the *index*.
- **additional_key** (Optional[Any]) – Any additional information used to seed random number generation.

Returns An indexed set of decisions from among the available *choices*.

Return type pd.Series

Raises *RandomnessError* – If any row in *p* contains *RESIDUAL_CHOICE* and the remaining weights in the row are not normalized or any row of *p contains more than one reference to 'RESIDUAL_CHOICE.*

class vivarium.framework.randomness.**RandomnessManager**

Access point for common random number generation.

configuration_defaults = {'randomness': {'additional_seed': None, 'key_columns': ['

name

setup(*builder*)

get_randomness_stream(*decision_point*, *for_initialization=False*)

Provides a new source of random numbers for the given decision point.

Parameters

- **decision_point** (str) – A unique identifier for a stream of random numbers. Typically represents a decision that needs to be made each time step like 'moves_left' or 'gets_disease'.
- **for_initialization** (bool) – A flag indicating whether this stream is used to generate key initialization information that will be used to identify simulants in the Common Random Number framework. These streams cannot be copied and should only be used to generate the state table columns specified in builder.configuration.randomness.key_columns.

Raises *RandomnessError :* – If another location in the simulation has already created a randomness stream with the same identifier.

Return type *RandomnessStream*

register_simulants(*simulants*)

Adds new simulants to the randomness mapping.

Parameters **simulants** (DataFrame) – A table with state data representing the new simulants. Each simulant should pass through this function exactly once.

Raises *RandomnessError :* – If the provided table does not contain all key columns specified in the configuration.

class vivarium.framework.randomness.**RandomnessInterface**(*manager*)

get_stream(*decision_point*, *for_initialization=False*)

Provides a new source of random numbers for the given decision point.

vivarium provides a framework for Common Random Numbers which allows for variance reduction when modeling counter-factual scenarios. Users interested in causal analysis and comparisons between

simulation scenarios should be careful to use randomness streams provided by the framework wherever randomness is employed.

Parameters

- **decision_point** (str) – A unique identifier for a stream of random numbers. Typically represents a decision that needs to be made each time step like 'moves_left' or 'gets_disease'.
- **for_initialization** (bool) – A flag indicating whether this stream is used to generate key initialization information that will be used to identify simulants in the Common Random Number framework. These streams cannot be copied and should only be used to generate the state table columns specified in `builder.configuration.randomness.key_columns`.

Returns An entry point into the Common Random Number generation framework. The stream provides vectorized access to random numbers and a few other utilities.

Return type *RandomnessStream*

register_simulants(*simulants*)

Registers simulants with the Common Random Number Framework.

Parameters **simulants** (DataFrame) – A section of the state table with new simulants and at least the columns specified in `builder.configuration.randomness.key_columns`. This function should be called as soon as the key columns are generated.

4.5.10 Resource Management

This module provides a tool to manage dependencies on resources within a `vivarium` simulation. These resources take the form of things that can be created and utilized by components, for example columns in the *state table* or *named value pipelines*.

Because these resources need to be created before they can be used, they are sensitive to ordering. The intent behind this tool is to provide an interface that allows other managers to register resources with the resource manager and in turn ask for ordered sequences of these resources according to their dependencies or raise exceptions if this is not possible.

exception vivarium.framework.resource.**ResourceError**

Error raised when a dependency requirement is violated.

class vivarium.framework.resource.**ResourceGroup**(*resource_type*, *resource_names*, *producer*, *dependencies*)

Resource groups are the nodes in the resource dependency graph.

A resource group represents the pool of resources produced by a single callable and all the dependencies necessary to produce that resource. When thinking of the dependency graph, this represents a vertex and all in-edges. This is a local-information representation that can be used to construct the entire dependency graph once all resources are specified.

type

The type of resource produced by this resource group's producer.

Must be one of `RESOURCE_TYPES`.

Return type str

names

The long names (including type) of all resources in this group.

Return type List[str]

producer
The method or object that produces this group of resources.

Return type Any

dependencies
The long names (including type) of dependencies for this group.

Return type List[str]

class vivarium.framework.resource.**ResourceManager**
Manages all the resources needed for population initialization.

name
The name of this manager.

Return type str

graph
The networkx graph representation of the resource pool.

Return type DiGraph

sorted_nodes
Returns a topological sort of the resource graph.

Notes

Topological sorts are not stable. Be wary of depending on order where you shouldn't.

add_resources(*resource_type*, *resource_names*, *producer*, *dependencies*)
Adds managed resources to the resource pool.

Parameters

- **resource_type** (str) – The type of the resources being added. Must be one of RESOURCE_TYPES.
- **resource_names** (List[str]) – A list of names of the resources being added.
- **producer** (Any) – A method or object that will produce the resources.
- **dependencies** (List[str]) – A list of resource names formatted as resource_type.resource_name that the producer requires.

Raises *ResourceError* – If either the resource type is invalid, a component has multiple resource producers for the column resource type, or there are multiple producers of the same resource.

class vivarium.framework.resource.**ResourceInterface**(*manager*)
The resource management system.

A resource in vivarium is something like a state table column or a randomness stream. These resources are used to initialize or alter the state of the simulation. Many of these resources might depend on each other and therefore need to be created or updated in a particular order. These dependency chains can be quite long and complex.

Placing the ordering responsibility on end users makes simulations very fragile and difficult to understand. Instead, the resource management system allows users to only specify local dependencies. The system then uses the local dependency information to construct a full dependency graph, validate that there are no cyclic dependencies, and return resources and their producers in an order that makes sense.

add_resources(*resource_type*, *resource_names*, *producer*, *dependencies*)
Adds managed resources to the resource pool.

Parameters

- **resource_type** (str) – The type of the resources being added. Must be one of RESOURCE_TYPES.
- **resource_names** (List[str]) – A list of names of the resources being added.
- **producer** (Any) – A method or object that will produce the resources.
- **dependencies** (List[str]) – A list of resource names formatted as resource_type.resource_name that the producer requires.

Raises *ResourceError* – If either the resource type is invalid, a component has multiple resource producers for the column resource type, or there are multiple producers of the same resource.

4.5.11 State Machine

A state machine implementation for use in vivarium simulations.

class vivarium.framework.state_machine.**Trigger**
An enumeration.

NOT_TRIGGERED = 0

START_INACTIVE = 1

START_ACTIVE = 2

class vivarium.framework.state_machine.**Transition**(*input_state*, *output_state*, *probability_func=<function Transition.<lambda>>*, *triggered=<Trigger.NOT_TRIGGERED: 0>*)
A process by which an entity might change into a particular state.

Parameters

- **input_state** (State) – The start state of the entity that undergoes the transition.
- **output_state** (State) – The end state of the entity that undergoes the transition.
- **probability_func** (*Callable*) – A method or function that describing the probability of this transition occurring.

name

setup(*builder*)

set_active(*index*)

set_inactive(*index*)

probability(*index*)

class vivarium.framework.state_machine.**State**(*state_id*)
An abstract representation of a particular position in a finite and discrete state space.

state_id
The name of this state. This should be unique

Type str

transition_set
A container for potential transitions out of this state.

Type *TransitionSet*

name

sub_components

setup(*builder*)

next_state(*index*, *event_time*, *population_view*)
Moves a population between different states using information this state's *transition_set*.

Parameters

- **index** (*iterable of ints*) – An iterable of integer labels for the simulants.
- **event_time** (*pandas.Timestamp*) – When this transition is occurring.
- **population_view** (vivarium.framework.population.PopulationView) – A view of the internal state of the simulation.

transition_effect(*index*, *event_time*, *population_view*)
Updates the simulation state and triggers any side-effects associated with entering this state.

Parameters

- **index** (*iterable of ints*) – An iterable of integer labels for the simulants.
- **event_time** (*pandas.Timestamp*) – The time at which this transition occurs.
- **population_view** (*vivarium.framework.population.PopulationView*) – A view of the internal state of the simulation.

cleanup_effect(*index*, *event_time*)

add_transition(*output*, *probability_func=<function State.<lambda>>*, *triggered=<Trigger.NOT_TRIGGERED: 0>*)
Builds a transition from this state to the given state.

output [State] The end state after the transition.

allow_self_transitions()

class vivarium.framework.state_machine.**Transient**
Used to tell _next_state to transition a second time.

class vivarium.framework.state_machine.**TransientState**(*state_id*)

class vivarium.framework.state_machine.**TransitionSet**(*state_name*, **transitions*, *allow_null_transition=False*)
A container for state machine transitions.

Parameters

- **state_name** (*object*) – The unique name of the state that instantiated this TransitionSet. Typically a string but any object implementing __str__ will do.
- **iterable** (*iterable*) – Any iterable whose elements are *Transition* objects.
- **allow_null_transition** (*bool, optional*) –

name

sub_components

setup (*builder*)
Performs this component's simulation setup and return sub-components.

Parameters **builder** (*engine.Builder*) – Interface to several simulation tools including access to common random number generation, in particular.

Returns This component's sub-components.

Return type iterable

choose_new_state (*index*)
Chooses a new state for each simulant in the index.

Parameters **index** (*iterable of ints*) – An iterable of integer labels for the simulants.

Returns

- **outputs** (*list*) – The possible end states of this set of transitions.
- **decisions** (*pandas.Series*) – A series containing the name of the next state for each simulant in the index.

append (*transition*)

extend (*transitions*)

class vivarium.framework.state_machine.**Machine** (*state_column*, *states=None*)
A collection of states and transitions between those states.

states
The collection of states represented by this state machine.

Type iterable of *State* objects

state_column
A label for the piece of simulation state governed by this state machine.

Type str

population_view
A view of the internal state of the simulation.

Type *pandas.DataFrame*

name

sub_components

setup (*builder*)
Performs this component's simulation setup and return sub-components.

Parameters **builder** (*engine.Builder*) – Interface to several simulation tools including access to common random number generation, in particular.

Returns This component's sub-components.

Return type iterable

add_states (*states*)

transition (*index*, *event_time*)
Finds the population in each state and moves them to the next state.

Parameters

- **index** (*iterable of ints*) – An iterable of integer labels for the simulants.
- **event_time** (*pandas.Timestamp*) – The time at which this transition occurs.

cleanup(*index*, *event_time*)

to_dot()
Produces a ball and stick graph of this state machine.

Returns A ball and stick visualization of this state machine.

Return type *graphviz.Digraph*

4.5.12 The Simulation Clock

The components here provide implementations of different kinds of simulation clocks for use in `vivarium`.

For more information about time in the simulation, see the associated *concept note*.

class vivarium.framework.time.**SimulationClock**
Defines a base implementation for a simulation clock.

name

time
The current simulation time.

Return type Union[datetime, Number]

stop_time
The time at which the simulation will stop.

Return type Union[datetime, Number]

step_size
The size of the next time step.

Return type Union[timedelta, Number]

step_forward()
Advances the clock by the current step size.

Return type None

step_backward()
Rewinds the clock by the current step size.

class vivarium.framework.time.**SimpleClock**
A unitless step-count based simulation clock.

configuration_defaults = {'time': {'end': 100, 'start': 0, 'step_size': 1}}

name

setup(*builder*)

vivarium.framework.time.**get_time_stamp**(*time*)

class vivarium.framework.time.**DateTimeClock**
A date-time based simulation clock.

configuration_defaults = {'time': {'end': {'day': 2, 'month': 7, 'year': 2010}, '

name

setup(*builder*)

class vivarium.framework.time.**TimeInterface**(*manager*)

clock()
Gets a callable that returns the current simulation time.

Return type Callable[[], Union[datetime, Number]]

step_size()
Gets a callable that returns the current simulation step size.

Return type Callable[[], Union[timedelta, Number]]

4.5.13 Framework Utility Functions

Collection of utility functions shared by the vivarium framework.

vivarium.framework.utilities.**from_yearly**(*value*, *time_step*)

vivarium.framework.utilities.**to_yearly**(*value*, *time_step*)

vivarium.framework.utilities.**rate_to_probability**(*rate*)

vivarium.framework.utilities.**probability_to_rate**(*probability*)

vivarium.framework.utilities.**collapse_nested_dict**(*d*, *prefix=None*)

vivarium.framework.utilities.**import_by_path**(*path*)
Import a class or function given it's absolute path.

Parameters **path** (str) – Path to object to import

Return type Callable

vivarium.framework.utilities.**handle_exceptions**(*func*, *logger*, *with_debugger*)
Drops a user into an interactive debugger if func raises an error.

Return type Callable

4.5.14 The Value Pipeline System

The value pipeline system is a vital part of the vivarium infrastructure. It allows for values that determine the behavior of individual *simulants* to be constructed across across multiple *components*.

For more information about when and how you should use pipelines in your simulations, see the value system *concept note*.

exception vivarium.framework.values.**DynamicValueError**
Indicates an improperly configured value was invoked.

vivarium.framework.values.**replace_combiner**(*value*, *mutator*, **args*, ***kwargs*)
Replace the previous pipeline output with the output of the mutator.

This is the default combiner.

Parameters

- **value** (*~T*) – The value from the previous step in the pipeline.
- **mutator** (Callable[..., ~T]) – A callable that takes in all arguments that the pipeline source takes in plus an additional last positional argument for the value from the previous stage in the pipeline.
- **kwargs** (*args,*) – The same args and kwargs provided during the invocation of the pipeline.

Returns

Return type A modified version of the input value.

vivarium.framework.values.**list_combiner**(*value*, *mutator*, **args*, ***kwargs*)

Aggregates source and mutator output into a list.

This combiner is meant to be used with a post-processor that does some kind of reduce operation like summing all values in the list.

Parameters

- **value** (List[~T]) – A list of all values provided by the source and prior mutators in the pipeline.
- **mutator** (Callable[..., ~T]) – A callable that returns some portion of this pipeline's final value.
- **kwargs** (*args,*) – The same args and kwargs provided during the invocation of the pipeline.

Return type List[~T]

Returns

- *The input list with new mutator portion of the pipeline value*
- *appended to it.*

vivarium.framework.values.**rescale_post_processor**(*value*, *time_step*)

Rescales annual rates to time-step appropriate rates.

This should only be used with a simulation using a *DateTimeClock* or another implementation of a clock that traffics in pandas date-time objects.

Parameters

- **value** (Union[ndarray, Series, DataFrame, Number]) – Annual rates, either as a number or something we can broadcast multiplication over like a numpy array or pandas data frame.
- **time_step** (Timedelta) – A pandas time delta representing the size of the upcoming time step.

Returns

Return type The annual rates rescaled to the size of the current time step size.

vivarium.framework.values.**union_post_processor**(*values*, *_*)

Computes a probability on the union of the sample spaces in the values.

Given a list of values where each value is a probability of an independent event, this post processor computes the probability of the union of the events.

Parameters **values** (List[Union[ndarray, Series, DataFrame, Number]]) – A list of independent proportions or probabilities, either as numbers or as a something we can broadcast addition and multiplication over.

Return type Union[ndarray, Series, DataFrame, Number]

Returns

- *The probability over the union of the sample spaces represented*
- *by the original probabilities.*

class vivarium.framework.values.**Pipeline**

A tool for building up values across several components.

Pipelines are lazily initialized so that we don't have to put constraints on the order in which components are created and set up. The values manager will configure a pipeline (set all of its attributes) when the pipeline source is created.

As long as a pipeline is not actually called in a simulation, it does not need a source or to be configured. This might occur when writing generic components that create a set of pipeline modifiers for values that won't be used in the particular simulation.

name

The name of the value represented by this pipeline.

source

A callable source for this pipeline's value.

mutators

A list of callables that directly modify the pipeline source or contribute portions of the value.

combiner

A strategy for combining the source and mutator values into the final value represented by the pipeline.

post_processor

An optional final transformation to perform on the combined output of the source and mutators.

manager

A reference to the simulation values manager.

class vivarium.framework.values.**ValuesManager**

Manager for the dynamic value system.

name

setup(*builder*)

on_post_setup(*_*)

Finalizes dependency structure for the pipelines.

register_value_producer(*value_name*, *source*, *requires_columns=()*, *requires_values=()*, *requires_streams=()*, *preferred_combiner=<function replace_combiner>*, *preferred_post_processor=None*)

Marks a `Callable` as the producer of a named value.

Return type `Callable`

register_value_modifier(*value_name*, *modifier*, *requires_columns=()*, *requires_values=()*, *requires_streams=()*)

Marks a `Callable` as the modifier of a named value.

Parameters

- **value_name** (`str`) – The name of the dynamic value pipeline to be modified.
- **modifier** (`Callable`) – A function that modifies the source of the dynamic value pipeline when called. If the pipeline has a `replace_combiner`, the modifier should accept the same arguments as the pipeline source with an additional last positional argument for the results of the previous stage in the pipeline. For the `list_combiner` strategy, the pipeline modifiers should have the same signature as the pipeline source.
- **requires_columns** (`List[str]`) – A list of the state table columns that already need to be present and populated in the state table before the pipeline modifier is called.

- **requires_values** (List[str]) – A list of the value pipelines that need to be properly sourced before the pipeline modifier is called.
- **requires_streams** (List[str]) – A list of the randomness streams that need to be properly sourced before the pipeline modifier is called.

get_value(*name*)
Retrieve the pipeline representing the named value.

Parameters **name** – Name of the pipeline to return.

Returns

- *A callable reference to the named pipeline. The pipeline arguments*
- *should be identical to the arguments to the pipeline source*
- (frequently just a pandas.Index representing the
- *simulants).*

keys()
Get an iterable of pipeline names.

Return type Iterable[str]

items()
Get an iterable of name, pipeline tuples.

Return type Iterable[Tuple[str, *Pipeline*]]

values()
Get an iterable of all pipelines.

Return type Iterable[*Pipeline*]

class vivarium.framework.values.**ValuesInterface**(*manager*)
Public interface for the simulation values management system.

The values system provides tools to build up a value across many components, allowing users to build components that focus on small groups of simulant attributes.

register_value_producer(*value_name*, *source*, *requires_columns=()*, *requires_values=()*, *requires_streams=()*, *preferred_combiner=<function replace_combiner>*, *preferred_post_processor=None*)
Marks a Callable as the producer of a named value.

Parameters

- **value_name** (str) – The name of the new dynamic value pipeline.
- **source** (Callable) – A callable source for the dynamic value pipeline.
- **requires_columns** (List[str]) – A list of the state table columns that already need to be present and populated in the state table before the pipeline source is called.
- **requires_values** (List[str]) – A list of the value pipelines that need to be properly sourced before the pipeline source is called.
- **requires_streams** (List[str]) – A list of the randomness streams that need to be properly sourced before the pipeline source is called.
- **preferred_combiner** (Callable) – A strategy for combining the source and the results of any calls to mutators in the pipeline. vivarium provides the strategies replace_combiner (the default) and list_combiner, which are importable from

vivarium.framework.values. Client code may define additional strategies as necessary.

- **preferred_post_processor** (Optional[Callable]) – A strategy for processing the final output of the pipeline. vivarium provides the strategies rescale_post_processor and union_post_processor which are importable from vivarium.framework.values. Client code may define additional strategies as necessary.

Returns

Return type A callable reference to the named dynamic value pipeline.

register_rate_producer(*rate_name*, *source*, *requires_columns=()*, *requires_values=()*, *requires_streams=()*)

Marks a Callable as the producer of a named rate.

This is a convenience wrapper around register_value_producer that makes sure rate data is appropriately scaled to the size of the simulation time step. It is equivalent to register_value_producer(value_name, source, preferred_combiner=replace_combiner, preferred_post_processor=rescale_post_processo

Parameters

- **rate_name** (str) – The name of the new dynamic rate pipeline.
- **source** (Callable[..., DataFrame]) – A callable source for the dynamic rate pipeline.
- **requires_columns** (List[str]) – A list of the state table columns that already need to be present and populated in the state table before the pipeline source is called.
- **requires_values** (List[str]) – A list of the value pipelines that need to be properly sourced before the pipeline source is called.
- **requires_streams** (List[str]) – A list of the randomness streams that need to be properly sourced before the pipeline source is called.

Returns

Return type A callable reference to the named dynamic rate pipeline.

register_value_modifier(*value_name*, *modifier*, *requires_columns=()*, *requires_values=()*, *requires_streams=()*)

Marks a Callable as the modifier of a named value.

Parameters

- **value_name** (str) – The name of the dynamic value pipeline to be modified.
- **modifier** (Callable) – A function that modifies the source of the dynamic value pipeline when called. If the pipeline has a replace_combiner, the modifier should accept the same arguments as the pipeline source with an additional last positional argument for the results of the previous stage in the pipeline. For the list_combiner strategy, the pipeline modifiers should have the same signature as the pipeline source.
- **requires_columns** (List[str]) – A list of the state table columns that already need to be present and populated in the state table before the pipeline modifier is called.
- **requires_values** (List[str]) – A list of the value pipelines that need to be properly sourced before the pipeline modifier is called.
- **requires_streams** (List[str]) – A list of the randomness streams that need to be properly sourced before the pipeline modifier is called.

get_value(*name*)
Retrieve the pipeline representing the named value.

Parameters **name** (str) – Name of the pipeline to return.

Return type *Pipeline*

Returns

- *A callable reference to the named pipeline. The pipeline arguments*
- *should be identical to the arguments to the pipeline source*
- (frequently just a pandas.Index representing the
- *simulants).*

4.5.15 Data Artifact Management

HDF Interface

A convenience wrapper around the tables and pandas HDF interfaces.

Public Interface

The public interface consists of 5 functions:

Table 1: HDF Public Interface

Function	Description
touch()	Creates an HDF file, wiping an existing file if necessary.
write()	Stores data at a key in an HDF file.
load()	Loads (potentially filtered) data from a key in an HDF file.
remove()	Clears data from a key in an HDF file.
get_keys()	Gets all available HDF keys from an HDF file.

Contracts

- All functions in the public interface accept both pathlib.Path and normal Python str objects for paths.
- All functions in the public interface accept only str objects as representations of the keys in the hdf file. The strings must be formatted as "type.name.measure" or "type.measure".

vivarium.framework.artifact.hdf.**touch**(*path*)
Creates an HDF file, wiping an existing file if necessary.

If the given path is proper to create a HDF file, it creates a new HDF file.

Parameters **path** (Union[str, Path]) – The path to the HDF file.

Raises ValueError – If the non-proper path is given to create a HDF file.

vivarium.framework.artifact.hdf.**write**(*path*, *entity_key*, *data*)
Writes data to the HDF file at the given path to the given key.

Parameters

- **path** (Union[str, Path]) – The path to the HDF file to write to.

- **entity_key** (str) – A string representation of the internal HDF path where we want to write the data. The key must be formatted as "type.name.measure" or "type.measure".
- **data** (Any) – The data to write. If it is a pandas object, it will be written using a pandas.HDFStore or pandas.to_hdf(). If it is some other kind of python object, it will first be encoded as json with json.dumps() and then written to the provided key.

Raises ValueError – If the path or entity_key are improperly formatted.

vivarium.framework.artifact.hdf.**load**(*path*, *entity_key*, *filter_terms*, *column_filters*)

Loads data from an HDF file.

Parameters

- **path** (Union[str, Path]) – The path to the HDF file to load the data from.
- **entity_key** (str) – A representation of the internal HDF path where the data is located.
- **filter_terms** (Optional[List[str]]) – An optional list of terms used to filter the rows in the data. The terms must be formatted in a way that is suitable for use with the where argument of pd.read_hdf(). Only filters applying to existing columns in the data are used.
- **column_filters** (Optional[List[str]]) – An optional list of columns to load from the data.

Raises ValueError – If the path or entity_key are improperly formatted.

Returns

Return type The data stored at the the given key in the HDF file.

vivarium.framework.artifact.hdf.**remove**(*path*, *entity_key*)

Removes a piece of data from an HDF file.

Parameters

- **path** (Union[str, Path]) – The path to the HDF file to remove the data from.
- **entity_key** (str) – A representation of the internal HDF path where the data is located.

Raises ValueError – If the path or entity_key are improperly formatted.

vivarium.framework.artifact.hdf.**get_keys**(*path*)

Gets key representation of all paths in an HDF file.

Parameters **path** (str) – The path to the HDF file.

Returns

Return type A list of key representations of the internal paths in the HDF.

class vivarium.framework.artifact.hdf.**EntityKey**(*key*)

A convenience wrapper that translates artifact keys.

This class provides several representations of the artifact keys that are useful when working with the pandas and tables HDF interfaces.

type

The type of the entity represented by the key.

Return type str

name

The name of the entity represented by the key

Return type str

measure
The measure associated with the data represented by the key.

Return type str

group_prefix
The HDF group prefix for the key.

Return type str

group_name
The HDF group name for the key.

Return type str

group
The full path to the group for this key.

Return type str

path
The full HDF path associated with this key.

Return type str

with_measure(*measure*)
Replaces this key's measure with the provided one.

Parameters **measure** (str) – The measure to replace this key's measure with.

Returns

Return type A new EntityKey with the updated measure.

4.5.16 Component Management

The Component Manager System

The vivarium component manager system is responsible for maintaining a reference to all of the managers and components in a simulation, providing an interface for adding additional components or managers, and applying default configurations and initiating the setup stage of the lifecycle. This module provides the default implementation and interface.

The *ComponentManager* is the first plugin loaded by the *SimulationContext* and managers and components are given to it by the context. It is called on to setup everything it holds when the context itself is setup.

exception vivarium.framework.components.manager.**ComponentConfigError**
Error while interpreting configuration file or initializing components

class vivarium.framework.components.manager.**OrderedComponentSet**(**args*)
A container for Vivarium components.

It preserves ordering, enforces uniqueness by name, and provides a subset of set-like semantics.

add(*component*)

update(*components*)

pop()

Return type Any

class vivarium.framework.components.manager.**ComponentManager**

Manages the initialization and setup of vivarium components.

Maintains references to all components and managers in a vivarium simulation, applies their default configuration and initiates their setup life-cycle stage.

The component manager maintains a separate list of managers and components and provides methods for adding to these lists and getting members that correspond to a specific type. It also initiates the setup lifecycle phase for all components and managers it controls. This is done first for managers and then components, and involves applying default configurations and calling the object's setup method.

name

The name of this component.

setup(*configuration*, *lifecycle_manager*)

Called by the simulation context.

add_managers(*managers*)

Registers new managers with the component manager.

Managers are configured and setup before components.

Parameters **managers** (Union[List[Any], Tuple[Any]]) – Instantiated managers to register.

add_components(*components*)

Register new components with the component manager.

Components are configured and setup after managers.

Parameters **components** (Union[List[Any], Tuple[Any]]) – Instantiated components to register.

get_components_by_type(*component_type*)

Get all components currently held by the component manager that are an instance of component_type.

Parameters **component_type** (Union[type, Tuple[type]]) – A component type.

Returns

Return type A list of components of type component_type.

get_component(*name*)

Get the component that has name if presently held by the component manager. Names are guaranteed to be unique.

Parameters **name** (str) – A component name.

Returns

Return type A component that has name name.

Raises ValueError – No component exists in the component manager with name.

list_components()

Get a mapping of component names to components held by the manager.

Returns

Return type A mapping of component names to components.

setup_components(*builder*)

Separately configure and set up the managers and components held by the component manager, in that order.

The setup process involves applying default configurations and then calling the manager or component's setup method. This can result in new components as a side effect of setup because components themselves have access to this interface through the builder in their setup method.

Parameters **builder** – Interface to several simulation tools.

apply_configuration_defaults(*component*)

class vivarium.framework.components.manager.**ComponentInterface**(*manager*)

The builder interface for the component manager system. This class defines component manager methods a vivarium component can access from the builder. It provides methods for querying and adding components to the *ComponentManager*.

get_component(*name*)

Get the component that has name if presently held by the component manager. Names are guaranteed to be unique.

Parameters **name** (str) – A component name.

Returns

Return type A component that has name name.

get_components_by_type(*component_type*)

Get all components that are an instance of component_type currently held by the component manager.

Parameters **component_type** (Type[+CT_co]) – A component type to retrieve, compared against internal components using isinstance().

Returns

Return type A list of components of type component_type.

list_components()

Get a mapping of component names to components held by the manager.

Returns

Return type A dictionary mapping component names to components.

The Component Configuration Parser

The *ComponentConfigurationParser* is responsible for taking a list or hierarchical ConfigTree method of the parser, which is used anytime a simulation is initialized from a model specification file.

There are three steps to this process.

1. Parsing the model specification's components
2. Validating the arguments and prepping each component
3. Importing and instantiating the actual components

exception vivarium.framework.components.parser.**ParsingError**

Error raised when component configurations are not specified correctly.

class vivarium.framework.components.parser.**ComponentConfigurationParser**

Parses component configuration from model specification and initializes components.

To define your own set of parsing rules, you should write a parser class that inherits from this class and overrides parse_component_config. You can then define a set of parsing rules that turn component configuration information into a list of strings where each string is the full python import path to the class followed by a set of parentheses containing initialization arguments.

For example, say your component configuration specifies `FancyClass` from the `important_module` of your `made_up_package` and that `FancyClass` has two initialization parameters, `'important_thing1'` and `'important_thing2'`. Your implementation of `parse_component_config` needs to generate the string `'made_up_package.important_module.FancyClass("important_thing1", "important_thing2")'` and include it in the list of components to generate.

All classes that are initialized from the `yaml` configuration must either take no arguments or take arguments specified as strings.

get_components(*component_config*)

Extracts component specifications from configuration information and returns initialized components.

This method encapsulates the three steps described above of parsing, validating/prepping, and importing/instantiating.

The first step of parsing is only done for component configurations that come in as a `ConfigTree`. Configurations that are provided in the form of a list are already assumed to be in the correct form.

Parameters **component_config** (Union[*ConfigTree*, List[~T]]) – A hierarchical component specification blob. This configuration information needs to be parsable into a full import path and a set of initialization arguments by the `parse_component_config` method.

Returns

Return type A list of initialized components.

parse_component_config(*component_config*)

Parses a hierarchical component specification into a list of standardized component definitions.

This default parser expects component configurations as a list of dicts. Each dict at the top level corresponds to a different package and has a single key. This key may be just the name of the package or a Python style import path to the module in which components live. The values of the top level dicts are a list of dicts or strings. If dicts, the keys are another step along the import path. If strings, the strings are representations of calls to the class constructor of components to be generated. This pattern may be arbitrarily nested.

Parameters **component_config** (Dict[str, Union[Dict[~KT, ~VT], List[~T]]]) – A hierarchical component specification blob.

Return type List[str]

Returns

- *A list of standardized component definitions. Component definition*
- *strings are specified as*
- `'absolute.import.path.ClassName("argument1", "argument2", ...)'`.

vivarium.framework.components.parser.**parse_component_config_to_list**(*component_config*)

Helper function for parsing hierarchical component configuration into a flat list.

This function recursively walks the component configuration dictionary, treating it like a prefix tree and building the import path prefix. When it hits a list, it prepends the built prefix onto each item in the list. If the dictionary contains multiple lists, the prefix-prepended lists are concated together. For example, a component configuration dictionary like the following:

```
component_config = {
    'vivarium_examples': {
```

(continues on next page)

(continued from previous page)

```
        'disease_model': {
            'population': ['BasePopulation()', 'Mortality()']
            'disease': ['SIS_DiseaseModel("diarrhea")]
        }
    }
}
```

would be parsed by this function into the following list:

```
['vivarium_examples.disease_model.population.BasePopulation()',
 'vivarium_examples.disease_model.population.Mortality()',
 'vivarium_examples.disease_model.disease.SIS_DiseaseModel("diarrhea")']
```

Parameters **component_config** (Dict[str, Union[Dict[~KT, ~VT], List[~T]]]) – A hierarchical component specification blob.

Return type List[str]

Returns

- *A flat list of strings, each string representing the full python import*
- *path to the component, the component name, and any arguments.*

vivarium.framework.components.parser.**prep_components**(*component_list*)

Transform component description strings into tuples of component paths and required arguments.

Parameters **component_list** (Union[List[str], Tuple[str]]) – The component descriptions to transform.

Returns

Return type List of component/argument tuples.

vivarium.framework.components.parser.**clean_args**(*args*, *path*)

Transform component arguments into a tuple, validating that each argument is a string.

Parameters

- **args** (List[~T]) – List of arguments to the component specified at path.
- **path** (str) – Path representing the component for which arguments are being cleaned.

Returns

Return type A tuple of arguments, each of which is guaranteed to be a string.

vivarium.framework.components.parser.**import_and_instantiate_components**(*component_list*)

Transform the list of tuples representing components into the actual instantiated component objects.

Parameters **component_list** (List[Tuple[str, Tuple[str]]]) – A list of tuples representing components, where the first element of each tuple is a string with the full import path to the component and the component name and the second element is a tuple of the arguments to the component.

Returns

Return type A list of instantiated component objects.

4.6 Interface

4.6.1 Vivarium Command Line Tools

`vivarium` provides the tool **`simulate`** for running simulations from the command line. It provides three subcommands:

Table 2: `simulate` sub-commands

Name	Description
run	Runs a single simulation from a model specification file.
test	Runs an example simulation that comes packaged with `vivarium`. | Useful as an installation test.
profile	Produces a profile of a simulation using the python `cProfile` module

For more information, see the *tutorial* on running simulations from the command line.

simulate

A command line utility for running a single simulation.

You may initiate a new run with the `run` sub-command, initiate a test run of a provided model specification with the `test` subcommand, or profile a simulation run with the `profile` subcommand.

```
simulate [OPTIONS] COMMAND [ARGS]...
```

profile

Run a simulation based on the provided MODEL_SPECIFICATION and profile the run.

```
simulate profile [OPTIONS] MODEL_SPECIFICATION
```

Options

`-o, --results_directory` `<results_directory>`
The directory to write results to. A folder will be created in this directory with the same name as the configuration file.

`--process, --no-process`
Automatically process the profile to human readable format with pstats, sorted by cumulative runtime, and dump to a file

Arguments

MODEL_SPECIFICATION
Required argument

run

Run a simulation from the command line.

The simulation itself is defined by the given MODEL_SPECIFICATION yaml file.

Within the results directory, which defaults to ~/vivarium_results if none is provided, a subdirectory will be created with the same name as the MODEL_SPECIFICATION if one does not exist. Results will be written to a further subdirectory named after the start time of the simulation run.

```
simulate run [OPTIONS] MODEL_SPECIFICATION
```

Options

-o, --results_directory <results_directory>
The directory to write results to. A folder will be created in this directory with the same name as the configuration file.

-v, --verbose
Report each time step.

--pdb
Drop into python debugger if an error occurs.

Arguments

MODEL_SPECIFICATION
Required argument

test

Run a test simulation using the `disease_model.yaml` model specification provided in the examples directory.

```
simulate test [OPTIONS]
```

4.6.2 Vivarium Interactive Tools

This module provides an interface for interactive simulation usage. The main part is the *InteractiveContext*, a sub-class of the main simulation object in `vivarium` that has been extended to include convenience methods for running and exploring the simulation in an interactive setting.

See the associated tutorials for *running* and *exploring* for more information.

class vivarium.interface.interactive.**InteractiveContext**(**args*, *setup=True*, ***kwargs*)
A simulation context with helper methods for running simulations interactively.

This class should not be instantiated directly. It should be created with a call to one of the helper methods provided in this module.

setup()

step(*step_size=None*)
Advance the simulation one step.

Parameters **step_size** (Union[timedelta, Number, None]) – An optional size of step to take. Must be the same type as the simulation clock's step size (usually a pandas.Timedelta).

run(*with_logging=True*)
Run the simulation for the duration specified in the configuration.

Parameters **with_logging** (bool) – Whether or not to log the simulation steps. Only works in an ipython environment.

Returns

Return type The number of steps the simulation took.

run_for(*duration*, *with_logging=True*)
Run the simulation for the given time duration.

Parameters

- **duration** (Union[timedelta, Number]) – The length of time to run the simulation for. Should be the same type as the simulation clock's step size (usually a pandas Timedelta).
- **with_logging** (bool) – Whether or not to log the simulation steps. Only works in an ipython environment.

Returns

Return type The number of steps the simulation took.

run_until(*end_time*, *with_logging=True*)
Run the simulation until the provided end time.

Parameters

- **end_time** (Union[datetime, Number]) – The time to run the simulation until. The simulation will run until its clock is greater than or equal to the provided end time.
- **with_logging** (bool) – Whether or not to log the simulation steps. Only works in an ipython environment.

Returns

Return type The number of steps the simulation took.

take_steps(*number_of_steps=1*, *step_size=None*, *with_logging=True*)
Run the simulation for the given number of steps.

Parameters

- **number_of_steps** (int) – The number of steps to take.
- **step_size** (Union[timedelta, Number, None]) – An optional size of step to take. Must be the same type as the simulation clock's step size (usually a pandas.Timedelta).
- **with_logging** (bool) – Whether or not to log the simulation steps. Only works in an ipython environment.

get_population(*untracked=False*)
Get a copy of the population state table.

Parameters **untracked** (bool) – Whether or not to return simulants who are no longer being tracked by the simulation.

Return type DataFrame

list_values()
List the names of all pipelines in the simulation.

Return type List[str]

get_value(*value_pipeline_name*)
Get the value pipeline associated with the given name.

Return type *Pipeline*

list_events()
List all event types registered with the simulation.

Return type List[str]

get_listeners(*event_type*)
Get all listeners of a particular type of event.

Available event types can be found by calling *InteractiveContext.list_events()*.

Parameters **event_type** (str) – The type of event to grab the listeners for.

Return type List[Callable]

get_emitter(*event_type*)
Get the callable that emits the given type of events.

Available event types can be found by calling *InteractiveContext.list_events()*.

Parameters **event_type** (str) – The type of event to grab the listeners for.

Return type Callable

list_components()
Get a mapping of component names to components currently in the simulation.

Returns

Return type A dictionary mapping component names to components.

get_component(*name*)
Get the component in the simulation that has name, if present. Names are guaranteed to be unique.

Parameters **name** (str) – A component name.

Returns

Return type A component that has the name name else None.

4.6.3 Interface Utility Functions

The functions defined here are used to support the interactive and command-line interfaces for vivarium.

vivarium.interface.utilities.**run_from_ipython**()
Taken from https://stackoverflow.com/questions/5376837/how-can-i-do-an-if-run-from-ipython-test-in-python

Return type bool

vivarium.interface.utilities.**log_progress**(*sequence*, *every=None*, *size=None*, *name='Items'*)

Taken from https://github.com/alexanderkuk/log-progress

exception vivarium.interface.utilities.**InteractiveError**

Error raised when the Interactive context is in an inconsistent state.

vivarium.interface.utilities.**raise_if_not_setup**(*system_type*)

vivarium.interface.utilities.**get_output_root**(*results_directory*, *model_specification_file*)

vivarium.interface.utilities.**add_logging_sink**(*sink*, *verbose*, *colorize=False*, *serialize=False*)

vivarium.interface.utilities.**configure_logging_to_terminal**(*verbose*)

vivarium.interface.utilities.**configure_logging_to_file**(*output_directory*)

CHAPTER 5

Glossary

Attribute A variable associated with each *simulant*. For example, each simulant may have an attribute to describe their age or position.

Component Any self-contained piece of code that can be plugged into the simulation to add some functionality. In `vivarium` we typically think of components as encapsulating and managing some behavior or attributes of the *simulants*.

Configuration The configuration is a set of parameters a user can modify to adjust the behavior of *components*. Components themselves may provide defaults for several configuration parameters.

Metrics A special *pipeline* in the simulation that produces the simulation outputs.

Model Specification A complete description of a `vivarium` model. This description details all *plugins*, *components*, and *configuration* required to run the model. A model specification is stored in a yaml model specification file which is parsable by the simulation framework.

Pipeline A `vivarium` value pipeline. A pipeline is a framework tool that allows users to dynamically construct and share data across several *components*.

Plugin A plugin is a python class intended to add additional functionality to the core framework. Unlike a normal *component* which adds new behaviors and attributes to *simulants*, a plugin adds new services to the framework itself. Examples might include a new simulation clock, a results handling service, or a logging service.

Simulant An individual or agent. One member of the population being simulated.

State Table The core representation of the population in the simulation. The state table consists of a row for each *simulant* and a column for each *attribute* the simulant possesses.

Python Module Index

v

Index

Symbols

A

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W