## Contents

<table>
<thead>
<tr>
<th>1</th>
<th>Full contents</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Covasim overview</td>
<td>3</td>
</tr>
<tr>
<td>1.1.1</td>
<td>About Covasim</td>
<td>3</td>
</tr>
<tr>
<td>1.1.2</td>
<td>Background</td>
<td>4</td>
</tr>
<tr>
<td>1.1.3</td>
<td>Requirements</td>
<td>4</td>
</tr>
<tr>
<td>1.1.4</td>
<td>Quick start guide</td>
<td>5</td>
</tr>
<tr>
<td>1.1.5</td>
<td>Full installation instructions</td>
<td>5</td>
</tr>
<tr>
<td>1.1.6</td>
<td>Usage examples</td>
<td>5</td>
</tr>
<tr>
<td>1.1.7</td>
<td>Module structure</td>
<td>5</td>
</tr>
<tr>
<td>1.1.8</td>
<td>Other folders</td>
<td>6</td>
</tr>
<tr>
<td>1.1.8.1</td>
<td>Bin</td>
<td>6</td>
</tr>
<tr>
<td>1.1.8.2</td>
<td>Data</td>
<td>6</td>
</tr>
<tr>
<td>1.1.8.3</td>
<td>Tutorials</td>
<td>7</td>
</tr>
<tr>
<td>1.1.8.4</td>
<td>Examples</td>
<td>7</td>
</tr>
<tr>
<td>1.1.8.5</td>
<td>Licenses</td>
<td>7</td>
</tr>
<tr>
<td>1.1.8.6</td>
<td>Tests</td>
<td>7</td>
</tr>
<tr>
<td>1.1.9</td>
<td>Disclaimer</td>
<td>7</td>
</tr>
<tr>
<td>1.2</td>
<td>Tutorials</td>
<td>7</td>
</tr>
<tr>
<td>1.2.1</td>
<td>T1 - Getting started</td>
<td>8</td>
</tr>
<tr>
<td>1.2.1.1</td>
<td>Hello world</td>
<td>8</td>
</tr>
<tr>
<td>1.2.1.2</td>
<td>Defining parameters and running simulations</td>
<td>9</td>
</tr>
<tr>
<td>1.2.1.3</td>
<td>Plotting results</td>
<td>11</td>
</tr>
<tr>
<td>1.2.1.4</td>
<td>Full usage example</td>
<td>12</td>
</tr>
<tr>
<td>1.2.2</td>
<td>T2 - Plotting, printing, and saving</td>
<td>13</td>
</tr>
<tr>
<td>1.2.2.1</td>
<td>Global plotting configuration</td>
<td>14</td>
</tr>
<tr>
<td>1.2.2.2</td>
<td>Printing objects</td>
<td>14</td>
</tr>
<tr>
<td>1.2.2.3</td>
<td>Plotting options</td>
<td>16</td>
</tr>
<tr>
<td>1.2.2.4</td>
<td>Saving options</td>
<td>19</td>
</tr>
<tr>
<td>1.2.3</td>
<td>T3 - Running scenarios</td>
<td>22</td>
</tr>
<tr>
<td>1.2.3.1</td>
<td>Running with MultiSims</td>
<td>22</td>
</tr>
<tr>
<td>1.2.3.2</td>
<td>Running with Scenarios</td>
<td>29</td>
</tr>
<tr>
<td>1.2.4</td>
<td>T4 - People and data</td>
<td>31</td>
</tr>
<tr>
<td>1.2.4.1</td>
<td>Data requirements</td>
<td>31</td>
</tr>
<tr>
<td>1.2.4.2</td>
<td>Demographic data</td>
<td>32</td>
</tr>
<tr>
<td>1.2.4.3</td>
<td>Epidemiological data scrapers</td>
<td>32</td>
</tr>
<tr>
<td>1.2.4.4</td>
<td>Data input and formats</td>
<td>33</td>
</tr>
</tbody>
</table>
1.2.4.5 People and contact network layers ................................................. 34
1.2.5 T5 - Using interventions ................................................................. 36
  1.2.5.1 Change beta and clip edges ......................................................... 36
  1.2.5.2 Testing interventions ................................................................. 39
  1.2.5.3 Contact tracing ......................................................................... 41
  1.2.5.4 Simple vaccination .................................................................... 43
  1.2.5.5 Age targeting and other subtargeting ........................................... 43
  1.2.5.6 Dynamic parameters ................................................................. 45
  1.2.5.7 Dynamic triggering ................................................................. 46
  1.2.5.8 Custom interventions ............................................................... 47
1.2.6 T6 - Using analyzers ................................................................. 51
  1.2.6.1 Age histograms and snapshots ..................................................... 51
  1.2.6.2 Transmission trees ................................................................. 52
  1.2.6.3 Custom analyzers ................................................................. 54
1.2.7 T7 - Calibration ................................................................. 57
  1.2.7.1 The Fit object ................................................................. 57
  1.2.7.2 Calibration approaches ........................................................... 60
  1.2.7.3 Built-in calibration ................................................................. 60
1.2.8 T8 - Vaccines and variants ......................................................... 70
  1.2.8.1 Using waning immunity ............................................................. 70
  1.2.8.2 Multi-variant modeling ............................................................. 72
  1.2.8.3 Advanced vaccination methods .................................................. 73
1.2.9 T9 - Deployment ................................................................. 80
  1.2.9.1 Dask ................................................................. 80
  1.2.9.2 Jupyter/IPython ................................................................. 81
1.2.10 T10 - Tips and tricks ............................................................... 82
  1.2.10.1 Versioning ................................................................. 82
  1.2.10.2 Working with dates ............................................................... 84
  1.2.10.3 Working with dictionaries .................................................... 85
1.2.11 T11 - Advanced features ............................................................ 86
  1.2.11.1 Defining populations with SynthPops ....................................... 86
  1.2.11.2 Defining contact layers .......................................................... 86
  1.2.11.3 Defining dynamic layers ....................................................... 88
  1.2.11.4 Defining custom population properties .................................... 89
  1.2.11.5 Changing Numba options ...................................................... 91
1.3 FAQ ................................................................. 93
  1.3.1 Usage questions ............................................................. 94
    1.3.1.1 What are the system requirements for Covasim? .......................... 94
    1.3.1.2 Can Covasim be run on HPC clusters? ....................................... 94
    1.3.1.3 What method is best for saving simulation objects? ...................... 94
    1.3.1.4 Does Covasim support time-dependent parameters? ....................... 95
    1.3.1.5 How can you introduce new infections into a simulation? ................ 95
    1.3.1.6 How can I find out what happened to a particular individual during the simulation? 95
    1.3.1.7 How do you set custom prognoses parameters (mortality rate, susceptibility etc.)? 95
    1.3.1.8 I want to generate a contact network for <insert location here>. How do I do this? 96
    1.3.1.9 Is it possible to model interacting geographical regions? ................ 96
    1.3.1.10 Is it possible to model births, non-COVID deaths, or migration? ........ 96
    1.3.1.11 I really don’t like Python, can I run Covasim via R? ...................... 96
  1.3.2 Conceptual questions ............................................................ 96
    1.3.2.1 What are the relationships between population size, number of agents, population scaling, and total population? 96
    1.3.2.2 Are test results counted from swab date or result date? .................. 98
    1.3.2.3 Is the underlying model capable of generating oscillations? .............. 98
    1.3.2.4 Why doesn’t anyone start off as infectious? ................................ 98
1.3.2.5 What are the valid states an agent can be in? ........................................... 98

1.3.3 Common problems ................................................................. 99
1.3.3.1 I’m getting different results to someone else, or to what I got previously, with the
same parameters. Why? ............................................................... 99
1.3.3.2 Why doesn’t the webapp accept long durations or large population sizes? .......... 99
1.3.3.3 Why do parallel simulations fail on Windows or in Jupyter notebooks? ................. 99

1.4 What’s new ................................................................. 100
1.4.1 Coming soon ................................................................. 100
1.4.2 Latest versions (3.1.x) ............................................................ 100
1.4.2.1 Version 3.1.1 (2021-12-06) ................................................. 100
1.4.2.2 Version 3.1.0 (2021-12-03) ................................................. 101

1.4.3 Versions 3.0.x (3.0.0 – 3.0.7) ..................................................... 104
1.4.3.1 Version 3.0.7 (2021-06-29) ................................................. 104
1.4.3.2 Version 3.0.6 (2021-06-21) ................................................. 104
1.4.3.3 Version 3.0.5 (2021-05-26) ................................................. 105
1.4.3.4 Version 3.0.4 (2021-05-19) ................................................. 105
1.4.3.5 Version 3.0.3 (2021-05-17) ................................................. 106
1.4.3.6 Version 3.0.2 (2021-04-26) ................................................. 106
1.4.3.7 Version 3.0.1 (2021-04-16) ................................................. 106
1.4.3.8 Version 3.0.0 (2021-04-13) ................................................. 107

1.4.4 Versions 2.x (2.0.0 – 2.1.2) ..................................................... 110
1.4.4.1 Version 2.1.2 (2021-03-31) ................................................. 110
1.4.4.2 Version 2.1.1 (2021-03-29) ................................................. 110
1.4.4.3 Version 2.1.0 (2021-03-23) ................................................. 111
1.4.4.4 Version 2.0.4 (2021-03-19) ................................................. 112
1.4.4.5 Version 2.0.3 (2021-03-11) ................................................. 113
1.4.4.6 Version 2.0.2 (2021-02-01) ................................................. 114
1.4.4.7 Version 2.0.1 (2021-01-31) ................................................. 114
1.4.4.8 Version 2.0.0 (2020-12-05) ................................................. 114

1.4.5 Versions 1.7.x (1.7.0 – 1.7.6) ..................................................... 117
1.4.5.1 Version 1.7.6 (2020-10-23) ................................................. 117
1.4.5.2 Version 1.7.5 (2020-10-13) ................................................. 117
1.4.5.3 Version 1.7.4 (2020-10-02) ................................................. 118
1.4.5.4 Version 1.7.3 (2020-09-30) ................................................. 118
1.4.5.5 Version 1.7.2 (2020-09-24) ................................................. 118
1.4.5.6 Version 1.7.1 (2020-09-23) ................................................. 118
1.4.5.7 Version 1.7.0 (2020-09-20) ................................................. 119

1.4.6 Versions 1.6.x (1.6.0 – 1.6.1) ..................................................... 120
1.4.6.1 Version 1.6.1 (2020-09-13) ................................................. 120
1.4.6.2 Version 1.6.0 (2020-09-08) ................................................. 120

1.4.7 Versions 1.5.x (1.5.0 – 1.5.3) ..................................................... 121
1.4.7.1 Version 1.5.3 (2020-09-01) ................................................. 121
1.4.7.2 Version 1.5.2 (2020-08-18) ................................................. 121
1.4.7.3 Version 1.5.1 (2020-08-17) ................................................. 122
1.4.7.4 Version 1.5.0 (2020-07-01) ................................................. 122

1.4.8 Versions 1.4.x (1.4.0 – 1.4.8) ..................................................... 122
1.4.8.1 Version 1.4.8 (2020-06-11) ................................................. 122
1.4.8.2 Version 1.4.7 (2020-06-02) ................................................. 123
1.4.8.3 Version 1.4.6 (2020-06-01) ................................................. 123
1.4.8.4 Version 1.4.5 (2020-05-31) ................................................. 123
1.4.8.5 Version 1.4.4 (2020-05-31) ................................................. 123
1.4.8.6 Version 1.4.3 (2020-05-30) ................................................. 123
1.4.8.7 Version 1.4.2 (2020-05-30) ................................................. 124
1.4.8.8 Version 1.4.1 (2020-05-29) ................................................. 124
1.5.8 Strain-specific parameters ........................................... 137
1.5.9 Time for disease progression ........................................ 137
1.5.10 Time for disease recovery .......................................... 137
1.5.11 Severity parameters .................................................. 138
1.5.12 Efficacy of protection measures ................................... 138
1.5.13 Events and interventions ............................................ 138
1.5.14 Health system parameters .......................................... 138

1.6 Data scrapers ............................................................. 138
1.6.1 1. Corona Data Scraper ............................................... 139
1.6.2 2. European Centre for Disease Prevention and Control .... 141
1.6.3 3. The COVID Tracking Project .................................... 141
1.6.4 4. Demographic data scraper ....................................... 142

1.7 Glossary ........................................................................ 143
1.8 Code of conduct ............................................................. 145
1.8.1 Our pledge .................................................................. 145
1.8.2 Our standards ............................................................. 145
1.8.3 Our responsibilities ..................................................... 146
1.8.4 Scope ........................................................................ 146
1.8.5 Enforcement ............................................................... 146
1.8.6 Attribution ................................................................. 146

1.9 Contributing to Covasim .................................................. 146
1.9.1 Getting started ............................................................ 147
1.9.2 Contribution types ........................................................ 147
1.9.3 Request new features or report bugs ............................... 147
   1.9.3.1 Open a feature request ........................................... 147
   1.9.3.2 Report a bug ........................................................ 148
1.9.4 Submit a pull request .................................................... 148
1.9.5 Resources .................................................................. 148

1.10 API reference ............................................................... 149
1.10.1 Subpackages ............................................................. 149
   1.10.1.1 covasim.data package ........................................ 149
1.10.2 Submodules .............................................................. 150
   1.10.2.1 covasim.analysis module .................................... 150
   1.10.2.2 covasim.base module .......................................... 160
   1.10.2.3 covasim.defaults module .................................... 169
   1.10.2.4 covasim.immunity module ................................... 170
   1.10.2.5 covasim.interventions module ............................... 170
   1.10.2.6 covasim.misc module .......................................... 183
   1.10.2.7 covasim.parameters module ................................ 188
   1.10.2.8 covasim.people module ....................................... 189
   1.10.2.9 covasim.plotting module ...................................... 192
   1.10.2.10 covasim.population module ................................. 192
   1.10.2.11 covasim.requirements module .............................. 195
   1.10.2.12 covasim.run module .......................................... 196
   1.10.2.13 covasim.settings module .................................... 206
   1.10.2.14 covasim.sim module .......................................... 206
   1.10.2.15 covasim.utils module ....................................... 214
   1.10.2.16 covasim.version module ..................................... 220

Python Module Index ................................................................ 221

Index ...................................................................................... 223
Covasim is a stochastic agent-based simulator, written in Python, for exploring and analyzing the COVID-19 epidemic.

**There’s a lot here, where should I start?**

- Take a quick look at the overview, which provides a general introduction.
- When you’re ready to sink your teeth in, the tutorials will help you get started using Covasim.
- If you’re looking for a specific feature or keyword, you should be able to find it with the search feature (top left).
- Still have questions? Send us an email at covasim@idmod.org. We’re happy to help!
1.1 Covasim overview

1.1.1 About Covasim

Covasim is a stochastic agent-based simulator for performing COVID-19 analyses. These include projections of indicators such as numbers of infections and peak hospital demand. Covasim can also be used to explore the potential impact of different interventions, including social distancing, school closures, testing, contact tracing, quarantine, and vaccination.

The original scientific paper describing Covasim is available at http://paper.covasim.org. The recommended citation is:


Covasim’s immunity module (including vaccines and variants) is described here:


The Covasim webapp is available at http://app.covasim.org, and the repository for it is available here.

Covasim was developed by the Institute for Disease Modeling, with additional contributions from the University of Copenhagen, the Burnet Institute, GitHub, and Microsoft.

Questions or comments can be directed to covasim@idmod.org, or on this project’s GitHub page. Full information about Covasim is provided in the documentation.
1.1.2 Background

Covasim has been used for analyses in over a dozen countries, both to inform policy decisions (including in the US, UK, and Australia), and as part of research studies. Academic papers that have been written using Covasim include:


7. **Schools are not islands: Balancing COVID-19 risk and educational benefits using structural and temporal countermeasures.** Cohen JA, Mistry D, Kerr CC, Klein DJ (under review; posted 2020-09-10). *medRxiv* 2020.09.08.20190942; doi: https://doi.org/10.1101/2020.09.08.20190942.


If you have written a paper or report using Covasim, we’d love to know about it! Please write to us here.

1.1.3 Requirements

Python 3.7-3.9 (64-bit). (Note: Python 2.7 and Python 3.10 are not supported, the latter being due to Numba not supporting Python 3.10 at the time of writing.)

We also recommend, but do not require, installing Covasim in a virtual environment. For more information, see documentation for e.g. Anaconda.
1.1.4 Quick start guide

Install with `pip install covasim`. If everything is working, the following Python commands should bring up a plot:

```python
import covasim as cv
sim = cv.Sim()
sim.run()
sim.plot()
```

1.1.5 Full installation instructions

If you would rather download the source code rather than using the `pip` package, follow these steps:

1. Clone a copy of the repository. If you intend to make changes to the code, we recommend that you fork it first.
2. (Optional) Create and activate a virtual environment.
3. Navigate to the root of the repository and install the Covasim Python package using one of the following options:
   • For normal installation (recommended):
     ```bash
     pip install -e .
     ```
   • To install Covasim and optional dependencies (be aware this may fail since it relies on nonstandard packages):
     ```bash
     pip install -e .[full]
     ```

The module should then be importable via `import covasim as cv`.

1.1.6 Usage examples

There are several examples in the `examples` folder. These can be run as follows:

• `python examples/simple.py`
  This example creates a figure using default parameter values.

• `python examples/run_sim.py`
  This shows a slightly more detailed example, including creating an intervention and saving to disk.

• `python examples/run_scenarios.py`
  This shows a more complex example, including running an intervention scenario, plotting uncertainty, and performing a health systems analysis.

Other examples in that folder are taken from the tutorials.

1.1.7 Module structure

All core model code is located in the `covasim` subfolder; standard usage is `import covasim as cv`. The `data` subfolder is described below.

The model consists of two core classes: the `Person` class (which contains information on health state), and the `Sim` class (which contains methods for running, calculating results, plotting, etc.).
The structure of the `covasim` folder is as follows, roughly in the order in which the modules are imported, building from most fundamental to most complex:

- `version.py`: Version, date, and license information.
- `requirements.py`: A simple module to check that imports succeeded, and turn off features if they didn’t.
- `utils.py`: Functions for choosing random numbers, many based on Numba, plus other helper functions.
- `misc.py`: Miscellaneous helper functions.
- `settings.py`: User-customizable options for Covasim (e.g. default font size).
- `defaults.py`: The default colors, plots, etc. used by Covasim.
- `parameters.py`: Functions for creating the parameters dictionary and loading the input data.
- `plotting.py`: Plotting scripts, including Plotly graphs for the webapp (used in other Covasim classes, and hence defined first).
- `base.py`: The `ParsObj` class, the fundamental class used in Covasim, plus basic methods of the `BaseSim` and `BasePeople` classes, and associated functions.
- `people.py`: The `People` class, for handling updates of state for each person.
- `population.py`: Functions for creating populations of people, including age, contacts, etc.
- `interventions.py`: The `Intervention` class, for adding interventions and dynamically modifying parameters, and classes for each of the specific interventions derived from it.
- `immunity.py`: The `strain` class, and functions for computing waning immunity and neutralizing antibodies.
- `sim.py`: The `Sim` class, which performs most of the heavy lifting: initializing the model, running, and plotting.
- `run.py`: Functions for running simulations (e.g. parallel runs and the `Scenarios` and `MultiSim` classes).
- `analysis.py`: The `Analyzer` class (for performing analyses on the sim while it’s running), the `Fit` class (for calculating the fit between the model and the data), the `TransTree` class, and other classes and functions for analyzing simulations.

The `data` folder within the Covasim package contains loading scripts for the epidemiological data in the root `data` folder, as well as data on age distributions for different countries and household sizes.

### 1.1.8 Other folders

Please see the readme in each subfolder for more information.

#### 1.1.8.1 Bin

This folder contains a command-line interface (CLI) version of Covasim; example usage:

```
covasim --pars "{pop_size:20000, pop_infected:1, n_days:360, rand_seed:1}"
```

Note: the CLI is currently not compatible with Windows. You will need to add this folder to your path to run from other folders.

#### 1.1.8.2 Data

Scripts to automatically scrape data (including demographics and COVID epidemiology data), and the data files themselves (which are not part of the repository).
1.1.8.3 Tutorials

This folder contains Jupyter notebooks for nine tutorials that walk you through using Covasim, from absolute basics to advanced topics such as calibration and creating custom populations.

1.1.8.4 Examples

This folder contains demonstrations of simple Covasim usage, with most examples taken from the tutorials.

Cruise ship

An early application of Covasim to the Diamond Princess cruise ship.

Calibration

Examples of how to calibrate simulations, including Optuna (also covered in the tutorial) and Weights and Biases.

1.1.8.5 Licenses

Licensing information and legal notices.

1.1.8.6 Tests

Integration, development, and unit tests. While not (yet) beautifully curated, these folders contain many usage examples. See the tests README for more information.

1.1.9 Disclaimer

The code in this repository was developed by IDM to support our research in disease transmission and managing epidemics. We’ve made it publicly available under the Creative Commons Attribution-ShareAlike 4.0 International License to provide others with a better understanding of our research and an opportunity to build upon it for their own work. We make no representations that the code works as intended or that we will provide support, address issues that are found, or accept pull requests. You are welcome to create your own fork and modify the code to suit your own modeling needs as contemplated under the Creative Commons Attribution-ShareAlike 4.0 International License. See the contributing and code of conduct READMEs for more information.

1.2 Tutorials

These tutorials walk through how to use Covasim. If you want to explore them interactively, you can run them on Binder via http://tutorials.covasim.org. To run locally, start a Jupyter environment in this folder (docs/tutorials). You can use either jupyter lab or jupyter notebook to run these tutorials.
1.2.1 T1 - Getting started

Installing and getting started with Covasim is quite simple.

To install, just type `pip install covasim`. If it worked, you should be able to import Covasim with `import covasim as cv`.

The basic design philosophy of Covasim is: common tasks should be simple. For example:

- Defining parameters
- Running a simulation
- Plotting results

This tutorial walks you through how to do these things.

Click here to open an interactive version of this notebook.

1.2.1.1 Hello world

To create, run, and plot a sim with default options is just:

```
[1]: import covasim as cv

sim = cv.Sim()
sim.run()
fig = sim.plot()
```

Covasim 3.1.1 (2021-12-06) -- © 2021 by IDM

Initializing sim with 20000 people for 60 days

Running 2020-03-01 ( 0/60) (0.21 s) ------------------------------- 2%
Running 2020-03-11 (10/60) (0.25 s) ------------------------------- 18%
Running 2020-03-21 (20/60) (0.29 s) ------------------------------- 34%
Running 2020-03-31 (30/60) (0.33 s) ------------------------------ 51%
Running 2020-04-10 (40/60) (0.38 s) ------------------------------ 67%
Running 2020-04-20 (50/60) (0.44 s) ----------------------------- 84%
Running 2020-04-30 (60/60) (0.52 s) ----------------------------- 100%

Simulation summary:

12,730 cumulative infections
596 cumulative reinfections
10,182 cumulative infectious
6,395 cumulative symptomatic cases
386 cumulative severe cases
114 cumulative critical cases
5,159 cumulative recoveries
14 cumulative deaths
0 cumulative tests
0 cumulative diagnoses
0 cumulative known deaths
0 cumulative quarantined people
0 cumulative vaccine doses
0 cumulative vaccinated people

Chapter 1. Full contents
### 1.2.1.2 Defining parameters and running simulations

Parameters are defined as a dictionary. The most common parameters to modify are the population size, the initial number of people infected, and the start and end dates of the simulation. We can define those as:

```python
[2]: pars = dict(
    pop_size = 50e3,
    pop_infected = 100,
    start_day = '2020-04-01',
    end_day = '2020-06-01',
)
```

Running a simulation is pretty easy. In fact, running a sim with the parameters we defined above is just:

```python
[3]: sim = cv.Sim(pars)
    sim.run()
```

```
Initializing sim with 50000 people for 61 days
Running 2020-04-01 ( 0/61) (0.46 s) --........................................ 2%
Running 2020-04-11 (10/61) (0.53 s) ...................................... 18%
```

(continues on next page)
This will generate a results dictionary `sim.results`. For example, the number of new infections per day is `sim.results['new_infections']`.

Rather than creating a parameter dictionary, any valid parameter can also be passed to the `sim` directly. For example, exactly equivalent to the above is:

```
[4]: sim = cv.Sim(pop_size=50e3, pop_infected=100, start_day='2020-04-01', end_day='2020-06-01')
sim.run()
```
You can mix and match too – pass in a parameter dictionary with default options, and then include other parameters as keywords (including overrides; keyword arguments take precedence). For example:

```python
sim = cv.Sim(pars, pop_infected=10) # Use parameters defined above, except start with
                                          10 infected people
sim.run()
```

Simulation summary:
- 10,392 cumulative infections
- 128 cumulative reinfections
- 6,710 cumulative infectious
- 4,185 cumulative symptomatic cases
- 195 cumulative severe cases
- 51 cumulative critical cases
- 2,473 cumulative recoveries
- 7 cumulative deaths
- 0 cumulative tests
- 0 cumulative diagnoses
- 0 cumulative known deaths
- 0 cumulative quarantined people
- 0 cumulative vaccine doses
- 0 cumulative vaccinated people

1.2.1.3 Plotting results

As you saw above, plotting the results of a simulation is rather easy too:

```python
fig = sim.plot()
```
### 1.2.1.4 Full usage example

Many of the details of this example will be explained in later tutorials, but to give you a taste, here’s an example of how you would run two simulations to determine the impact of a custom intervention aimed at protecting the elderly.

```python
# Custom intervention -- see Tutorial 5
import covasim as cv

def protect_elderly(sim):
    if sim.t == sim.day('2020-04-01'):
        elderly = sim.people.age>70
        sim.people.rel_sus[elderly] = 0.0

pars = dict(
    pop_type = 'hybrid', # Use a more realistic population model
    location = 'japan', # Use population characteristics for Japan
    pop_size = 50e3, # Have 50,000 people total in the population
    pop_infected = 100, # Start with 100 infected people
    n_days = 90, # Run the simulation for 90 days
    verbose = 0, # Do not print any output
)
```

(continues on next page)
# Running with multisims -- see Tutorial 3

```python
s1 = cv.Sim(pars, label='Default')
s2 = cv.Sim(pars, interventions=protect_elderly, label='Protect the elderly')
msim = cv.MultiSim([s1, s2])
msim.run()
fig = msim.plot(to_plot=['cum_deaths', 'cum_infections'])
```

## 1.2.2 T2 - Plotting, printing, and saving

This tutorial provides a brief overview of options for plotting results, printing objects, and saving results.

Click [here](#) to open an interactive version of this notebook.
1.2.2.1 Global plotting configuration

Covasim allows the user to set various options that apply to all plots. You can change the font size, default DPI, whether plots should be shown by default, etc. (for the full list, see `cv.options.help()`). For example, we might want higher resolution, to turn off automatic figure display, close figures after they’re rendered, and to turn off the messages that print when a simulation is running. We can do this using built-in defaults for Jupyter notebooks (and then run a sim) with:

```python
[1]: import covasim as cv
    cv.options.set('jupyter', verbose=0) # Standard options for Jupyter notebook
    sim = cv.Sim()
    sim.run()
```

Covasim 3.1.1 (2021-12-06) -- © 2021 by IDM

```
[1]: Sim(<no label>; 2020-03-01 to 2020-04-30; pop: 20000 random; epi: 12730, 14)
```

1.2.2.2 Printing objects

There are three levels of detail available for most objects (sims, multisims, scenarios, and people). The shortest is `brief()`:

```python
[2]: sim.brief()
    Sim(<no label>; 2020-03-01 to 2020-04-30; pop: 20000 random; epi: 12730, 14)
```

You can get more detail with `summarize()`:

```python
[3]: sim.summarize()
```

```
Simulation summary:
    12,730 cumulative infections
    596 cumulative reinfections
    10,182 cumulative infectious
    6,395 cumulative symptomatic cases
    386 cumulative severe cases
    114 cumulative critical cases
    5,159 cumulative recoveries
    14 cumulative deaths
    0 cumulative tests
    0 cumulative diagnoses
    0 cumulative known deaths
    0 cumulative quarantined people
    0 cumulative vaccine doses
    0 cumulative vaccinated people
```

Finally, to show the full object, including all methods and attributes, use `disp()`:

```python
[4]: sim.disp()
    <covasim.sim.Sim at 0x7f5a6a35bc70>
```

```
Methods:
    _brief() finalize() make_transtree()
    _disp() finalize_analyz... plot()
```

(continues on next page)
_get_ia() finalize_intervention plot_result()
brief() get_analyzer() rescale()
calibrate() get_analyzers() reset_layer_pars()
compute_doubling() get_intervention() result_keys()
compute_fit() get_intervention() run()
compute_gen_time() init_analyzers() save()
compute_r_eff() init_immunity() set_metadata()
compute_results() init_intervention() set_seed()
compute_states() init_people() shrink()
compute_summary() init_results() step()
compute_yield() init_variants() summarize()
copy() initialize() to_df()
date() layer_keys() to_excel()
day() load() to_json()
disp() load_data() update_pars()
export_pars() load_population() validate_layer_
export_results() make_age_histogram validate_pars()

Properties:
datevec npts tvec
n scaled_pop_size

__default_ver: None
__legacy_trans: False
__orig_pars: {'interventions': [], 'analyzers': []}
  complete: True
    created: datetime.datetime(2021, 12, 6, 6, 32, 10, 200109)
    data: None
    datafile: None
    git_info: {'covasim': {'version': '3.1.1',
    'branch': 'Branch N/A',
    'hash': 'H [...]}
  initialized: True
    label: None
  load_pop: False
    pars: {'pop_size': 20000, 'pop_infected': 20, 'pop_type':
     'random', 'loca [...]}
    people: People(n=20000; layers: a)
    popdict: None
    popfile: None
  rescale_vec: array([1., 1., 1., ..., 1., 1., 1., 1., 1., 1., 1., ...]
  results: #0. cum_infections:
    <covasim.base.Result at 0x7f5a3eac1310>
      ------------
  results_ready: True
  save_pop: False
    simfile: 'covasim_2021-Dec-06_06.32.10.sim'
    summary: #0. cum_infections: 12730.0
    #1. cum_reinfections: 596.0
    #2. [...]}
  t: 60
  version: '3.1.1'
1.2.2.3 Plotting options

While a sim can be plotted using default settings simply by `sim.plot()`, this is just a small fraction of what’s available. First, note that results can be plotted directly using e.g. Matplotlib. You can see what quantities are available for plotting with `sim.results.keys()` (remember, it’s just a dict). A simple example of plotting using Matplotlib is:

```python
import pylab as pl  # Shortcut for import matplotlib.pyplot as plt
pl.plot(sim.results['date'], sim.results['new_infections'])
```

However, as you can see, this isn’t ideal since the default formatting is...not great. (Also, note that each result is a Result object, not a simple Numpy array; like a pandas dataframe, you can get the array of values directly via e.g. `sim.results.new_infections.values`.)

An alternative, if you only want to plot a single result, such as new infections, is to use the `plot_result()` method:

```python
sim.plot_result('new_infections')
```
You can also select one or more quantities to plot with the `to_plot` argument, e.g.

```python
sim.plot(to_plot=['new_infections', 'cum_infections'])
```
Another useful option is to plot an overview of everything in a simulation. We can do this with the `to_plot='overview'` argument. It’s quite a lot of information so we might also want to make a larger figure for it, which we can do by passing additional arguments via the `fig_args` argument (which is passed to `plt.figure()`).

```
[8]: sim.plot(to_plot='overview', fig_args=dict(figsize=(30,15)))
```
While we can save this figure using Matplotlib’s built-in `savefig()`, if we use Covasim’s `cv.savefig()` we get a couple advantages:

```python
[9]: cv.savefig('my-fig.png')
```

```python
[9]: 'my-fig.png'
```

<Figure size 640x480 with 0 Axes>

First, it saves the figure at higher resolution by default (which you can adjust with the `dpi` argument). But second, it stores information about the code that was used to generate the figure as metadata, which can be loaded later. Made an awesome plot but can’t remember even what script you ran to generate it, much less what version of the code? You’ll never have to worry about that again.

```python
[10]: cv.get_png_metadata('my-fig.png')
```

<table>
<thead>
<tr>
<th>Covasim version:</th>
<th>3.1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Covasim branch:</td>
<td>Branch N/A</td>
</tr>
<tr>
<td>Covasim hash:</td>
<td>Hash N/A</td>
</tr>
<tr>
<td>Covasim date:</td>
<td>Date N/A</td>
</tr>
<tr>
<td>Covasim caller branch:</td>
<td>Branch N/A</td>
</tr>
<tr>
<td>Covasim caller hash:</td>
<td>Hash N/A</td>
</tr>
<tr>
<td>Covasim caller date:</td>
<td>Date N/A</td>
</tr>
<tr>
<td>Covasim caller filename:</td>
<td>/home/docs/checkouts/readthedocs.org/user_builds/institute--for-disease-modeling-covasim/envs/latest/lib/python3.8/site-packages/covasim/misc.py</td>
</tr>
<tr>
<td>Covasim current time:</td>
<td>2021-Dec-06 06:32:17</td>
</tr>
<tr>
<td>Covasim calling file:</td>
<td>/tmp/ipykernel_1116/3042824350.py, line 1</td>
</tr>
</tbody>
</table>

### 1.2.2.4 Saving options

Saving sims is also pretty simple. The simplest way to save is simply

```python
[11]: sim.save('my-awesome-sim.sim')
```
Technically, this saves as a gzipped pickle file (via `sc.saveobj()` using the Sciris library). By default this does not save the people in the sim since they are very large (and since, if the random seed is saved, they can usually be regenerated). If you want to save the people as well, you can use the `keep_people` argument. For example, here’s what it would look like to create a sim, run it halfway, save it, load it, change the overall transmissibility (beta), and finish running it:

```python
[12]: sim_orig = cv.Sim(start_day='2020-04-01', end_day='2020-06-01', label='Load & save example')
sim_orig.run(until='2020-05-01')
sim_orig.save('my-half-finished-sim.sim')  # Note: Covasim always saves the people if the sim isn't finished running yet

sim = cv.load('my-half-finished-sim.sim')
sim['beta'] *= 0.3
sim.run()
sim.plot(to_plot=['new_infections', 'n_infectious', 'cum_infections'])
```

Aside from saving the entire simulation, there are other export options available. You can export the results and parameters to a JSON file (using `sim.to_json()`), but probably the most useful is to export the results to an Excel...
workbook, where they can easily be stored and processed with e.g. Pandas:

```python
import pandas as pd

sim.to_excel('my-sim.xlsx')
df = pd.read_excel('my-sim.xlsx')
print(df)
```

Object saved to /home/docs/checkouts/readthedocs.org/user_builds/institute-for-disease-modeling-covasim/checkouts/latest/docs/tutorials/my-sim.xlsx.

<table>
<thead>
<tr>
<th>date</th>
<th>t</th>
<th>cum_infections</th>
<th>cum_reinfections</th>
<th>cum_infectious</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020-04-01</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2020-04-02</td>
<td>1</td>
<td>21</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2020-04-03</td>
<td>2</td>
<td>21</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2020-04-04</td>
<td>3</td>
<td>23</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>2020-04-05</td>
<td>4</td>
<td>28</td>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>date</th>
<th>cum_symptomatic</th>
<th>cum_severe</th>
<th>cum_critical</th>
<th>cum_recoveries</th>
<th>cum_deaths</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020-04-01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2020-04-02</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2020-04-03</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2020-04-04</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>date</th>
<th>prevalence</th>
<th>incidence</th>
<th>r_eff</th>
<th>doubling_time</th>
<th>test_yield</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020-04-01</td>
<td>0.001000</td>
<td>0.000000</td>
<td>2.766969</td>
<td>NaN</td>
<td>0</td>
</tr>
<tr>
<td>2020-04-02</td>
<td>0.001050</td>
<td>0.000050</td>
<td>3.018511</td>
<td>NaN</td>
<td>0</td>
</tr>
<tr>
<td>2020-04-03</td>
<td>0.001050</td>
<td>0.000000</td>
<td>2.602426</td>
<td>NaN</td>
<td>0</td>
</tr>
<tr>
<td>2020-04-04</td>
<td>0.001150</td>
<td>0.000100</td>
<td>2.537009</td>
<td>14.878453</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>date</th>
<th>rel_test_yield</th>
<th>frac_vaccinated</th>
<th>pop_nabs</th>
<th>pop_protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020-04-01</td>
<td>0.000258</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>2020-04-02</td>
<td>0.000518</td>
<td>0.000342</td>
<td>0.000468</td>
<td>0.000468</td>
</tr>
<tr>
<td>2020-04-03</td>
<td>0.000778</td>
<td>0.000543</td>
<td>0.000468</td>
<td>0.000468</td>
</tr>
</tbody>
</table>

(continues on next page)
1.2.3 T3 - Running scenarios

While running individual sims can be interesting for simple explorations, at some point it will almost always be necessary to run a large number of simulations simultaneously - to explore different scenarios, to perform calibration, or simply to get uncertainty bounds on a single projection. This tutorial explains how to do that.

Click here to open an interactive version of this notebook.

1.2.3.1 Running with MultiSims

The most common way to run multiple simulations is with the MultiSim object. As the name suggests, this is a relatively simple container for a number of sims. However, it contains powerful methods for plotting, statistics, and running all the sims in parallel.

Running one sim with uncertainty

Making and running a multisim based on a single sim is pretty easy:

```
[1]:  import covasim as cv
    cv.options.set('jupyter', verbose=0)
    sim = cv.Sim()
    msim = cv.MultiSim(sim)
    msim.run(n_runs=5)
    msim.plot()
```
If you run a multisim with a single sim input as above, it will change the random seed for each sim, which is what leads to the variability you see.

By default, the multisim simply plots each simulation. These simulations are stored in the `sims` attribute, which is just a simple list of sims:

```python
[2]: for sim in msim.sims:
    sim.brief()
```

```bash
Sim("Sim 0"; 2020-03-01 to 2020-04-30; pop: 20000 random; epi: 12730, 14)
Sim("Sim 1"; 2020-03-01 to 2020-04-30; pop: 20000 random; epi: 13582, 21)
Sim("Sim 2"; 2020-03-01 to 2020-04-30; pop: 20000 random; epi: 12989, 12)
Sim("Sim 3"; 2020-03-01 to 2020-04-30; pop: 20000 random; epi: 8467, 9)
Sim("Sim 4"; 2020-03-01 to 2020-04-30; pop: 20000 random; epi: 8861, 4)
```

However, often (especially when you run the same parameters with different random seeds), you don’t care about the individual sims, you want to see the statistics for the sims. You can calculate either the mean or the median of the results across all the sims as follows:

```python
[3]: msim.mean()
msim.plot_result('new_infections')
```
[3]:

```
msim.median()
msim.plot_result('new_infections')
```

Chapter 1. Full contents
You can see these are similar, but slightly different. You can also treat each of the individual sims as part of a larger single sim, and “combine” the results into one sim:

```
[5]: msim.combine()
msim.plot_result('new_infections')
```
Note how now there is no uncertainty and the total number of infections is 5x higher than in the previous plots, since we just added 5 different sims together.

Each of these operations modifies the msim.base_sim object, and does not affect the actual list of stored sims, which is why you can go back and forth between them.

**Running different sims**

Often you don’t want to run the same sim with different seeds, but instead want to run a set of different sims. That’s also very easy – for example, here’s how you would do a sweep across beta values:

```python
[6]: import numpy as np

betas = np.linspace(0.010, 0.020, 5) # Sweep beta from 0.01 to 0.02 with 5 values
sims = []
for beta in betas:
    sim = cv.Sim(beta=beta, label=f'Beta = {beta}')
    sims.append(sim)
msim = cv.MultiSim(sims)
msim.run()
m sim.plot_result('cum_infections')
```
As you would expect, higher beta values have more infections.

Finally, note that you can use multisims to do very compact scenario explorations – here we are using the command `cv.parallel()`, which is an alias for `cv.MultiSim().run()`:

```python
[7]:
    def protect_elderly(sim):
        if sim.t == sim.day('2021-04-01'):
            elderly = sim.people.age>70
            sim.people.rel_sus[elderly] = 0.0

    pars = {'start_day':'2021-03-01', 'n_days':120}
    s1 = cv.Sim(pars, label='Default')
    s2 = cv.Sim(pars, label='Protect the elderly', interventions=protect_elderly)
    cv.parallel(s1, s2).plot(to_plot=['cum_deaths', 'cum_infections'])
```
**Gotcha:** Because `multiprocess` pickles the sims when running them, `sims[0]` (before being run by the multisim) and `msim.sims[0]` are not the same object. After calling `msim.run()`, always use sims from the multisim object, not from before. In contrast, if you don’t run the multisim (e.g. if you make a multisim from already-run sims), then `sims[0]` and `msim.sims[0]` are indeed exactly the same object.

**Advanced usage**

Finally, you can also merge or split different multisims together. Here’s an example that’s similar to before, except it shows how to run a multisim of different seeds for the same beta value, but then merge multisims for different beta values together into one multisim to rule them all:

```python
n_sims = 3
betas = [0.012, 0.016, 0.018]
msims = []
for beta in betas:
    sims = []
```
```python
for s in range(n_sims):
    sim = cv.Sim(pop_size=10e3, beta=beta, rand_seed=s, label=f'Beta = {beta}')[1]
    msim = cv.MultiSim([sim])
    msim.run()
    msim.mean()
    msims.append(msim)

merged = cv.MultiSim.merge(msims, base=True)
merged.plot(color_by_sim=True)
```

As you can see, running this way lets you run not just different values, but run different values with uncertainty. Which brings us to...

### 1.2.3.2 Running with Scenarios

Most of the time, you’ll want to run with multisims since they give you the most flexibility. However, in certain cases, Scenario objects let you achieve the same thing more simply. Unlike MultiSims, which are completely agnostic about what sims you include, scenarios always start from the same base sim. They then modify the parameters as you specify, and finally add uncertainty if desired. For example, this shows how you’d use scenarios to run the example...
similar to the one above.

```python
[9]: # Set base parameters -- these will be shared across all scenarios
    basepars = {'pop_size':10e3}
    
    # Configure the settings for each scenario
    scenarios = {
        'baseline': {
            'name':'Baseline',
            'pars': {}
        },
        'high_beta': {
            'name':'High beta (0.020)',
            'pars': {
                'beta': 0.020,
            }
        },
        'low_beta': {
            'name':'Low beta (0.012)',
            'pars': {
                'beta': 0.012,
            }
        }
    }
    
    # Run and plot the scenarios
    scens = cv.Scenarios(basepars=basepars, scenarios=scenarios)
    scens.run()
    scens.plot()
```
1.2.4 T4 - People and data

Modeling without data is like riding a bicycle while blindfolded – rarely dull, but often you don’t get to where you want to go. This tutorial shows how to use data with Covasim, and gives a brief introduction to people, populations, and contact layers.

Click here to open an interactive version of this notebook.

1.2.4.1 Data requirements

Covasim is intentionally designed to be flexible with data requirements, acknowledging that some settings have large amounts of data, while others have very little. There are, however, some minimum data requirements if a real-world context is being modeled (as opposed to a theoretical exploration). These are:

- **Population size:** To interpret results correctly, you need to know what the correct denominator is. Typically this is the city or country you’re modeling, but sometimes it’s not as simple as it may seem, especially early in an epidemic when cases tend to be concentrated near the source of the outbreak.
• **Deaths per day:** Often considered to be the most reliable source of data, since deaths tend to be under-reported less than infections; however, significant under-reporting can still occur, so always treat reported deaths as the lower limit of the true value.

• **Diagnosed cases per day:** Diagnosed cases are the most responsive measure of the epidemic conditions; however, they are dependent on both the number of infections and the testing rate. In settings with low or highly variable rates of testing, they may not be especially reliable.

In addition to these essential data requirements, several other pieces of data are useful to have. These are: - **Seroprevalence:** Although seroprevalence rates wane over time, they are generally a more reliable indicator of total infections than diagnosed cases. - **Tests per day:** The number of tests can be used directly in the `test_num` intervention (see tutorial 5), and is very useful for interpreting diagnoses data. - **Vaccinations per day:** In locations with high vaccination rates, understanding when the vaccines were rolled out (and with which vaccines) is important for estimating current immunity levels. - **Policy interventions:** While changes in transmission can often be seen directly in the data on cases and deaths, at minimum it is a useful sanity check to see if these changes line up with changes in policy such as mobility restrictions or mask mandates.

### 1.2.4.2 Demographic data

Covasim includes pre-downloaded data on country (and US state) age distributions and household size distributions. As we saw in Tutorial 1, you can load these data simply by using the `location` parameter. You can show a list of all available locations with `cv.data.show_locations()`. The data themselves are simply a set of dictionaries, and these can be modified directly; for example, to add a custom age distribution for Johannesburg would look like this:

```python
# Note data format and key names!
joburg_pop = {
    '0-9':  286620,
    '10-19': 277020,
    '20-29': 212889,
    '30-39': 161329,
    '40-49': 104399,
    '50-59':  51716,
    '60-69':  36524,
    '70-79':  22581,
    '80+':   7086,
}

cv.data.country_age_data.data['Johannesburg'] = joburg_pop
```

You can then use these data via `sim = cv.Sim(location='Johannesburg')`.

### 1.2.4.3 Epidemiological data scrapers

Covasim includes a script to automatically download time series data on diagnoses, deaths, and other epidemiological information from several major sources of COVID-19 data. These include the Corona Data Scraper, the European Centre for Disease Prevention and Control, and the COVID Tracking Project. These scrapers provide data for a large number of locations (over 4000 at the time of writing), including the US down to the county level and many other countries down to the district level. The data they download is already in the correct format for Covasim.

**Note:** These data sources are frequently updated, and some may no longer work. Please contact us at info@covasim.org if you're having trouble.
1.2.4.4 Data input and formats

The correct input data format for Covasim looks like this:

```python
import pandas as pd
df = pd.read_csv('example_data.csv')
print(df)
```

<table>
<thead>
<tr>
<th>date</th>
<th>new_diagnoses</th>
<th>new_tests</th>
<th>new_deaths</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020-03-01</td>
<td>1</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-02</td>
<td>3</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-03</td>
<td>2</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-04</td>
<td>8</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-05</td>
<td>20</td>
<td>38</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-06</td>
<td>9</td>
<td>61</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-07</td>
<td>6</td>
<td>43</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-08</td>
<td>13</td>
<td>98</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-09</td>
<td>6</td>
<td>93</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-10</td>
<td>25</td>
<td>170</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-11</td>
<td>28</td>
<td>368</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-12</td>
<td>27</td>
<td>437</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-13</td>
<td>22</td>
<td>291</td>
<td>2</td>
</tr>
<tr>
<td>2020-03-14</td>
<td>43</td>
<td>328</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-15</td>
<td>76</td>
<td>1147</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-16</td>
<td>65</td>
<td>1438</td>
<td>1</td>
</tr>
<tr>
<td>2020-03-17</td>
<td>88</td>
<td>1209</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-18</td>
<td>86</td>
<td>1269</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-19</td>
<td>115</td>
<td>1195</td>
<td>1</td>
</tr>
<tr>
<td>2020-03-20</td>
<td>51</td>
<td>529</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-21</td>
<td>55</td>
<td>482</td>
<td>3</td>
</tr>
<tr>
<td>2020-03-22</td>
<td>95</td>
<td>1106</td>
<td>2</td>
</tr>
<tr>
<td>2020-03-23</td>
<td>74</td>
<td>471</td>
<td>1</td>
</tr>
<tr>
<td>2020-03-24</td>
<td>63</td>
<td>438</td>
<td>4</td>
</tr>
<tr>
<td>2020-03-25</td>
<td>178</td>
<td>1111</td>
<td>2</td>
</tr>
<tr>
<td>2020-03-26</td>
<td>83</td>
<td>621</td>
<td>1</td>
</tr>
<tr>
<td>2020-03-27</td>
<td>140</td>
<td>1059</td>
<td>2</td>
</tr>
<tr>
<td>2020-03-28</td>
<td>137</td>
<td>951</td>
<td>1</td>
</tr>
<tr>
<td>2020-03-29</td>
<td>150</td>
<td>964</td>
<td>0</td>
</tr>
<tr>
<td>2020-03-30</td>
<td>144</td>
<td>1058</td>
<td>1</td>
</tr>
<tr>
<td>2020-04-03</td>
<td>145</td>
<td>1058</td>
<td>1</td>
</tr>
<tr>
<td>2020-04-11</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

The data can be CSV, Excel, or JSON format. There must be a column named `date` (not “Date” or “day” or anything else). Otherwise, each column label must start with `new_` (daily) or `cum_` (cumulative) and then be followed by any of: `tests`, `diagnoses`, `deaths`, `severe` (corresponding to hospitalizations), or `critical` (corresponding to ICU admissions). While other columns can be included and will be loaded, they won’t be parsed by Covasim. Note that if you enter a `new_` (daily) column, Covasim will automatically calculate a `cum_` (cumulative) column for you.

**Note:** Sometimes date information fails to be read properly, especially when loading from Excel files via pandas. If you encounter this problem, see Tutorial 8 for help on fixing this.

This example shows how a simulation can load in the data, and how it automatically plots it. (We’ll cover interventions properly in the next tutorial.)

```python
import covasim as cv
cv.options.set('jupyter', verbose=0)
```
As you can see, this is not a great fit to data – but we’ll come to calibration in Tutorial 7.

1.2.4.5 People and contact network layers

Agents in Covasim are contained in an object called People, which contains all of the agents’ properties, as well as methods for changing them from one state to another (e.g., from susceptible to infected).

Agents interact with each other via one or more contact layers. You can think of each agent as a node in a mathematical graph, and each connection as an edge. By default, Covasim creates a single random contact network where each
agent is connected to 20 other agents, completely at random. However, this is not a very realistic representation of households, workplaces, schools, etc.

For greater realism, Covasim also comes with a “hybrid” population option, which provides a more realism while still being fast to generate. (It’s called “hybrid” because it’s a combination of the random network and the SynthPops network, described in Tutorial 9, which is much more realistic but requires a lot of data and is computationally intensive.) The hybrid option provides four contact layers: households ‘h’, schools ‘s’, workplaces ‘w’, and community interactions ‘c’. Each layer is defined by (a) which agents are connected to which other agents, and (b) the weight of each connection (i.e., transmission probability). Specifically:

- Households are small clusters, usually 2-5 people (depending on country), consisting of a wide variety of ages; this setting has high transmission probability
- Schools are split into classrooms, usually about 20 students each, of students aged 6–22; this setting has medium transmission probability
- Workplaces are clusters of about 5-20 people, aged (approximately) 22-65; this setting has medium transmission probability
- Community contacts (representing parks, restaurants, transport, places of worship, etc.) are estimated as 20 random contacts per day; this setting has low transmission probability

Note that for most countries, you can load default data (age distribution and household size, both from the UN) by using the location keyword when creating a sim. For example, to create a realistic (i.e. hybrid) population 10,000 people for Bangladesh and plot the results, you would do:

```
[3]: pars = dict(
    pop_size = 10_000, # Alternate way of writing 10000
    pop_type = 'hybrid',
    location = 'Bangladesh', # Case insensitive
)

sim = cv.Sim(pars)
sim.initialize() # Create people
fig = sim.people.plot() # Show statistics of the people
```
1.2.5 T5 - Using interventions

Interventions are one of the most critical parts of Covasim. This tutorial shows how to implement standard interventions, as well as how to define your own custom interventions.

Click here to open an interactive version of this notebook.

1.2.5.1 Change beta and clip edges

The most basic intervention in Covasim is “change beta”: this changes overall transmission rates on the specified day(s), no questions asked. For example:

```python
[1]: import covasim as cv
    cv.options.set('jupyter', verbose=0)

    # Define baseline parameters and sim
    pars = dict(
        start_day = '2020-03-01',
        end_day = '2020-06-01',
        pop_type = 'hybrid',
    )
    orig_sim = cv.Sim(pars, label='Baseline')

    # Define sim with change_beta
    cb = cv.change_beta(days=['2020-04-15', '2020-05-01', '2020-05-15'], changes=[0.2, 1. 
        → 5, 0.7])
    sim = cv.Sim(pars, interventions=cb, label='With beta changes')

    # Run and plot
    msim = cv.parallel(orig_sim, sim)
    msim.plot()
```

Covasim 3.1.1 (2021-12-06) -- © 2021 by IDM
A few things to note here:

- By default, interventions are shown with vertical dashed lines. You can turn this off by passing do_plot=False to the intervention.
- Look at the new infections plot (middle) for the clearest illustration of the impact the intervention has.
- Note that like other “parameters”, you can either pass interventions to the sim directly or as part of the pars dictionary; the examples below illustrate these options.
- All interventions handle dates automatically, so days can be specified as strings, integer numbers of days from the start of the sim, or datetime objects. You can even mix and match (e.g. days=[10, '2020-05-05']), although this is not recommended for obvious reasons!

If your simulation has different layers (discussed in Tutorial 1), you can also define the beta change to only apply to a single layer. For example, an intervention that closed 90% of schools (‘s’) on September 1st would be written as:

```python
close_schools = cv.change_beta(days='2020-09-01', changes=0.1, layers='s')
```

The “clip edges” intervention works similarly to the “change beta” intervention, with two important distinctions:

1. Whereas beta changes affect the per-contact transmission probability, clip edges change the number of contacts.
   For example, in the close_schools example above, each school-aged child has an unchanged number of contacts (about 20), but 10 times lower probability of infection per contact (about 0.1% per contact per day). If
clip edges were used instead, then each child would have unchanged probability of infection per contact (about 1% per contact per day), but 10 times fewer contacts (about 2).

2. While beta changes can be any value (greater or less than 1), as the name implies, clip edges can only clip edges, not create new ones. Edges that are clipped are saved and can later be restored, but you can never have a change of greater than 1.0 with clip edges.

Overall, it’s best to use \texttt{change\_beta} for interventions that reduce per-contact risk (e.g., masks), and \texttt{clip\_edges} for interventions that reduce the number of contacts per person. However, in practice, it tends to make relatively little difference to the simulation results. If in doubt, use \texttt{change\_beta}.

You can use \texttt{change\_beta} to do some gnarly things, like create sinusoidally varying infection probability:

```python
[2]:
import numpy as np
import sciris as sc
import covasim as cv

pars = sc.objdict(
    beta = 0.008,
    n_agents = 50e3,
    n_days = 180,
    verbose = 0,
)

beta_days = np.arange(pars.n_days)
beta_vals = np.cos(2*np.pi*beta_days/90)**2+0.5
beta = cv.change_beta(beta_days, beta_vals, do_plot=False)

s1 = cv.Sim(pars, label='Normal')
s2 = cv.Sim(pars, interventions=beta, label='Waves')

msim = cv.parallel(s1, s2)
msim.plot(['r_eff'])
```
### 1.2.5.2 Testing interventions

There are two types of testing interventions in Covasim:

1. **test_num** performs the specified number of tests per day;

2. **test_prob** performs testing with the specified probability.

Typically, if you have actual data on number of tests, you would use **test_num**; otherwise, you would usually use **test_prob**. For **test_num**, other than the number of tests performed, the main parameter is **symp_prob**, which determines how much more likely someone with (true positive) COVID symptoms is to test than a person who is asymptomatic or uninfected. Higher values of **symp_prob** give more diagnoses per number of tests, since symptomatic people preferentially test. The default value is 100, meaning that on any given day, a person experiencing COVID symptoms is 100 times more likely to test than someone who isn’t.

Likewise, **test_prob** also lets you specify the (daily) probability of symptomatic vs. asymptomatic testing. Unlike **test_num**, however, the total number of tests is determined by these probabilities. For example, since symptomatic testing probabilities are usually much higher than asymptomatic, if the epidemic is growing, more people will be symptomatic and thus go to be tested.
Note that test probabilities for `test_prob` are daily probabilities, not total. If you set `symp_prob = 0.3`, this means a person has a 30% chance of testing per day of symptoms, meaning that on average they will go to be tested after 2-3 days. Given that symptoms last for up to 8 days, the total probability that they will test at some point during their illness is much higher than the daily probability.

The example below illustrates (a) `test_num` with data, (b) `test_num` with a fixed number of tests per day, and (c) `test_prob`.

```python
[3]: import covasim as cv

# Define the testing interventions
tn_data = cv.test_num('data')
tn_fixed = cv.test_num(daily_tests=500, start_day='2020-03-01')
tp = cv.test_prob(symp_prob=0.2, asymp_prob=0.001, start_day='2020-03-01')

# Define the default parameters
pars = dict(
    pop_size = 50e3,
    pop_infected = 100,
    start_day = '2020-02-01',
    end_day = '2020-03-30',
)

# Define the simulations
sim1 = cv.Sim(pars, datafile='example_data.csv', interventions=tn_data, label='Number of tests from data')
sim2 = cv.Sim(pars, interventions=tn_fixed, label='Constant daily number of tests')
sim3 = cv.Sim(pars, interventions=tp, label='Probability-based testing')

# Run and plot results
msim = cv.parallel(sim1, sim2, sim3)
msim.plot(to_plot=['new_infections', 'new_tests', 'new_diagnoses', 'cum_diagnoses'])
1.2.5.3 Contact tracing

Contact tracing is the process of notifying the contacts of a diagnosed positive COVID case and letting them know that they have been exposed, so they can be tested, go into quarantine, or both. Since contact tracing uses diagnosed people to start with, and since people are only diagnosed if they are tested, you must include at least one testing intervention in your sim before including contact tracing. The main option for contact tracing is `trace_prob`, the probability that a person in each layer will be traced (usually high for households and lower for other layers). For example:

```python
# Define the testing and contact tracing interventions
tp = cv.test_prob(symp_prob=0.2, asymp_prob=0.001, symp_quar_prob=1.0, asymp_quar_prob=1.0, do_plot=False)
ct = cv.contact_tracing(trace_probs=dict(h=1.0, s=0.5, w=0.5, c=0.3), do_plot=False)
```

# Define the default parameters
pars = dict(
    pop_type = 'hybrid',
    pop_size = 50e3,
    pop_infected = 100,
)

(continues on next page)
start_day = '2020-02-01',
end_day = '2020-05-01',
interventions = [tp, ct],
)

# Create and run the simulation
sim = cv.Sim(pars)
sim.run()

# Plot the simulation

cv.options.set(font_size=8, show=True)  # Make the font size a bit smaller here so the labels don't overlap
sim.plot(to_plot=['new_infections', 'new_tests', 'new_diagnoses', 'cum_diagnoses', 'new_quarantined', 'test_yield'])
cv.options.set(font_size='default', show=False)  # Reset to the default value

Since it's assumed that known contacts are placed into quarantine (with efficacy \texttt{sim['quar_factor']})
the number of contacts who are successfully traced each day is equal to the number of people newly quarantined (bottom left panel). As is commonly seen using testing and tracing as the only means of epidemic control, these programs stop the epidemic from growing exponentially, but do not bring it to zero.
Since these interventions happen at \( t=0 \), it’s not very useful to plot them. Note that we have turned off plotting by passing `do_plot=False` to each intervention.

### 1.2.5.4 Simple vaccination

Vaccines can do one of two things: they can stop you from becoming infected in the first place (acquisition blocking), or they can stop you from developing symptoms or severe disease once infected (symptom blocking). The Covasim `simple_vaccine` intervention lets you control both of these options. In its simplest form, a vaccine is like a change beta intervention. For example, this vaccine:

```python
vaccine = cv.simple_vaccine(days=30, prob=1.0, rel_sus=0.3, rel_symp=1.0)
```

is equivalent to this beta change:

```python
cb = cv.change_beta(days=30, changes=0.3)
```

But that’s not very realistic. A vaccine given on days 30 and 44 (two weeks later), with efficacy of 50% per dose which accumulates, given to 60% of the population, and which blocks 50% of acquisition and (among those who get infected even so) 90% of symptoms, would look like this:

```python
vaccine = cv.simple_vaccine(days=[30, 44], cumulative=[0.5, 0.5], prob=0.6, rel_sus=0.5, rel_symp=0.1)
```

For more advanced vaccine usage, see T8 - Vaccines and variants.

### 1.2.5.5 Age targeting and other subtargeting

For some interventions, you want them to preferentially apply to some people in the population, e.g. the elderly. For testing and vaccine interventions, this can be done using the `subtarget` argument. Subtargeting is quite flexible in terms of what it accepts, but the easiest approach is usually to define a function to do it. This function must return a dictionary with keys `inds` (which people to subtarget) and `vals` (the values they are subtargeted with). This example shows how to define a 20% transmission-blocking, 94% symptom-blocking vaccine with varying coverage levels by age (for the vaccine intervention, `subtarget` modifies `prob`, the probability of a person being vaccinated) applied on day 20:

```python
import cvasim as cv
import numpy as np

# Define the vaccine subtargeting
def vaccinate_by_age(sim):
    young = cv.bool(sim.people.age < 50)  # cv.bool() returns indices of people matching this condition, i.e. people under 50
    middle = cv.bool((sim.people.age >= 50) * (sim.people.age < 75))  # Multiplication means "and" here
    old = cv.bool(sim.people.age > 75)
    inds = sim.people.uid  # Everyone in the population -- equivalent to np.arange(len(sim.people))
    vals = np.ones(len(sim.people))  # Create the array
    vals[young] = 0.1  # 10% probability for people <50
    vals[middle] = 0.5  # 50% probability for people 50-75
    vals[old] = 0.9  # 90% probability for people >75
    output = dict(inds=inds, vals=vals)
    return output

# Define the vaccine
```
vaccine = cv.simple_vaccine(days=20, rel_sus=0.8, rel_symp=0.06, subtarget=vaccinate_by_age)

# Create, run, and plot the simulations
sim1 = cv.Sim(label='Baseline')
sim2 = cv.Sim(interventions=vaccine, label='With age-targeted vaccine')
msim = cv.parallel(sim1, sim2)
msim.plot()

If you have a simple conditional, you can also define subtargeting using a lambda function, e.g. this is a vaccine with 90% probability of being given to people over age 75, and 10% probability of being applied to everyone else (i.e. people under age 75%):

vaccine = cv.simple_vaccine(days=20, prob=0.1, subtarget=dict(inds=lambda sim: cv.true(sim.people.age>50), vals=0.9))

However, lambda functions are not known for their easy readability, so their use is discouraged.
1.2.5.6 Dynamic parameters

A slightly more advanced intervention is the dynamic parameters intervention. This is not an intervention the same way the others are; instead, it provides a consistent way to modify parameters dynamically. For example, to change the number of imported infections on day 15 from 0 to 100 (representing, say, resuming international travel), and then change back to 0 on day 30:

```python
[6]: import covasim as cv

# Define the dynamic parameters
imports = cv.dynamic_pars(n_imports=dict(days=[15, 30], vals=[100, 0]))

# Create, run, and plot the simulations
sim1 = cv.Sim(label='Baseline')
sim2 = cv.Sim(interventions=imports, label='With imported infections')
msim = cv.parallel(sim1, sim2)
msim.plot()
```

You can see the sudden jump in new infections when importations are turned on.
1.2.5.7 Dynamic triggering

Another option is to replace the `days` arguments with custom functions defining additional criteria. For example, perhaps you only want your beta change intervention to take effect once infections get to a sufficiently high level. Here’s a fairly complex example (feel free to skip the details) that toggles the intervention on and off depending on the current number of people who are infectious.

This example illustrates a few different features:

- The simplest change is just that we’re supplying `days=inf_thresh` instead of a number or list. If we had `def inf_thresh(interv, sim): return [20,30]` this would be the same as just setting `days=[20,30].`

- Because the first argument this function gets is the intervention itself, we can stretch the rules a little bit and name this variable `self` – as if we’re defining a new method for the intervention, even though it’s actually just a function.

- We want to keep track of a few things with this intervention – namely when it toggles on and off, and whether or not it’s active. Since the intervention is just an object, we can add these attributes directly to it.

- Finally, this example shows some of the flexibility in how interventions are plotted – i.e. shown in the legend with a label and with a custom color.

```python
[7]: def inf_thresh(self, sim, thresh=500):
    # Dynamically define on and off days for a beta change -- use self like it's a_
    # method
    # Meets threshold, activate
    if sim.people.infectious.sum() > thresh:
        if not self.active:
            self.active = True
            self.t_on = sim.t
            self.plot_days.append(self.t_on)
    # Does not meet threshold, deactivate
    else:
        if self.active:
            self.active = False
            self.t_off = sim.t
            self.plot_days.append(self.t_off)
    return [self.t_on, self.t_off]
```

```python
# Set up the intervention
on = 0.2 # Beta less than 1 -- intervention is on
off = 1.0 # Beta is 1, i.e. normal -- intervention is off
changes = [on, off]
plot_args = dict(label='Dynamic beta', show_label=True, line_args={'c':'blue'})
cb = cv.change_beta(days=inf_thresh, changes=changes, **plot_args)
```

```python
# Run the simulation and plot
sim = cv.Sim(interventions=cb)
sim.run().plot()
```
1.2.5.8 Custom interventions

If you’re still standing after the previous example, Covasim also lets you do things that are even more complicated, namely define an arbitrary function or class to act as an intervention instead. If a custom intervention is supplied as a function (as it was in Tutorial 1 as well), then it receives the sim object as its only argument, and is called on each timestep. It can perform arbitrary manipulations to the sim object, such as changing parameters, modifying state, etc.

This example reimplements the dynamic parameters example above, except using a hard-coded custom intervention:

```python
[8]: def dynamic_imports(sim):
    if sim.t == 15:
        sim['n_imports'] = 100
    elif sim.t == 30:
        sim['n_imports'] = 0
    return

sim = cv.Sim(interventions=dynamic_imports)
```

However, function-based interventions only take you so far.
We saw in Tutorial 1 how you could define a simple “protect the elderly” intervention with just a few lines of code:

```python
[9]: def protect_elderly(sim):
    if sim.t == sim.day('2020-04-01'):
        elderly = sim.people.age>70
        sim.people.rel_sus[elderly] = 0.0
```

This example explains how to create an intervention object that does much the same thing, but is more fully-featured because it uses the Intervention class.

```python
[10]: import numpy as np
import pylab as pl
import covasim as cv

class protect_elderly(cv.Intervention):
    def __init__(self, start_day=None, end_day=None, age_cutoff=70, rel_sus=0.0, 
                 *args, **kwargs):
        super().__init__(**kwargs)  # NB: This line must be included
        self.start_day = start_day
        self.end_day = end_day
        self.age_cutoff = age_cutoff
        self.rel_sus = rel_sus
        return

    def initialize(self, sim):
        super().initialize()  # NB: This line must also be included
        self.start_day = sim.day(self.start_day)  # Convert string or dateobject
        # dates into an integer number of days
        self.end_day = sim.day(self.end_day)
        self.days = [self.start_day, self.end_day]
        self.elderly = sim.people.age > self.age_cutoff  # Find the elderly people
        self.exposed = np.zeros(sim.npts)  # Initialize results
        self.tvec = sim.tvec  # Copy the time vector into this intervention
        return

    def apply(self, sim):
        self.exposed[sim.t] = sim.people.exposed[self.elderly].sum()

        # Start the intervention
        if sim.t == self.start_day:
            sim.people.rel_sus[self.elderly] = self.rel_sus

        # End the intervention
        elif sim.t == self.end_day:
            sim.people.rel_sus[self.elderly] = 1.0

        return

    def plot(self):
        pl.figure()
```

(continues on next page)
pl.plot(self.tvec, self.exposed)
pl.xlabel('Day')
pl.ylabel('Number infected')
pl.title('Number of elderly people with active COVID')
return

While this example is fairly long, hopefully it’s fairly straightforward:

• __init__() just does what init always does; it’s needed to create the class instance. For interventions, it’s usually used to store keyword arguments and perform some basic initialization (the first two lines).

• initialize() is similar to init, with the difference that it’s invoked when the sim itself is initialized. It receives the sim as an input argument. This means you can use it to do a fairly powerful things. Here, since sim.people already exists, we can calculate up-front who the elderly are so we don’t have to do it on every timestep.

• apply() is the crux of the intervention. It’s run on every timestep of the model, and also receives sim as an input. You almost always use sim.t to get the current timestep, here to turn the intervention on and off. But as this example shows, its real power is that it can make direct modifications to the sim itself (sim.people.rel_sus[self.elderly] = self.rel_sus). It can also perform calculations and store data in itself, as shown here with self.exposed (although in general, analyzers are better for this, since they happen at the end of the timestep, while interventions happen in the middle).

• plot() is a custom method that shows a plot of the data gathered during the sim. Again, it’s usually better to use analyzers for this, but for something simple like this it’s fine to double-dip and use an intervention.

Here is what this custom intervention looks like in action. Note how it automatically shows when the intervention starts and stops (with vertical dashed lines).

```python
# Define and run the baseline simulation
pars = dict(
    pop_size = 50e3,
    pop_infected = 100,
    n_days = 90,
    verbose = 0,
)
orig_sim = cv.Sim(pars, label='Default')

# Define the intervention and the scenario sim
protect = protect_elderly(start_day='2020-04-01', end_day='2020-05-01', rel_sus=0.1)  
→ # Create intervention
sim = cv.Sim(pars, interventions=protect, label='Protect the elderly')

# Run and plot
msim = cv.parallel(orig_sim, sim)
msim.plot()
```

1.2. Tutorials
Because we collected information in the intervention itself, we can also plot that:

```python
# Plot intervention
protect = msim.sims[1].get_intervention(protect_elderly)  # Find intervention by type
protect.plot()
```
1.2.6 T6 - Using analyzers

Analyzers are objects that do not change the behavior of a simulation, but just report on its internal state, almost always something to do with `sim.people`. This tutorial takes you through some of the built-in analyzers and gives a brief example of how to build your own.

Click here to open an interactive version of this notebook.

1.2.6.1 Age histograms and snapshots

Age histograms and snapshots both take “pictures” of the `sim.people` object at specified points in time. This is because while most of the information from `sim.people` is retrievable at the end of the sim from the stored events, it’s much easier to see what’s going on at the time. While the snapshot literally just makes a copy of the people object, the age histogram calculates results for different age bins:

```
[1]: import covasim as cv
cv.options.set('jupyter', verbose=0)

sim = cv.Sim(interventions=cv.test_prob(0.5), analyzers=cv.age_histogram())
sim.run()
agehist = sim.get_analyzer() # Only one analyzer so we can retrieve it like this
agehist.plot()
```
1.2.6.2 Transmission trees

Another useful analysis to perform on the simulation is to calculate the transmission tree – i.e., who infected whom. Since all this information is stored in the sim, it doesn’t have to be inserted at run time; it can be added to an already finished simulation:

```
[2]: tt = sim.make_transtree()
    fig1 = tt.plot()
    fig2 = tt.plot_histograms()
```
Covasim, Release 3.1.1

1.2. Tutorials

Number of transmissions per person

Numbers of events and transmissions

Number of transmissions, by transmissions per person

Proportion of transmissions, by proportion of population

Proportion of infections caused (%)

Proportion of population, ordered by the number of people they infected (%)

Cumulative infections

0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5 20.0
Day
0
500
1000
1500

0 20 40 60
Day
0

0 200 400 600 800
Count
Numbers of events and transmissions

Number of events (events x transmissions per event)

Cumulative infections

0 500 1000 1500
Day
0

0 20 40 60
Day
0

1.2.6.3 Custom analyzers

Analyzers don’t have to be complicated. While analyzers can derive from the `cv.Analyzer` class, they can also just be simple functions, unless they need to keep track of or update internal state. Here’s an example of an analyzer that does keep track of internal state, namely by storing and plotting the S, E, I, and R compartments over time.

```python
import pylab as pl
import sciris as sc
import covasim as cv

class store_seir(cv.Analyzer):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)  # This is necessary to initialize the class properly
        self.t = []
        self.S = []
        self.E = []
        self.I = []
        self.R = []
        return

    def apply(self, sim):
        ppl = sim.people  # Shorthand
        self.t.append(sim.t)
        self.S.append(ppl.susceptible.sum())
        self.E.append(ppl.exposed.sum() - ppl.infectious.sum())
        self.I.append(ppl.infectious.sum())
        self.R.append(ppl.recovered.sum() + ppl.dead.sum())
        return

    def plot(self):
        pl.figure()
        pl.plot(self.t, self.S, label='S')
        pl.plot(self.t, self.E, label='E')
        pl.plot(self.t, self.I, label='I')
        pl.plot(self.t, self.R, label='R')
        pl.legend()
        pl.xlabel('Day')
        pl.ylabel('People')
        sc.setylim()  # Reset y-axis to start at 0
        sc.commaticks()  # Use commas in the y-axis labels
        return

sim = cv.Sim(n_days=180, analyzers=store_seir(label='seir'))
sim.run()
seir = sim.get_analyzer('seir')  # Retrieve by label
seir.plot()
```
And here’s an example of an analyzer that doesn’t need any internal state, so it can just be a function: it simply reports if anyone aged 88 is currently infected.

```python
[4]: def check_88(sim):
   people_who_are_88 = sim.people.age.round() == 88 # Find everyone who’s aged 88 (to the nearest year)
   people_exposed = sim.people.exposed # Find everyone who’s infected with COVID
   people_who_are_88_with_covid = cv.true(people_who_are_88 * people_exposed) # Multiplication is the same as logical "and"
   n = len(people_who_are_88_with_covid) # Count how many people there are
   if n:
       print(f'Oh no! {n} people aged 88 have covid on timestep {sim.t} \{"\times\}n')
   return

sim = cv.Sim(n_days=120, analyzers=check_88, verbose=0)
sim.run()
```

Oh no! 2 people aged 88 have covid on timestep 20
Oh no! 2 people aged 88 have covid on timestep 21
Oh no! 2 people aged 88 have covid on timestep 22
Oh no! 2 people aged 88 have covid on timestep 23
Oh no! 3 people aged 88 have covid on timestep 24
Oh no! 3 people aged 88 have covid on timestep 25
Oh no! 3 people aged 88 have covid on timestep 26
Oh no! 3 people aged 88 have covid on timestep 27
Oh no! 3 people aged 88 have covid on timestep 28
Oh no! 3 people aged 88 have covid on timestep 29
Oh no! 3 people aged 88 have covid on timestep 30
(continues on next page)
Oh no! 3 people aged 88 have covid on timestep 31
Oh no! 3 people aged 88 have covid on timestep 32
Oh no! 3 people aged 88 have covid on timestep 33
Oh no! 3 people aged 88 have covid on timestep 34
Oh no! 3 people aged 88 have covid on timestep 35
Oh no! 3 people aged 88 have covid on timestep 36
Oh no! 3 people aged 88 have covid on timestep 37
Oh no! 4 people aged 88 have covid on timestep 38
Oh no! 5 people aged 88 have covid on timestep 39
Oh no! 6 people aged 88 have covid on timestep 40
Oh no! 7 people aged 88 have covid on timestep 41
Oh no! 7 people aged 88 have covid on timestep 42
Oh no! 8 people aged 88 have covid on timestep 43
Oh no! 12 people aged 88 have covid on timestep 44
Oh no! 12 people aged 88 have covid on timestep 45
Oh no! 14 people aged 88 have covid on timestep 46
Oh no! 15 people aged 88 have covid on timestep 47
Oh no! 15 people aged 88 have covid on timestep 48
Oh no! 16 people aged 88 have covid on timestep 49
Oh no! 17 people aged 88 have covid on timestep 50
Oh no! 16 people aged 88 have covid on timestep 51
Oh no! 15 people aged 88 have covid on timestep 52
Oh no! 18 people aged 88 have covid on timestep 53
Oh no! 19 people aged 88 have covid on timestep 54
Oh no! 20 people aged 88 have covid on timestep 55
Oh no! 21 people aged 88 have covid on timestep 56
Oh no! 21 people aged 88 have covid on timestep 57
Oh no! 22 people aged 88 have covid on timestep 58
Oh no! 25 people aged 88 have covid on timestep 59
Oh no! 28 people aged 88 have covid on timestep 60
Oh no! 28 people aged 88 have covid on timestep 61
Oh no! 28 people aged 88 have covid on timestep 62
Oh no! 27 people aged 88 have covid on timestep 63
Oh no! 28 people aged 88 have covid on timestep 64
Oh no! 28 people aged 88 have covid on timestep 65
Oh no! 27 people aged 88 have covid on timestep 66
Oh no! 27 people aged 88 have covid on timestep 67
Oh no! 25 people aged 88 have covid on timestep 68
Oh no! 22 people aged 88 have covid on timestep 69
Oh no! 19 people aged 88 have covid on timestep 70
Oh no! 18 people aged 88 have covid on timestep 71
Oh no! 16 people aged 88 have covid on timestep 72
Oh no! 15 people aged 88 have covid on timestep 73
Oh no! 13 people aged 88 have covid on timestep 74
Oh no! 10 people aged 88 have covid on timestep 75
Oh no! 10 people aged 88 have covid on timestep 76
Oh no! 10 people aged 88 have covid on timestep 77
Oh no! 10 people aged 88 have covid on timestep 78
Oh no! 10 people aged 88 have covid on timestep 79
Oh no! 8 people aged 88 have covid on timestep 80
Oh no! 8 people aged 88 have covid on timestep 81
Oh no! 9 people aged 88 have covid on timestep 82
Oh no! 10 people aged 88 have covid on timestep 83
Oh no! 10 people aged 88 have covid on timestep 84
Oh no! 8 people aged 88 have covid on timestep 85
Oh no! 7 people aged 88 have covid on timestep 86
Oh no! 7 people aged 88 have covid on timestep 87

(continues on next page)
1.2.7 T7 - Calibration

We saw in Tutorial 4 how to load and plot data. But the next step is to actually calibrate the model to the data, i.e. find the model parameters that are the most likely explanation for the observed data. This tutorial gives an introduction to the Fit object and some recipes for optimization approaches.

Click here to open an interactive version of this notebook.

1.2.7.1 The Fit object

The Fit object is responsible for quantifying how well a given model run matches the data. Let’s consider a simple example, building on Tutorial 4:

```python
[1]: import covasim as cv
cv.options.set('jupyter', verbose=0)
```

(continues on next page)
pars = dict(
    pop_size = 20_000,
    start_day = '2020-02-01',
    end_day = '2020-04-11',
    beta = 0.015,
)
sim = cv.Sim(pars=pars, datafile='example_data.csv', interventions=cv.test_num(daily_tests='data'))
sim.run()
sim.plot(to_plot=['cum_tests', 'cum_diagnoses', 'cum_deaths'])

We can see that tests match extremely well (they’re input data!), diagnoses match reasonably well, and deaths match poorly. Can the Fit object capture our intuition about this?

[2]: fit = sim.compute_fit()
print(fit.mismatches)
print(fit.mismatch)

#0. cum_deaths: 84.78260869565216
#1. cum_tests: 0.0

(continues on next page)
So the results seem to match our intuition. (Note that by default the Fit object uses normalized absolute difference, but other estimates, such as mean squared error, are also possible.)

What if we improve the fit? Does the mismatch reduce?

```python
[3]: sim['rel_death_prob'] = 2  # Double the death rate since deaths were too low
sim.initialize(reset=True)  # Reinitialize the sim

# Rerun and compute fit
sim.run()
fit = sim.compute_fit()

# Output
sim.plot()
fit.plot()
print(fit.mismatches)
print(fit.mismatch)
```

As expected, the fit is improved.
1.2.7.2 Calibration approaches

Calibration is a complex and dark art and cannot be covered fully here; many books have been written about it and it continues to be an area of active research. A good review article about calibrating agent-based models like Covasim is available here. Calibration is usually expressed as an optimization problem: specifically, find a vector of parameters \( \theta \) that minimizes the mismatch between the data \( D \) and the model \( M(\theta) \).

In practice, most calibration is done simply by hand, as in the example above. Once deaths are “calibrated”, the user might modify testing assumptions so that the diagnoses match. Since we are only fitting to deaths and diagnoses, the model is then “calibrated”.

However, automated approaches to calibration are possible as well. The simplest is probably the built-in SciPy optimization functions, e.g. `scipy.optimize`. A wrinkle here is that normal gradient descent methods will not work with Covasim or other agent-based models, due to the stochastic variability between model runs that makes the landscape very “bumpy”. One way of getting around this is to use many different runs and take the average, e.g.:

```python
import covasim as cv
import numpy as np
import scipy

def objective(x, n_runs=10):
    print(f'Running sim for beta={x[0]}, rel_death_prob={x[1]}')
    pars = dict(
        pop_size = 20_000,
        start_day = '2020-02-01',
        end_day = '2020-04-11',
        beta = x[0],
        rel_death_prob = x[1],
        verbose = 0,
    )
    sim = cv.Sim(pars=pars, datafile='/home/cliffk/idm/covasim/docs/tutorials/example_data.csv', interventions=cv.test_num(daily_tests='data'))
    msim = cv.MultiSim(sim)
    mismatches = []
    for sim in msim.sims:
        fit = sim.compute_fit()
        mismatches.append(fit.mismatch)
        mismatch = np.mean(mismatches)
    return mismatch

guess = [0.015, 1]  # Initial guess of parameters -- beta and relative death probability
pars = scipy.optimize.minimize(objective, x0=guess, method='nelder-mead')  # Run the optimization
```

This should converge after roughly 3-10 minutes, although you will likely find that the improvement is minimal.

What’s happening here? Trying to overcome the limitations of an algorithm that expects deterministic results simply by running more sims is fairly futile – if you run \( N \) sims and average them together, you’ve only reduced noise by \( N \), i.e. you have to average together 100 sims to reduce noise by a factor of 10, and even that might not be enough. Clearly, we need a more powerful approach.

1.2.7.3 Built-in calibration

One such package we have found works reasonably well is called Optuna. It is built into Covasim as `sim.calibrate()` (it’s not installed by default, so please install it first with `pip install optuna`). Do not expect
this to be a magic bullet solution: you will likely still need to try out multiple different parameter sets for calibration, manually update the values of uncalibrated parameters, check if the data actually make sense, etc. Even once all these things are in place, it still needs to be run for enough iterations, which might be a few hundred iterations for 3-4 calibrated (free) parameters or tens of thousands of iterations for 10 or more free parameters. The example below should get you started, but best to expect that it will not work for your particular use case without significant modification!

```
[4]: ''
Example for running built-in calibration with Optuna''

import sciris as sc
import covasim as cv

# Create default simulation
pars = sc.objdict(
    pop_size = 20_000,
    start_day = '2020-02-01',
    end_day = '2020-04-11',
    beta = 0.015,
    rel_death_prob = 1.0,
    interventions = cv.test_num(daily_tests='data'),
    verbose = 0,
)

sim = cv.Sim(pars=pars, datafile='example_data.csv')

# Parameters to calibrate -- format is best, low, high
calib_pars = dict(
    beta = [pars.beta, 0.005, 0.20],
    rel_death_prob = [pars.rel_death_prob, 0.5, 3.0],
)

if __name__ == '__main__':
    # Run the calibration
    n_trials = 20
    n_workers = 4
    calib = sim.calibrate(calib_pars=calib_pars, n_trials=n_trials, n_workers=n_workers)

[I 2021-12-06 06:29:48.180] A new study created in RDB with name: covasim_calibration
[I 2021-12-06 06:29:49.560] Trial 0 finished with value: 390.1866367633344 and parameters: ('beta': 0.02472114564187514, 'rel_death_prob': 0.6539395897685286).
   - Best is trial 0 with value: 390.1866367633344.
[I 2021-12-06 06:29:49.728] Trial 1 finished with value: 1216.069303193143 and parameters: ('beta': 0.08051698655601072, 'rel_death_prob': 1.2571780830433488).
   - Best is trial 0 with value: 390.1866367633344.
[I 2021-12-06 06:29:50.978] Trial 2 finished with value: 2083.639805480304 and parameters: ('beta': 0.11077686386120086, 'rel_death_prob': 2.0034928035447113).
   - Best is trial 0 with value: 390.1866367633344.
   - Best is trial 0 with value: 390.1866367633344.
[I 2021-12-06 06:29:52.344] Trial 4 finished with value: 1368.48470044855 and parameters: ('beta': 0.06978745801495069, 'rel_death_prob': 1.7444108792037367).
   - Best is trial 0 with value: 390.1866367633344.
[I 2021-12-06 06:29:52.586] Trial 5 finished with value: 2106.899120664387 and parameters: ('beta': 0.11079415122376196, 'rel_death_prob': 2.392797558356755).
   - Best is trial 0 with value: 390.1866367633344.

(continues on next page)
Trial 6 finished with value: 1062.1731358529112 and parameters: {'beta': 0.03341431697143469, 'rel_death_prob': 1.8925583875799161}.
- Best is trial 0 with value: 390.1866367633344.

Trial 7 finished with value: 2892.164367402225 and parameters: {'beta': 0.15508440311519098, 'rel_death_prob': 2.9616935502325292}.
- Best is trial 0 with value: 390.1866367633344.

Trial 8 finished with value: 2300.1658746724693 and parameters: {'beta': 0.08649659798588784, 'rel_death_prob': 2.7543068373894855}.
- Best is trial 0 with value: 390.1866367633344.

Trial 9 finished with value: 1455.2380423679886 and parameters: {'beta': 0.12566256060197206, 'rel_death_prob': 1.3824008454582932}.
- Best is trial 11 with value: 146.65896877914463.

Trial 10 finished with value: 3108.4591641870584 and parameters: {'beta': 0.18350420591955913, 'rel_death_prob': 2.8621609146753437}.
- Best is trial 11 with value: 146.65896877914463.

Trial 11 finished with value: 146.65896877914463 and parameters: {'beta': 0.005375656541709441, 'rel_death_prob': 0.5042390759255477}.
- Best is trial 11 with value: 146.65896877914463.

Trial 12 finished with value: 110.12457254518809 and parameters: {'beta': 0.011560523833422825, 'rel_death_prob': 0.5164208788068061}.
- Best is trial 12 with value: 110.12457254518809.

Trial 13 finished with value: 140.1216858373673 and parameters: {'beta': 0.008734894128077604, 'rel_death_prob': 0.623680740642698}.
- Best is trial 12 with value: 110.12457254518809.

Trial 14 finished with value: 88.21135142336902 and parameters: {'beta': 0.013103401465194855, 'rel_death_prob': 0.5435085536530619}.
- Best is trial 14 with value: 88.21135142336902.

Trial 15 finished with value: 737.954989563441 and parameters: {'beta': 0.050421330357321624, 'rel_death_prob': 0.8751734445720557}.
- Best is trial 14 with value: 88.21135142336902.

Trial 16 finished with value: 880.6943011764445 and parameters: {'beta': 0.05419104161894099, 'rel_death_prob': 1.013602035770908}.
- Best is trial 14 with value: 88.21135142336902.

Trial 17 finished with value: 841.98538817338 and parameters: {'beta': 0.057210889428077604, 'rel_death_prob': 0.9766392610393954}.
- Best is trial 14 with value: 88.21135142336902.

Trial 18 finished with value: 1000.0318648132522 and parameters: {'beta': 0.03911932285462054, 'rel_death_prob': 1.0405280384741267}.
- Best is trial 14 with value: 88.21135142336902.

Trial 19 finished with value: 96.120708797722 and parameters: {'beta': 0.015900369970194433, 'rel_death_prob': 0.751785302519252}.
- Best is trial 14 with value: 88.21135142336902.

Trial 20 finished with value: 209.7464138206688 and parameters: {'beta': 0.014536344981265941, 'rel_death_prob': 0.7999803420287836}.
- Best is trial 14 with value: 88.21135142336902.
[I 2021-12-06 06:30:04,872] Trial 25 finished with value: 632.8323044810587 and parameters: {'beta': 0.042470939630231, 'rel_death_prob': 0.7549052357442119}. Best is trial 14 with value: 88.21135142336902.

[I 2021-12-06 06:30:06,009] Trial 26 finished with value: 906.1232624239465 and parameters: {'beta': 0.07641243346265292, 'rel_death_prob': 0.8188463448295625}. Best is trial 14 with value: 88.21135142336902.

[I 2021-12-06 06:30:06,269] Trial 27 finished with value: 1216.6224408224896 and parameters: {'beta': 0.0722456946107525, 'rel_death_prob': 1.2324594992325624}. Best is trial 14 with value: 88.21135142336902.

[I 2021-12-06 06:30:07,401] Trial 28 finished with value: 1099.8045920859795 and parameters: {'beta': 0.06418410723448698, 'rel_death_prob': 1.259794598651943}. Best is trial 14 with value: 88.21135142336902.

[I 2021-12-06 06:30:07,708] Trial 29 finished with value: 1440.5300439667806 and parameters: {'beta': 0.09612708255418681, 'rel_death_prob': 1.5596057285270204}. Best is trial 14 with value: 88.21135142336902.

[I 2021-12-06 06:30:08,650] Trial 30 finished with value: 357.19656259714884 and parameters: {'beta': 0.021679445520447493, 'rel_death_prob': 1.553893935202204}. Best is trial 14 with value: 88.21135142336902.

[I 2021-12-06 06:30:09,022] Trial 31 finished with value: 509.9811253719412 and parameters: {'beta': 0.026740138282013257, 'rel_death_prob': 0.6805764623798962}. Best is trial 14 with value: 88.21135142336902.

[I 2021-12-06 06:30:09,924] Trial 32 finished with value: 378.55609095350184 and parameters: {'beta': 0.024292062288107895, 'rel_death_prob': 0.5242153640881778}. Best is trial 14 with value: 88.21135142336902.

[I 2021-12-06 06:30:09,985] Trial 33 finished with value: 140.1216858373673 and parameters: {'beta': 0.008762813792770038, 'rel_death_prob': 0.5056476459260479}. Best is trial 14 with value: 88.21135142336902.

[I 2021-12-06 06:30:10,866] Trial 34 finished with value: 144.1627437047564 and parameters: {'beta': 0.00597452934724196, 'rel_death_prob': 0.6843469590304303}. Best is trial 14 with value: 88.21135142336902.

[I 2021-12-06 06:30:10,905] Trial 35 finished with value: 141.24050717235866 and parameters: {'beta': 0.005120956127635913, 'rel_death_prob': 0.6840267267533198}. Best is trial 14 with value: 88.21135142336902.

[I 2021-12-06 06:30:12,221] Trial 36 finished with value: 1338.388550872674 and parameters: {'beta': 0.04377917500046565, 'rel_death_prob': 2.2769161524616175}. Best is trial 14 with value: 88.21135142336902.

[I 2021-12-06 06:30:12,370] Trial 37 finished with value: 2431.241950526269 and parameters: {'beta': 0.13859882111601293, 'rel_death_prob': 2.328527686674377}. Best is trial 14 with value: 88.21135142336902.

[I 2021-12-06 06:30:13,425] Trial 38 finished with value: 237.40806945863125 and parameters: {'beta': 0.02123181874736125, 'rel_death_prob': 0.9376793359835718}. Best is trial 14 with value: 88.21135142336902.

[I 2021-12-06 06:30:13,561] Trial 39 finished with value: 182.43504907403297 and parameters: {'beta': 0.018063974119405635, 'rel_death_prob': 0.902640977300549}. Best is trial 14 with value: 88.21135142336902.

[I 2021-12-06 06:30:14,851] Trial 40 finished with value: 626.78653905937737 and parameters: {'beta': 0.03470667282254178, 'rel_death_prob': 0.8910216529069218}. Best is trial 14 with value: 88.21135142336902.

[I 2021-12-06 06:30:15,152] Trial 41 finished with value: 1609.21847371144 and parameters: {'beta': 0.19865988017570096, 'rel_death_prob': 1.143696123112417}. Best is trial 14 with value: 88.21135142336902.

[I 2021-12-06 06:30:15,845] Trial 42 finished with value: 88.17238086778879 and parameters: {'beta': 0.014658641850953356, 'rel_death_prob': 0.610450270947137}. Best is trial 42 with value: 88.17238086778879.

[I 2021-12-06 06:30:16,171] Trial 43 finished with value: 74.41477550295332 and parameters: {'beta': 0.015644564127385754, 'rel_death_prob': 0.6156891871143325}. Best is trial 43 with value: 74.41477550295332.
Trial 44 finished with value: 553.99586792719 and parameters: {'beta': 0.03089147573384466, 'rel_death_prob': 0.6274246377059536}. Best is trial 43 with value: 74.4177550295332.

Trial 45 finished with value: 465.5481414042723 and parameters: {'beta': 0.0299228616926584, 'rel_death_prob': 0.6719418776865646}. Best is trial 43 with value: 74.4177550295332.

Trial 46 finished with value: 78.38166718479371 and parameters: {'beta': 0.015769149745229447, 'rel_death_prob': 0.7263637696202627}. Best is trial 43 with value: 74.4177550295332.

Trial 47 finished with value: 1925.2872274281651 and parameters: {'beta': 0.06290400598904239, 'rel_death_prob': 2.5424274038693007}. Best is trial 43 with value: 74.4177550295332.

Trial 48 finished with value: 1421.0196740240708 and parameters: {'beta': 0.061786751116939626, 'rel_death_prob': 1.8319281554879798}. Best is trial 43 with value: 74.4177550295332.

Trial 49 finished with value: 1327.2576275702804 and parameters: {'beta': 0.048830200675546095, 'rel_death_prob': 2.0480235837170686}. Best is trial 43 with value: 74.4177550295332.

Trial 50 finished with value: 1347.8120975263134 and parameters: {'beta': 0.08607852001738546, 'rel_death_prob': 1.4380633126436735}. Best is trial 43 with value: 74.4177550295332.

Trial 51 finished with value: 105.18308389217036 and parameters: {'beta': 0.01403846828862633, 'rel_death_prob': 0.6217466773332628}. Best is trial 43 with value: 74.4177550295332.

Trial 52 finished with value: 95.31342985299995 and parameters: {'beta': 0.015415639732559733, 'rel_death_prob': 0.7658497083459304}. Best is trial 43 with value: 74.4177550295332.

Trial 53 finished with value: 94.86299240573788 and parameters: {'beta': 0.015309290106272425, 'rel_death_prob': 0.7564008522050857}. Best is trial 43 with value: 74.4177550295332.

Trial 54 finished with value: 670.7170138117867 and parameters: {'beta': 0.03553293173111602, 'rel_death_prob': 0.8117706820033146}. Best is trial 43 with value: 74.4177550295332.

Trial 55 finished with value: 728.7521650308656 and parameters: {'beta': 0.03263810635738369, 'rel_death_prob': 1.0606037531787378}. Best is trial 43 with value: 74.4177550295332.

Trial 56 finished with value: 639.57409956909747 and parameters: {'beta': 0.02529491276909747, 'rel_death_prob': 1.0931162693260115}. Best is trial 43 with value: 74.4177550295332.

Trial 57 finished with value: 312.97997068881284 and parameters: {'beta': 0.022968928020786962, 'rel_death_prob': 0.620926247455964}. Best is trial 43 with value: 74.4177550295332.

Trial 58 finished with value: 769.8168050806057 and parameters: {'beta': 0.11748981493976007, 'rel_death_prob': 0.6019651318492055}. Best is trial 43 with value: 74.4177550295332.

Trial 59 finished with value: 1110.8651241284363 and parameters: {'beta': 0.11711027348222276, 'rel_death_prob': 0.9789055193128524}. Best is trial 43 with value: 74.4177550295332.

Trial 60 finished with value: 117.02535861799311 and parameters: {'beta': 0.012637488609287317, 'rel_death_prob': 0.7642017575891364}. Best is trial 43 with value: 74.4177550295332.

Trial 61 finished with value: 1260.4126881911445 and parameters: {'beta': 0.15049042592263506, 'rel_death_prob': 0.9673148194334464}. Best is trial 43 with value: 74.4177550295332.
[I 2021-12-06 06:30:28,947] Trial 63 finished with value: 97.62523870853133 and
(parameters: {'beta': 0.014009732315682854, 'rel_death_prob': 0.7493477892856604}.

Best is trial 43 with value: 74.1477550295332.

[I 2021-12-06 06:30:29,166] Trial 64 finished with value: 142.48057023582183 and
(parameters: {'beta': 0.01429925365687652, 'rel_death_prob': 0.7643103025033169}.

Best is trial 43 with value: 74.1477550295332.

[I 2021-12-06 06:30:30,084] Trial 65 finished with value: 168.28340809166406 and
(parameters: {'beta': 0.01726006962607253, 'rel_death_prob': 0.8547213799069879}.

Best is trial 43 with value: 74.41477550295332.

[I 2021-12-06 06:30:30,306] Trial 66 finished with value: 849.8648798685438 and
(parameters: {'beta': 0.0385388187023326, 'rel_death_prob': 1.1956780382739103}.

Best is trial 43 with value: 74.41477550295332.

[I 2021-12-06 06:30:30,999] Trial 67 finished with value: 526.171559266332 and
(parameters: {'beta': 0.026826878703600884, 'rel_death_prob': 0.5448674836633214}.

Best is trial 43 with value: 74.41477550295332.

[I 2021-12-06 06:30:31,657] Trial 68 finished with value: 137.17402407070213 and
(parameters: {'beta': 0.009402544753272624, 'rel_death_prob': 0.8532424189889245}.

Best is trial 43 with value: 74.41477550295332.

[I 2021-12-06 06:30:32,338] Trial 69 finished with value: 254.07625349735756 and
(parameters: {'beta': 0.019475095518037323, 'rel_death_prob': 0.583780438993774}.

Best is trial 43 with value: 74.41477550295332.

Making results structure...
Processed 80 trials; 0 failed
Removed existing calibration covasim Calibration.db
Calibration for 80 total trials completed in 50.1 s.
Initial parameter values:
#0. beta: 0.015
#1. rel_death_prob: 1.0

Best parameter values:
#0. beta: 0.015644564127385754
#1. rel_death_prob: 0.6156891871143325

Mismatch before calibration: 101.465
Mismatch after calibration: 74.4148
Percent improvement: 26.7%

So it improved the fit (see above), but let's visualize this as a plot:

```python
# Plot the results
calib.plot_trend()
calib.plot_sims(to_plot=['cum_tests', 'cum_diagnoses', 'cum_deaths'])
```

![Cumulative tests](image1)
![Cumulative diagnoses](image2)
![Cumulative deaths](image3)
Compared to `scipy.optimize.minimize()`, Optuna took less time and produced a much better fit. However, it’s still far from perfect – more iterations, and calibrating more parameters beyond just these two, would still be required before the model could be considered “calibrated”.

Sometimes you want to calibrate a parameter that isn’t a built-in parameter – for example, part of an intervention. You can do this using the `custom_fn` keyword argument. In this example, we calibrate the test probability by modifying the `test_prob` intervention directly.

```python
[6]: pars = dict(
    verbose = 0,
    start_day = '2020-02-05',
    pop_size = 1000,
    pop_scale = 4,
    interventions = cv.test_prob(symp_prob=0.01),
)

sim = cv.Sim(pars, datafile='example_data.csv')

calib_pars = dict(
    beta = [0.013, 0.005, 0.020],
    test_prob = [0.01, 0.00, 0.30]
)

def set_test_prob(sim, calib_pars):
    (continues on next page)
tp = sim.get_intervention(cv.test_prob)
tp.symp_prob = calib_pars['test_prob']

return sim

calib = sim.calibrate(calib_pars=calib_pars, custom_fn=set_test_prob, n_trials=5)
calib.plot_all()
calib.plot_sims(to_plot=['cum_deaths', 'cum_diagnoses'])

[I 2021-12-06 06:30:41,645] Trial 2 finished with value: 136.04022024960972 and parameters: {'beta': 0.011608058087546867, 'test_prob': 0.10010501564541523}. Best is trial 1 with value: 95.1337442240409.

[I 2021-12-06 06:30:41,953] Trial 4 finished with value: 105.35697849341611 and parameters: {'beta': 0.00963532079617595, 'test_prob': 0.1640065518371752}. Best is trial 1 with value: 95.1337442240409.

[I 2021-12-06 06:30:42,258] Trial 6 finished with value: 121.68202080405578 and parameters: {'beta': 0.00810419049581134, 'test_prob': 0.2352308055988633}. Best is trial 1 with value: 95.1337442240409.

[I 2021-12-06 06:30:42,345] Trial 7 finished with value: 121.68202080405578 and parameters: {'beta': 0.0158608846688399, 'test_prob': 0.199406336282283}. Best is trial 1 with value: 95.1337442240409.

[I 2021-12-06 06:30:42,604] Trial 9 finished with value: 142.4365641054436 and parameters: {'beta': 0.006866037133806003, 'test_prob': 0.2352308055988633}. Best is trial 1 with value: 95.1337442240409.

Making results structure...
Processed 10 trials; 0 failed
Removed existing calibration covasim_calibration.db
Calibration for 10 total trials completed in 1.8 s.

Initial parameter values:
#0. beta: 0.013
#1. test_prob: 0.01

Best parameter values:
#0. beta: 0.01712384580177092
#1. test_prob: 0.1463134790482913

Mismatch before calibration: 141.953
Mismatch after calibration: 95.1337
Percent improvement: 33.0%
Cumulative deaths

Before calibration
After calibration
Data

Cumulative diagnoses

Before calibration
After calibration
Data
1.2.8 T8 - Vaccines and variants

This tutorial covers several of the features new to Covasim 3.0, including waning immunity, multi-variant modelling, and advanced vaccination methods.

Click here to open an interactive version of this notebook.

1.2.8.1 Using waning immunity

By default, immunity is assumed to grow and then wane over time according to a two-part exponential decay of neutralizing antibodies. This can be changed by setting `use_waning=False`, such that infection is assumed to confer lifelong perfect immunity, meaning that people who have been infected cannot be infected again. When `use_waning` is set to True, agents in the simulation are assigned a peak level of neutralizing antibodies upon infection or vaccination, drawn from a distribution defined in the parameter dictionary. This level grows to its peak and then decays over time, leading to declines in the efficacy of protection against infection, symptoms, and severe symptoms. The following example creates simulations without waning immunity, and compares it to simulations with different speeds of immunity waning.
import numpy as np
import covasim as cv
import pylab as pl

cv.options.set('jupyter', verbose=0)

# Create sims with and without waning immunity
sim_nowaning = cv.Sim(n_days=120, use_waning=False, label='No waning immunity')
sim_waning = cv.Sim(n_days=120, label='Waning immunity')

# Now create an alternative sim with faster decay for neutralizing antibodies
sim_fasterwaning = cv.Sim(
    label='Faster waning immunity',
    n_days=120,
    nab_decay=dict(form='nab_growth_decay', growth_time=21, decay_rate1=0.07, decay_time1=47, decay_rate2=0.02, decay_time2=106)
)

# Create a multisim, run, and plot results
msim = cv.parallel(sim_nowaning, sim_waning, sim_fasterwaning)
msim.plot()
1.2.8.2 Multi-variant modeling

The next examples show how to introduce new variants into a simulation. These can either be known variants of concern, or custom new variants. New variants may have differing levels of transmissibility, symptomaticity, severity, and mortality. When introducing new variants, use_waning must be set to True. The model includes known information about the levels of cross-immunity between different variants. Cross-immunity can also be manually adjusted.

[2]: # Define three new variants: B117, B1351, and a custom-defined variant
alpha = cv.variant('alpha', days=0, n_imports=10)
beta = cv.variant('beta', days=0, n_imports=10)
custom = cv.variant(label='3x more transmissible', variant={'rel_beta': 3.0}, days=7, n_imports=10)

# Create the simulation
sim = cv.Sim(variants=[alpha, beta, custom], pop_infected=10, n_days=32)

# Run and plot
sim.run()
sim.plot('variant')
1.2.8.3 Advanced vaccination methods

The intervention `cv.BaseVaccination` allows you to introduce a selection of known vaccines into the model, each of which is pre-populated with known parameters on their efficacy against different variants, their durations of protection, and the levels of protection that they afford against infection and disease progression. The prioritization of vaccines is implemented with derived classes that implement specific allocation algorithms. Covasim 3.0 comes with two such algorithms:

- `cv.vaccinate_prob()` - specify a daily probability of vaccination for each person
- `cv.vaccinate_num()` - specify a sequence of people to vaccinate, and the number of available doses each day

When using any of these vaccination interventions, `use_waning` must be set to `True`.

Probability-based vaccination

The intervention `cv.vaccinate_prob()` allows you specify the daily probability that each individual gets vaccinated. The following example illustrates how to use the `cv.vaccinate_prob()` intervention.

```python
# Create some base parameters
pars = dict(
    beta = 0.015,
    n_days = 90,
)

# Define probability based vaccination
pfizer = cv.vaccinate_prob(vaccine='pfizer', days=20, prob=0.8)

# Create and run the sim
sim = cv.Sim(pars=pars, interventions=pfizer)
sim.run()
sim.plot(['new_infections', 'cum_infections', 'new_doses', 'cum_doses'])
```
Sequence-based vaccination

To use `cv.vaccinate_num`, it is necessary to specify the vaccine prioritization - for example, this may involve defining priority groups like 1A, 1B etc. depending on the setting. The vaccine prioritization is specified as an ordered sequence of people to vaccinate, so in almost all cases, a function can be defined that takes in a `cv.People` instance, and returns an array of indices specifying the order in which people get vaccinated. This function could also incorporate steps such as randomizing the order of people within priority groups, or removing some people from the sequence to account for vaccine hesitancy and peak coverage not reaching 100%. A simple example of a prioritization function would be to simply sort by age in descending order i.e.

```python
[4]: def prioritize_by_age(people):
    return np.argsort(-people.age)
```

**Note:** Because age prioritization is so commonly used, you can use the shortcut `sequence='age'` rather than specify this function.

This function can be passed to `cv.vaccinate_num` where it will be evaluated during initialization, and therefore will run after the population has been generated. In cases where the `cv.People` have been generated offline and are
being loaded instead of generated, it’s possible to pass a pre-computed sequence of indices to `cv.vaccinate_num` rather than a prioritization function that returns the sequence.

The example below also shows how to use a simple Analyzer to capture additional information about the vaccine state each timestep.

```python
# Record the number of doses each person has received each day so we can plot the rollout in this example. Without a custom analyzer, only the total number of doses will be recorded.

n_doses = []

# Define sequence based vaccination
pfizer = cv.vaccinate_num(vaccine='pfizer', sequence=prioritize_by_age, num_doses=100)

sim = cv.Sim(
    pars=pars,
    interventions=pfizer,
    analyzers=lambda sim: n_doses.append(sim.people.doses.copy())
)
sim.run()

pl.figure()

n_doses = np.array(n_doses)

fully_vaccinated = (n_doses == 2).sum(axis=1)

first_dose = (n_doses == 1).sum(axis=1)

pl.stackplot(sim.tvec, first_dose, fully_vaccinated)

pl.legend(['First dose', 'Fully vaccinated'])
```

Notice how second doses are prioritized, so after 21 days, there is a backlog of people requiring second doses, so the first doses are suspended until all of the second doses have been delivered. In reality, the pace of vaccination typically increases following the commencement of vaccination, so capacity increases over time. The `doses_per_day` argu-
ment allows this increase to be captured. There are several ways to specify a time-varying dose, including a date-based
dictionary to facilitate calibration when the number of doses each day is known. A simple option is to use a function
that returns the number of doses to distribute based on the `cv.Sim` - for example:

```
[6]: def num_doses(sim):
    if sim.t < 50:
        return sim.t*10
    else:
        return 500
```

This function corresponds to the vaccination rate increasing linearly for the first 50 days, before then stabilizing. The
function can be passed directly into the `cv.vaccinate_num` intervention:

```
[7]: n_doses = []
pfizer = cv.vaccinate_num(vaccine='pfizer', sequence=prioritize_by_age, num_doses=num_doses)
    sim = cv.Sim(
        pars=pars,
        interventions=pfizer,
        analyzers=lambda sim: n_doses.append(sim.people.doses.copy())
    )
sim.run()
```

```
pl.figure()
n_doses = np.array(n_doses)
fully_vaccinated = (n_doses == 2).sum(axis=1)
first_dose = (n_doses == 1).sum(axis=1)
pl.stackplot(sim.tvec, first_dose, fully_vaccinated)
pl.legend(['First dose', 'Fully vaccinated']);
```

Now the increase in capacity means that when second doses are due, there are sufficient additional doses available to
continue distributing first doses. Further customization, particularly to customize second dose prioritization depending on the specific policies implemented in a particular setting, can be readily achieved by implementing a new class deriving from `cv.BaseVaccination` in exactly the same way `cv.vaccinate_prob` and `cv.vaccinate_num` are implemented.

**Boosters**

By default, the vaccination interventions in Covasim are targeted towards unvaccinated people. If you want to include boosters in your simulation, you can use the `booster=True` argument when calling `cv.vaccinate`. You also need to use the `subtarget` argument so that we know who is going to receive a booster shot:

```python
# Define the number of boosters
def num_boosters(sim):
    if sim.t < 50:  # None over the first 50 days
        return 0
    else:
        return 50  # Then 100/day thereafter

# Only give boosters to people who have had 2 doses
booster_target = {'inds': lambda sim: cv.true(sim.people.doses != 2), 'vals': 0}
booster = cv.vaccinate_num(vaccine='pfizer', sequence=prioritize_by_age, ...
                          subtarget=booster_target, booster=True, num_doses=num_boosters)

# Track doses
n_doses = []
n_doses_boosters = []

# Create a sim with boosters
sim_booster = cv.Sim(
    pars = pars,
    interventions = [pfizer, booster],
    label = 'With booster',
    analyzers = lambda sim: n_doses_boosters.append(sim.people.doses.copy())
)
sim_booster.run()

# Plot the sims with and without boosters together
cv.MultiSim([sim, sim_booster]).plot(to_plot=['cum_infections', 'cum_severe', 'cum...
                        'deaths','pop_nabs'])

# Plot doses again
pl.figure()
n_doses = np.array(n_doses_boosters)
fully_vaccinated = (n_doses == 2).sum(axis=1)
first_dose = (n_doses == 1).sum(axis=1)
boosted = (n_doses > 2).sum(axis=1)
pl.stackplot(sim.tvec, first_dose, fully_vaccinated, boosted)
pl.legend(["First dose", 'Fully vaccinated', 'Boosted'], loc='upper left');
In this example there isn’t a large difference in the epidemic dynamics, because over a short run-time (120 days) immunity doesn’t wane very much, so boosters don’t have a large effect. However, this example can be adapted to longer run-times, where boosters have a larger impact.

**Prior immunity methods**

The longer that the COVID-19 pandemic persists, the less feasible/desirable it becomes to simulate the entire ~2 year history of the epidemic. However, starting simulations 1-2 years into the pandemic is also not ideal, as this will fail to capture the pre-existing immunity profile of the population. A good alternative is to use the `cv.prior_immunity` method. The following example starts a simulation in which 50% of the population are assumed to be vaccinated with the Pfizer vaccine 360 days before the simulation begins:

```python
[9]: pfizer = cv.historical_vaccinate_prob(vaccine='pfizer', days=[-360], prob=0.5)
    sim = cv.Sim(
        n_days=1,
        interventions=pfizer,
        analyzers=cv.nab_histogram(days=[0], edges=np.linspace(-4,2,12+1))
    )
    sim.run()
    sim.get_analyzer().plot()
```
The histogram of neutralizing antibody levels after 1 day of simulation shows that there are already significant levels of immunity in the population.

This next example will initialize the population with immunity levels corresponding to a prior wave of infections peaking 120 days before the simulation starts, with 5% of the population having been infected:
1.2.9 T9 - Deployment

This tutorial provides several useful recipes for deploying Covasim.

Click here to open an interactive version of this notebook.

1.2.9.1 Dask

Dask is a powerful library for multiprocessing and “scalable” analytics. Using Dask (rather than the built-in multiprocessing) for parallelization is relatively straightforward:

```python
import dask
from dask.distributed import Client
import numpy as np
```
import covasim as cv

def run_sim(index, beta):
    ''' Run a standard simulation '''
    sim = cv.Sim(beta=beta, label=f'Sim {index}, beta={beta}')
    sim.run()
    return sim

if __name__ == '__main__':

    # Run settings
    n = 8
    n_workers = 4
    betas = np.sort(np.random.random(n))

    # Create and queue the Dask jobs
    client = Client(n_workers=n_workers)
    queued = []
    for i, beta in enumerate(betas):
        run = dask.delayed(run_sim)(i, beta)
        queued.append(run)

    # Run and process the simulations
    sims = list(dask.compute(*queued))
    msim = cv.MultiSim(sims)
    msim.plot(color_by_sim=True)

1.2.9.2 Jupyter/IPython

Using Jupyter and Voilà, you can build a Covasim-based webapp in minutes. First, install the required dependencies:

```bash
pip install jupyter jupyterlab jupyterhub ipympl voila
```

Here is a very simple interactive webapp that runs a multisim (in parallel!) when the button is pressed, and displays the results:

```python
import numpy as np
import covasim as cv
import ipywidgets as widgets

# Create the button and output area
button = widgets.Button(description='Run')
output = widgets.Output()

@output.capture()
def run():
    ''' Stochastically run a parallelized multisim '''
    sim = cv.Sim(verbos=0, pop_size=20e3, n_days=100, rand_seed=np.random.randint(99))
    msim = cv.MultiSim(sim)
    msim.run(n_runs=4)
    return msim.plot()
```

(continues on next page)
def click(b):
    ''' Rerun on click '''
    output.clear_output(wait=True)
    run()

# Create and show the app
button.on_click(click)
app = widgets.VBox([button, output])
display(app)

If you save this as e.g. msim.ipynb, then you can turn it into a web server simply by typing voila msim.ipynb.

1.2.10 T10 - Tips and tricks

This tutorial contains suggestions that aren’t essential to follow, but which may make your life easier.

Click here to open an interactive version of this notebook.

1.2.10.1 Versioning

Covasim contains a number of built-in tools to make it easier to keep track of where results came from. The simplest of these is that if you save an image using cv.savefig() instead of pl.savefig(), it will automatically store information about the script and Covasim version that generated it:

[1]:

```python
import covasim as cv
cv.options.set('jupyter', verbose=0)

sim = cv.Sim()
sim.run()
sim.plot()
```
This can be extremely useful for figuring out where that intriguing result you generated 3 weeks ago came from!

This information is also stored in sims and multisims themselves:
Finally, the function `cv.check_version()` and `cv.check_save_version()` are useful if you want to ensure that users are running the right version of your code. Placing `cv.check_save_version('2.0.0')` will save a file with the information above to the current folder – again, useful for debugging exactly what changed and when. (You can also provide additional information to it, e.g. to also save the versions of 3rd-party packages you’re importing). `cv.check_version()` by itself can be used to provide a warning or even raise an exception (if `die=True`) if the version is not what’s expected:

```
[4]: cv.check_version('1.5.0')
Note: Covasim is newer than expected (3.1.1 vs. 1.5.0)
[4]: 1
```

### 1.2.10.2 Working with dates

Dates can be tricky to work with. Covasim comes with a number of built-in features to work with dates. By default, by convention Covasim works with dates in the format `YYYY-MM-DD`, e.g. `'2020-12-01'`. However, it can handle a wide variety of other date and `datetime` objects. In particular, `sim` objects know when they start and end, and can use this to do quite a bit of date math:

```
[5]: sim = cv.Sim(start_day='20201122', end_day='2020-12-09 02:14:58.727703')
sim.initialize()  # Date conversion happens on initialization
print(sim['start_day'])
print(sim['end_day'])
print(sim.day(sim['end_day']))  # Prints the number of days until the end day, i.e. the length of the sim
2020-11-22
2020-12-09
17
```

You can also easily calculate the difference between two dates, or generate a range of dates. These are returned as strings by default, but can be converted to `datetime` objects via Sciris:

```
[6]: import sciris as sc
print(cv.daydiff('2020-06-01', '2020-07-01', '2020-08-01'))
dates = cv.date_range('2020-04-04', '2020-04-12')
print(dates)
print(sc.readdate(dates))
[30, 31]
```

(continues on next page)
Finally, one gotcha is that when loading Excel spreadsheets in pandas, dates are loaded in pandas' internal Timestamp[ns64] format, which nothing else seems to be able to read. If this happens to you, the solution (as far as Covasim is concerned) is to convert to a datetime.date:

```python
data = pd.read_excel(filename)
data['date'] = data['date'].dt.date
```

### 1.2.10.3 Working with dictionaries

“I already know how to work with dictionaries”, you say. Yes, you do. But there are a couple tricks that might make things easier.

Covasim is built on Sciris, which includes containers `odict` and `objdict`. While these are documented elsewhere, a couple examples will serve to illustrate how they work.

An odict is just an ordered dict that you can refer to by position as well as by key. For example:

```python
mydict = sc.odict(foo=[1,2,3], bar=[4,5,6])  # Assignment is the same as ordinary _dictionaries
print('Entry foo:', mydict['foo'])
print('Entry 0:', mydict[0])  # Access by key or by index
for i,key,value in mydict.enumitems():  # Additional methods for iteration
    print(f'Item {i} is named {key} and has value {value}')
```

Entry foo: [1, 2, 3]
Entry 0: [1, 2, 3]
Item 0 is named foo and has value [1, 2, 3]
Item 1 is named bar and has value [4, 5, 6]

An objdict is exactly the same as an odict except it lets you reference keys as if they were attributes:

```python
myobjdict = sc.objdict(foo=[1,2,3], bar=[4,5,6])
print('Entry foo:', myobjdict['foo'])
print('Entry 0:', myobjdict[0])  # Access by key or by index
print('"Attribute" foo:', myobjdict.foo)
```

Entry foo: [1, 2, 3]
Entry 0: [1, 2, 3]
"Attribute" foo: [1, 2, 3]

Using this approach, you can get all the power and flexibility of dictionaries, while writing code as succinctly as possible. For example:

```python
total_pop = 44_483  # This many total people
pars= sc.objdict(
    pop_type = 'hybrid',
    pop_size = 10e3,
)
pars.pop_scale = total_pop/pars.pop_size  # Instead of pars['pop_scale'] = total_pop/
    -pars['pop_size']
sim = cv.Sim(**pars)  # It's still a dict, so you can treat it as one!
```

For example, the `results` object is an objdict. This means that although you can use e.g. `sim.results['new_infections']`, you can also use `sim.results.new_infections`.  

---

**Covasim, Release 3.1.1**

(continued from previous page)
Finally, Sciris also contains a function called `mergedicts`. This acts very similarly to `dict.update()`, with the main difference being that it returns the result of merging the two dictionaries. This is especially useful for handling keyword arguments in functions:

```python
[10]: def myfunc(args=None, **kwargs):
    defaults = dict(foo=[1,2,3], bar=[4,5,6])
    merged_args = sc.mergedicts(defaults, args, kwargs)
    print(merged_args)
myfunc(args=dict(bar=18), other_args='can be anything')
{'foo': [1, 2, 3], 'bar': 18, 'other_args': 'can be anything'}
```

As you can see, it merged the default settings, the arguments supplied to the function via the keyword `args`, and then other keywords, into a single dictionary.

### 1.2.11 T11 - Advanced features

This tutorial covers advanced features of Covasim, including custom population options and changing the internal computational methods.

Click here to open an interactive version of this notebook.

#### 1.2.11.1 Defining populations with SynthPops

For complex populations, we suggest using SynthPops, a Python library designed specifically for this purpose. In contrast to the population methods built-in to Covasim, SynthPops uses data to produce synthetic populations that are statistically indistinguishable from real ones. For a relatively complex example of how SynthPops was used to create a complex school network for the Seattle region, see here.

#### 1.2.11.2 Defining contact layers

As mentioned in Tutorial 1, contact layers are the graph connecting the people in the simulation. Each person is a node, and each contact is an edge. While enormous complexity can be used to define realistic contact networks, a reasonable approximation in many cases is random connectivity, often with some age assortativity. Here is an example for generating a new contact layer, nominally representing public transportation, and adding it to a simulation:

```python
[1]: import numpy as np
   import covasim as cv
   cv.options.set('jupyter', verbose=0)
   # Create the first sim
   orig_sim = cv.Sim(pop_type='hybrid', n_days=120, label='Default hybrid population')
   orig_sim.initialize()  # Initialize the population
   # Create the second sim
   sim = orig_sim.copy()
   # Define the new layer, 'transport'
   n_people = len(sim.people)
   n_contacts_per_person = 0.5
   n_contacts = int(n_contacts_per_person*n_people)
```

(continues on next page)
contacts_p1 = cv.choose(max_n=n_people, n=n_contacts)
contacts_p2 = cv.choose(max_n=n_people, n=n_contacts)
beta = np.ones(n_contacts)
layer = cv.Layer(p1=contacts_p1, p2=contacts_p2, beta=beta)  # Create the new layer

# Add this layer in and re-initialize the sim
sim.people.contacts.add_layer(transport=layer)
sim.reset_layer_pars()  # Automatically add layer 'q' to the parameters using default values
sim.initialize()  # Reinitialize
sim.label = f'Transport layer with {n_contacts_per_person} contacts/person'

# Run and compare
msim = cv.parallel(orig_sim, sim)
msim.plot()
1.2.11.3 Defining dynamic layers

You can also define custom layers that update dynamically, e.g. based on a supplied number of contacts per day. To do this, create a new `Layer` class and define the `update()` method. For example:

```python
import covasim as cv
import numpy as np
import sciris as sc

class CustomLayer(cv.Layer):
    ''' Create a custom layer that updates daily based on supplied contacts '''

    def __init__(self, layer, contact_data):
        ''' Convert an existing layer to a custom layer and store contact data '''
        for k,v in layer.items():
            self[k] = v
        self.contact_data = contact_data
        return

    def update(self, people):
        ''' Update the contacts '''
        pop_size = len(people)
        n_new = self.contact_data[people.t]  # Pull out today's contacts
        self['p1'] = np.array(cv.choose_r(max_n=pop_size, n=n_new), dtype=cv.default_int)  # Choose with replacement
        self['p2'] = np.array(cv.choose_r(max_n=pop_size, n=n_new), dtype=cv.default_int)  # Paired contact
        self['beta'] = np.ones(n_new, dtype=cv.default_float)  # Per-contact transmission (just 1.0)
        return

# Define some simple parameters
pars = sc.objdict(
    pop_size = 1000,
    n_days = 90,
)

# Set up and run the simulation
base_sim = cv.Sim(pars, label='Default simulation')
sim = cv.Sim(pars, dynam_layer=True, label='Dynamic layers')
sim.initialize()

# Update to custom layer
for key in sim.layer_keys():
    contact_data = np.random.randint(pars.pop_size*10, pars.pop_size*20, size=pars.n_days+1)  # Generate a number of contacts for today
    sim.people.contacts[key] = CustomLayer(sim.people.contacts[key], contact_data)

# Run and plot
msim = cv.parallel(base_sim, sim)
msim.plot()
```
1.2.11.4 Defining custom population properties

Another useful feature is adding additional features to people, for use in subtargeting. For example, this example shows how to define a subpopulation with higher baseline mortality rates. This is a simple example illustrating how you would identify and target people based on whether or not the have a prime-number index, based on the protecting the elderly example from Tutorial 1.

```python
[3]:
import numpy as np
import sciris as sc
import covasim as cv

def protect_the_prime(sim):
    if sim.t == sim.day('2020-04-01'):
        are_prime = sim.people.prime
        sim.people.rel_sus[are_prime] = 0.0

pars = dict(
    pop_type = 'hybrid',
    pop_infected = 100,
)

(continues on next page)
n_days = 90,
verbose = 0,
)

# Default simulation
orig_sim = cv.Sim(pars, label='Default')

# Create the simulation
sim = cv.Sim(pars, label='Protect the prime', interventions=protect_the_prime)
sim.initialize()

# Define whom to target
sim.people.prime = np.array([sc.isprime(i) for i in range(len(sim.people))])

# Run and plot
msim = cv.parallel(orig_sim, sim)
msim.plot()
1.2.11.5 Changing Numba options

Finally, this example shows how you can change the default Numba calculation options. It’s not recommended – especially running with multithreading, which is faster but gives stochastically unreproducible results – but it’s there if you want it.

```python
import covasim as cv

# Create a standard 32-bit simulation
sim32 = cv.Sim(label='32-bit, single-threaded (default)', verbose='brief')
sim32.run()

# Use 64-bit instead of 32
precision=64
sim64 = cv.Sim(label='64-bit, single-threaded', verbose='brief')
sim64.run()

# Use parallel threading
numba_parallel=True
sim_par = cv.Sim(label='64-bit, multi-threaded', verbose='brief')
sim_par.run()

# Reset to defaults
sim32b = cv.Sim(label='32-bit, single-threaded (restored)', verbose='brief')
sim32b.run()

# Plot
msim = cv.MultiSim([sim32, sim64, sim_par, sim32b])
msim.plot()
```

Sim("32-bit, single-threaded (default)"; 2020-03-01 to 2020-04-30; pop: 20000 random; epi: 12730, 14)
Reloading Covasim so changes take effect...
Reload complete. Note: for some options to take effect, you may also need to delete Covasim's __pycache__ folder.
Sim("64-bit, single-threaded"; 2020-03-01 to 2020-04-30; pop: 20000 random; epi: 13209, 16)
Reloading Covasim so changes take effect...
Reload complete. Note: for some options to take effect, you may also need to delete Covasim's __pycache__ folder.
Sim("64-bit, multi-threaded"; 2020-03-01 to 2020-04-30; pop: 20000 random; epi: 13209, 16)
Reloading Covasim so changes take effect...
Covasim 3.1.1 (2021-12-06) -- © 2021 by IDM
Reload complete. Note: for some options to take effect, you may also need to delete Covasim's __pycache__ folder.
Sim("32-bit, single-threaded (restored)"; 2020-03-01 to 2020-04-30; pop: 20000 random; epi: 12730, 14)
1.3 FAQ

This document contains answers to frequently (and some not so frequently) asked questions. If there are others you’d like to see included, please email us at covasim@idmod.org.

Contents

- Usage questions
  - What are the system requirements for Covasim?
  - Can Covasim be run on HPC clusters?
  - What method is best for saving simulation objects?
  - Does Covasim support time-dependent parameters?
  - How can you introduce new infections into a simulation?
  - How can I find out what happened to a particular individual during the simulation?
How do you set custom prognoses parameters (mortality rate, susceptibility etc.)?

I want to generate a contact network for <insert location here>. How do I do this?

Is it possible to model interacting geographical regions?

Is it possible to model births, non-COVID deaths, or migration?

I really don’t like Python, can I run Covasim via R?

• Conceptual questions
  
  What are the relationships between population size, number of agents, population scaling, and total population?
  
  Are test results counted from swab date or result date?
  
  Is the underlying model capable of generating oscillations?
  
  Why doesn’t anyone start off as infectious?
  
  What are the valid states an agent can be in?

• Common problems
  
  I’m getting different results to someone else, or to what I got previously, with the same parameters. Why?
  
  Why doesn’t the webapp accept long durations or large population sizes?
  
  Why do parallel simulations fail on Windows or in Jupyter notebooks?

1.3.1 Usage questions

1.3.1.1 What are the system requirements for Covasim?

If your system can run scientific Python (Numpy, SciPy, and Matplotlib), then you can probably run Covasim. Covasim requires 1 GB of RAM per 1 million people, and can simulate roughly 5-10 million person-days per second. A typical use case, such as a population of 100,000 agents running for 500 days, would require 100 MB of memory and take about 5-10 seconds to run.

1.3.1.2 Can Covasim be run on HPC clusters?

Yes. On a single-node setup, it is quite easy: in fact, `MultiSim` objects will automatically scale to the number of cores available. This can also be specified explicitly with `e.g. msim.run(n_cpus=24)`.

For more complex use cases (e.g. running across multiple virtual machines), we recommend using `Celery`; please email us for more information.

1.3.1.3 What method is best for saving simulation objects?

The recommended way to save a simulation is simply via `sim.save(filename)`. By default, this does not save the people (`sim.people`), since they are very large (i.e., 7 KB without people vs. 7 MB with people for 100,000 agents). However, if you really want to save the people, pass `keep_people=True`.

To load, you can use `cv.load(filename)` or `cv.Sim.load(filename)` (they are identical except for type checking). Under the hood, Covasim uses `sc.saveobj()` from `Sciris`, which in turn is a gzipped pickle. If you
need a more “portable” format, you can use `sim.to_json()`, but note that this will only save the results and the parameters, not the full sim object.

1.3.1.4 Does Covasim support time-dependent parameters?

Typically, parameters are held constant for the duration of the simulation. However, it is possible to modify them dynamically – see the `cv.dynamic_pars()` “intervention”.

1.3.1.5 How can you introduce new infections into a simulation?

These are referred to as *importations*. You can set the `n_imports` parameter for a fixed number of importations each day (or make it time-varying with `cv.dynamic_pars()`, as described above). Alternatively, you can infect people directly using `sim.people.infect()`. Since version 3.0, you can also import specific strains on a given day: e.g., `cv.Sim(strains=cv.strain('b117', days=50, n_imports=10)`.

1.3.1.6 How can I find out what happened to a particular individual during the simulation?

This can be done using the `story` method after the simulation has been run:

```python
import covasim as cv
sim = cv.Sim()
sim.people.story(12)
```

1.3.1.7 How do you set custom prognoses parameters (mortality rate, susceptibility etc.)?

Most parameters can be set quite simply, e.g.:

```python
import covasim as cv
sim = cv.Sim(beta=0.008)
```

or:

```python
import covasim as cv
pars = dict(beta=0.008, verbose=0)
sim = cv.Sim(pars)
```

However, prognoses parameters are a bit different since they’re a dictionary of dictionaries of arrays. Usually the easiest solution is to create the simulation first, and then modify these parameters before initializing the sim:

```python
import covasim as cv
sim = cv.Sim()
sim['prognoses']['death_probs'][-1] *= 2  # Double the risk of death in the oldest age group
```

Another option is to create the parameters first, then modify them and provide them to the sim:

```python
import covasim as cv
prognoses = cv.get_prognoses()
prognoses['death_probs'][-1] *= 2
sim = cv.Sim(prognoses=prognoses)
```
One thing to be careful of is that since the prognoses are used when the population properties are set, you must make any changes to them before you initialize the sim (i.e. \texttt{sim.initialize()}). If you want to change prognoses for an already-created simulation, it is best to call \texttt{sim.init_people()} to ensure the sim parameters (\texttt{sim.pars}) are synchronized with the people parameters (\texttt{sim.people.pars}).

1.3.1.8 I want to generate a contact network for <insert location here>. How do I do this?

There are a few options. For many cases, the default options work reasonably well, i.e. \texttt{sim = cv.Sim(pop_type='hybrid', location='eswatini')}). If you want to use location that is not currently supported, there is generally a lot of data required (census data, school enrolment rates, workplace size and participation rates, etc.). Detailed contact networks are generally created using the \texttt{SynthPops} library.

Another option is to adapt the functions in \texttt{population.py} for your purposes. Covasim can also read in fairly generic representations of populations; for example you could create a random network and then modify the edge list (i.e. \texttt{sim.peoplecontacts}) to reflect the network you want. Please email us for more information.

1.3.1.9 Is it possible to model interacting geographical regions?

Possible, but not easy. Your best option is to create a single simulation where the contact network structure reflects the different regions. Please email us for more information.

1.3.1.10 Is it possible to model births, non-COVID deaths, or migration?

Not currently, but what you can do is create the final population size, and then remove all contacts for the people you want to be “absent”. In terms of transmission, a susceptible person who is unconnected to anyone else is effectively identical to a person who doesn’t exist. You can then “create” these people by adding or restoring their contacts to other people. However, be mindful that results that are population averages (e.g. prevalence) will be wrong since they will include these inactive people, and interventions (such as \texttt{cv.test_num()}) will also apply to these people.

1.3.1.11 I really don’t like Python, can I run Covasim via R?

Actually, you can! R’s \texttt{reticulate} package lets you easily interface between Python and R. For example:

```r
library(reticulate)
cv <- import('covasim')
sim <- cv$Sim()
sim$run()
sim$plot()
```

(NB: if the above doesn’t bring up a figure, try adding \texttt{plt <- import('matplotlib.pyplot')} and \texttt{plt$show()}.)

1.3.2 Conceptual questions

1.3.2.1 What are the relationships between population size, number of agents, population scaling, and total population?

The terms are a bit confusing and may be refactored in a future version of Covasim. The \texttt{pop_size} parameter actually controls the number of \texttt{agents} in the simulation (note: you can also use the parameter \texttt{n_agents}, it’s just an alias for \texttt{pop_size}). In many cases the number of agents is the same as the “total population size” or “scaled population size” being simulated, i.e., the actual number of people. The “actual number of people” (not agents) is
available in the simulation as `sim.scaled_pop_size`. If `pop_scale` is greater than 1, the total population size will be greater than the number of agents. You can also set the `scaled_pop` parameter (which is the total population size), in which case `pop_scale` will be calculated automatically. Some examples might help make this clearer:

**Example 1.** You want to simulate a population of 100,000 people. This will only take a few seconds to run, so you set `pop_size = 100e3` and `pop_scale = 1`. In this example the population size is 100,000, the scaled population size is 100,000, the number of agents is 100,000, and the number of people being represented is also 100,000. Life is simple and you are happy.

**Example 2.** You want to simulate a population of 1,000,000 people. This would take too long to run easily (several minutes per run), so you set `pop_size = 200e3` and `pop_scale = 5` with dynamic rescaling on (rescale = True). (Note: this is exactly equivalent to setting `n_agents = 200e3` and `scaled_pop = 1e6`, in which case `pop_scale` will be automatically set to 5.) In this example the (simulated) population size is 200,000, the (final) scaled population size is 1,000,000, the number of agents is always 200,000, and the (final) number of people being represented is 1,000,000. Since dynamic rescaling is on, when the simulation starts, one agent represents one person, but only 200,000 people are included in the simulation (the other 800,000 are not infected and are not exposed to anyone who is infected, so are not represented in the sim). As more and more people become infected – say, 10,000 infections – 200,000 people is no longer enough to accurately represent the epidemic; since 10,000 infections out of 200,000 people is prevalence of 5%, whereas the real prevalence is 1% (10,000 infections out of 1,000,000 people). Dynamic rescaling kicks in (rescale_threshold = 0.05, the current prevalence level), and half of the infected people are converted back to susceptibles (rescale_factor = 2). There are now 5,000 infected agents in the model, corresponding to 10,000 infected people, i.e. one agent now counts as (represents) two people. This is equivalent to saying that for any given agent in the model (e.g., an infected 57-year-old woman who has 2 household contacts and 8 workplace contacts), there is another identical person somewhere else in the population.

**Example 3.** As in example 2, but you turn dynamic rescaling off (rescale = False). In this case, from the very beginning of the simulation, one agent represents 5 people (since `pop_scale = 5`). This is basically the same as running a simulation of 200,000 agents with `pop_scale = 1` and then multiplying the results (e.g., cumulative number of infections) by a factor of 5 after the simulation finishes running: each infection counts as 5 infections, each death counts as 5 deaths, etc. Note that with dynamic rescaling off, the number of seed infections should be divided by `pop_scale` in order to give the same results.

**TLDR?** Except for a few corner cases (e.g., calculating transmission trees), you should get nearly identical results with and without dynamic rescaling, so feel free to use it (it’s turned on by default). That said, it’s always best to use as small of a population scale factor as you can, although once you reach roughly 200,000 agents, using more agents shouldn’t make much difference.

This example illustrates the three different ways to simulate a population of 100,000 people:

```python
import covasim as cv

s1 = cv.Sim(n_days=120, pop_size=200e3, pop_infected=50, pop_scale=1, rescale=True, label='Full population')
s2 = cv.Sim(n_days=120, pop_size=20e3, pop_infected=50, pop_scale=10, rescale=True, label='Dynamic rescaling')
s3 = cv.Sim(n_days=120, pop_size=20e3, pop_infected=5, pop_scale=10, rescale=False, label='Static rescaling')

msim = cv.MultiSim([s1, s2, s3])
msim.run(verbosity=-1)
msim.plot()
```

Note that using the full population and using dynamic rescaling give virtually identical results, whereas static scaling gives slightly different results.
1.3.2.2 Are test results counted from swab date or result date?

The results are reported for the date of the test which came back positive, not the the date of diagnosis. This reason for this is that in most places, this is how the data are reported – if they do 100 tests on August 1st, say, and there is a 2-4 day test delay so 5 of these tests come back positive on each of August 2nd, 3rd, 4th, then in most places, this would be reported as 100 tests on August 1st, 15 diagnoses on August 1st (even though the lab work was done over August 2-4), and 85 negative tests on August 1st. The reason for doing it this way – both in real world reporting and in the model – is because otherwise you have a situation where if there is a big change in the number of tests from day to day, you could have more diagnoses on that day than tests. However, in terms of the model, the test delay is still being correctly taken into account. Specifically, `sim.people.date_pos_test` is used to (temporarily) store the date of the positive test, which is what’s shown in the plots, but `sim.people.date_diagnosed` has the correct (true) diagnosis date for each person. For example:

```python
import covasim as cv
tn = cv.test_num(daily_tests=100, start_day=10, test_delay=10)
sim = cv.Sim(interventions=tn)
sim.run()
sim.plot(to_plot=['new_infections', 'new_tests', 'new_diagnoses'])
```

shows that positive tests start coming back on day 10 (the start day of the intervention), but:

```python
>>> np.nanmin(sim.people.date_diagnosed)
20.0
```

shows that the earliest date a person is actually diagnosed is on day 20 (the start day of the intervention plus the test delay).

1.3.2.3 Is the underlying model capable of generating oscillations?

Yes, although oscillatory modes are not a natural state of the system – you can get them with a combination of high infection rates, low testing rates, and high contact tracing rates with significant delays. This will create little clusters that grow stochastically until someone gets tested, then most of the cluster gets traced and shut down, but a few people usually escape to start the next cluster.

1.3.2.4 Why doesn’t anyone start off as infectious?

If you run a simple simulation (e.g. `sim = cv.Sim().run()`), you might notice that no one starts off as infectious (`sim.results.n_infectious[0]` is 0). This is because they all start in the just-infected or “exposed” state, when they’re not infectious yet (it takes a few days for someone to become infectious, determined by the parameter `exp2inf`, exposed to infectious duration). This corresponds to the “E” compartment of an SEIR model.

1.3.2.5 What are the valid states an agent can be in?

This table shows what combination of the states an agent can be in. Blue indicates that if an agent is in state A, they must also be in state B; orange shows that if an agent is in state A, they cannot be in state B; green shows that if an agent is in state A, they may or may not be in state B.
1.3.3 Common problems

1.3.3.1 I’m getting different results to someone else, or to what I got previously, with the same parameters. Why?

One of the trickiest aspects of working with agent-based models is getting the random number stream right. Covasim uses both numpy and numba random number streams. These are usually initialized automatically when a simulation is created/run (via `cv.set_seed(seed)`), which you can call directly as well), but anything that disrupts the random number stream will result in differences between two simulation runs. This is also why seemingly trivial changes (e.g., adding an intervention that doesn’t actually do anything) can cause simulation trajectories to diverge.

In addition, random number streams sometimes change with different library versions. For example, due to a bugfix, random number streams changed between numba 0.48 and 0.49. Therefore, simulation run with numba 0.48 or earlier won’t (exactly) match simulations run with numba 0.49 or later.

If you’re having trouble reproducing results between simulations that should be the same, check: (a) the Covasim version, (b) the numpy version, (c) the numba version, and (d) the SynthPops version (if using). If all these match but results still differ, then a useful debugging strategy can be to insert `print(np.random.rand())` at various points throughout the code to see at what point the two versions diverge.

1.3.3.2 Why doesn’t the webapp accept long durations or large population sizes?

The webapp is limited by the results needing to be returned before the request times out. However, when running directly via Python, you are limited only by your computer’s RAM (and your patience) in terms of simulation duration or population size.

1.3.3.3 Why do parallel simulations fail on Windows or in Jupyter notebooks?

If you are running on Windows, because of the way Python’s multiprocessing library is implemented, you must start the run from inside a `__main__` block (see discussion here). For example, instead of this:

```python
import covasim as cv
sims = [cv.Sim(pop_infected=100, beta=0.005*i, label=f'Beta factor {i}') for i in range(5)]
msim = cv.MultiSim(sims)
msim.run()
msim.plot()
```

do this:

```python
import covasim as cv
sims = [cv.Sim(pop_infected=100, beta=0.005*i, label=f'Beta factor {i}') for i in range(5)]
msim = cv.MultiSim(sims)

if __name__ == '__main__':
    msim.run()
    msim.plot()
```

When parallelizing inside Jupyter notebooks, sometimes a “Duplicate signature” error will be encountered. This is because of how multiprocessing conflicts with Jupyter’s internal threading (see discussion here). One solution is to move `msim.run()` (or other parallel command) to a separate .py file, and not have it be part of the notebook itself. This problem should be fixed in version 2.0 though, so if you’re using an older version, consider upgrading.
1.4 What’s new

All notable changes to the codebase are documented in this file. Changes that may result in differences in model output, or are required in order to run an old parameter set with the current version, are flagged with the term “Regression information”.

Contents

- Coming soon
- Latest versions (3.1.x)
  - Versions 3.0.x (3.0.0 – 3.0.7)
  - Versions 2.x (2.0.0 – 2.1.2)
  - Versions 1.7.x (1.7.0 – 1.7.6)
  - Versions 1.6.x (1.6.0 – 1.6.1)
  - Versions 1.5.x (1.5.0 – 1.5.3)
  - Versions 1.4.x (1.4.0 – 1.4.8)
  - Versions 1.3.x (1.3.0 – 1.3.5)
  - Versions 1.2.x (1.2.0 – 1.2.3)
  - Versions 1.1.x (1.1.0 – 1.1.7)
  - Versions 1.0.x (1.0.0 – 1.0.3)
- Prerelease versions (0.27.0 – 0.32.1)

1.4.1 Coming soon

These are the major improvements we are currently working on. If there is a specific bugfix or feature you would like to see, please create an issue.

- Additional nuance in how immunity is modeled (planned for v3.2)
- Multi-region and geographical support (planned for v3.3)
- Economics and costing analysis (planned for v3.4)

1.4.2 Latest versions (3.1.x)

1.4.2.1 Version 3.1.1 (2021-12-06)

Performance improvements

Performance improvements have been implemented in all aspects of Covasim, including:

- By changing certain imports to be just-in-time rather than up-front, module import time (import covasim as cv) was roughly halved (from about 0.7 s to 0.4 s).
- Population generation methods have been refactored; initializing a sim is now twice as fast (from about 0.4 s to 0.2 s for 20,000 people).
• Immunity and infection methods have been refactored, leading to about a 30% improvement in run time (from about 0.7 to 0.5 s for 20,000 people 60 days).

Bugfixes

• Fixed a bug in which sim.pars and sim.people.pars were not the same object. In almost all cases, the latter should now be a link to the former.

• Fixed a bug whereby interventions and analyzers were not being shrunk correctly.

• Fixed a bug with 2nd vaccine doses not being rescheduled if zero doses were given on the day they were scheduled.

Other updates

• A new convenience function has been added: cv.parallel(sim1, sim2) is equivalent to cv.MultiSim([sim1, sim2]).run().

• Calibrations now have multiple new analysis and plotting features, including calib.plot_trend(), calib.plot_best(), calib.to_json(), etc. calib.plot() has been renamed to calib.plot_sims().

• By default, calibrations now keep going if a sim encounters an exception; to restore the previous behavior, use sim.calibrate(die=True). A calibration run with a single worker now does not use multiprocess, to simplify debugging.

• There is a new option for changing the thousands separator (e.g. to use European formatting), via e.g. cv.options.set(sep='.'). This does not yet apply to plots, but will in a future version.

• A convenience method has been added for setting correct plot options for Jupyter: cv.options.set('jupyter').

• Population generation functions make_random_contacts() and make_microstructured_contacts() were updated to generate edgelists rather than lists-of-dicts.

• cv.poisson_test() was removed as as it was no longer being used.

• Tutorials, examples, and the FAQ have all been updated. In particular, all tutorials are now available to be run interactively with Binder via http://tutorials.covasim.org.

• Regression information: The new infection calculation method is mathematically identical but draws differently from the random number stream, giving stochastically different results than before. To revert to the previous (slower) calculation method, set sim._legacy_trans = True after initialization. This legacy option is automatically enabled if running with an earlier version of parameters, e.g. cv.Sim(version='3.1.0'). Calls to calib.plot() should be replaced with calib.plot_sims(). If you were using cv.poisson_test(), you’re on your own now, but are invited to copy it from an older version of Covasim.

• GitHub info: PR 1249

1.4.2.2 Version 3.1.0 (2021-12-03)

This version contains important updates to the parameters around immunity. It also introduces additional features designed to help with policy questions relevant at this stage of the pandemic, including support for boosters and the ability to initialize a population with pre-existing immunity. Although we will continue to update parameter values as new data come in, the immunity and vaccine features are now out of the beta stage and ready to use.
Covasim, Release 3.1.1

Highlights

- **New immunity parameters**: Waning immunity and cross-immunity functions have been updated to match currently available empirical data.

- **Additional flexibility with vaccines**: Several new vaccines have been added (e.g. Sinopharm), and additional options have been provided to enable booster doses, simplify age targeting, etc.

- **Historical immunity**: To avoid the need to calibration to past epidemic waves and vaccine rollouts, new interventions have been added that let you control immunity levels from historical events.

Immunity-related parameter changes

- By default, simulations now use full immune dynamics (i.e. use_waning=True).

- When NAbs are primed, they are normalized to be equivalent to “vaccine NAbs”. This is done so that when we check immunity, we can calculate immune protection using a single curve and account for multiple sources of immunity (vaccine and natural).

- Antibody kinetics were adjusted based on recent observational data suggesting a faster decay of NAbs and subsequent protection against infection. Source: https://www.thelancet.com/journals/lancet/article/PIIS0140-6736(21)02183-8/fulltext

- A parameter trans_redux has been added to capture the reduction in transmission for breakthrough infections.

- Cross-immunity parameters have been updated.

- Default variant names now follow WHO conventions, e.g. 'alpha' rather than 'b117'. (The other names can still be used, however.)

- 'sinopharm' and 'sinovac' have been added as built-in vaccines. Aliases have been added for other vaccines (e.g. 'spikevax' for Moderna).

- Vaccine interventions now support booster doses.

- Age-targeting of vaccines can now be specified as e.g. sequence='age'.

Changes to states and results

- people.vaccinations has been renamed to people.doses, and keeps track of how many doses of any vaccine each agent has had. Likewise, new_vaccinations and cum_vaccinations have been renamed new_doses and cum_doses.

- People have a new state, n_breakthroughs, which tracks how many breakthrough infections they’ve had.

- NAb states have been updated: prior_symptoms has been removed and t_nab_event (the time when they were infected or vaccinated) has been added.

- A new result, n_imports, has been added, which counts the number of imported infections (including from variants).

New functions, methods and classes

- Added three new interventions designed to initiate a population with some prior immunity. The class cv.prior_immunity() is a wrapper for two options, cv.historical_vaccinate_prob() and cv.historical_wave().
• `cv.historical_vaccinate_prob()` allocates vaccines parametrized by the daily probability of being vaccinated. Unlike `cv.vaccinate_prob()`, this function allows vaccination prior to \( t=0 \) (and continuing into the simulation).

• `cv.historical_wave()` imprints a historical (pre \( t=0 \)) wave of infections in the population NAbs.

• A new analyzer, `cv.nab_histogram()`, allows easy computation of statistics relating to NAbs.

Bugfixes

• Keyword arguments to `cv.Fit()` are now correctly passed to `cv.compute_gof()`. (Thanks to Zishu Liu for finding this bug.)

• The transmission tree can now be exported using the latest version of NetworkX. (Thanks to Alexander Zarebski for finding this bug.)

• The \( r_{eff} \) calculation method has been updated to avoid divide-by-zero issues.

• Rescaling now does not reset vaccination status; previously, dynamic rescaling erased it.

• Previously, `cv.clip_edges()` and `cv.vaccinate_prob()` used a lot of memory; these “memory leaks” have been fixed with new `finalize()` methods.

• Some results (e.g. number of tests) were being incorrectly rounded to integers prior to rescaling; this has been fixed.

• Imported infections are now sampled without replacement.

• Scenarios now re-initialize the sim object. The scenario label now matches the scenario name rather than key.

Other changes

• Result fields can now be accessed as keys as well as attributes, e.g. any combination of `msim.results[\'r_eff\'][\'high\']` and `msim.results.r_eff.high` works.

• Interventions and analyzers now have a `shrink()` method, for cleaning up memory-hungry intermediate results at the end of a simulation.

• By default, calibration now removes the database of individual trials. Set `keep_db=True` to keep it. There is also a `remove_db()` method to manually remove the database.

• Population creation methods have been updated to be more flexible, with keyword arguments being passed to helper functions.

• Simulation summaries now by default use comma-separated values. To change this to e.g. a dot, you can set a global option: `cv.options.set(sep='.'), or e.g. `sim.summarize(sep='')`.

• `cv.diff_sims()` can now optionally skip specific results using the `skip` keyword.

• Vaccination is now included in the regression tests.

Regression information

• Results for simulations with `use_waning=True` will be substantially different due to the update in parameters and functional form.

• \( r_{eff} \) results will not match previous versions due to the change in calculation method (but differences should be slight).
Simulations that have been saved to disk which include variants may not work correctly. If this is an issue, please email us and we can help write a migration script.

GitHub info: PR 1130

1.4.3 Versions 3.0.x (3.0.0 – 3.0.7)

1.4.3.1 Version 3.0.7 (2021-06-29)

- Added parameters for the Delta variant.
- Refactored the NAb decay function to match the published version of Khoury et al. (the previous implementation matched the preprint).
- Added optional capacity limit for cv.contact_tracing to cap the maximum number of people that can be traced each day.
- When loading a population from file, this is now done during sim initialization (sim.initialize()); previously this was done as part of sim creation (cv.Sim()). This fixed a bug with immunity characteristics not being initialized correctly. (Thanks to Paula Sanz-Leon for identifying and proposing a fix.)
- Fixed a log of 0 warning with NAbs.
- Fixed n_beds_hosp = 0 and n_beds_icu = 0 being ignored (for no limit, use n_beds_hosp = None or n_beds_hosp = np.inf; thanks to Ankit Majhi for finding this bug).
- Added a more helpful error message if you try to export a MultiSim to JSON or Excel without reducing it first. (Thanks to Andrew Clark for finding this bug.)
- Regression information: Due to the change in NAb decay function, simulations run with use_waning = True will be slightly different than before. We are aiming to have a (relatively) stable version by Covasim v3.1; in the mean time, this aspect of the model may continue to receive frequent updates.

GitHub info: PR 1102

1.4.3.2 Version 3.0.6 (2021-06-21)

- Added alpha, beta, and gamma as aliases for variants B117, B1351, and P1, respectively.
- Split vaccine implementation to separate the state changes associated with vaccinating a person from the allocation/prioritization of vaccine distribution. The base class cv.BaseVaccination implements vaccinating individuals, and derived classes define the cv.BaseVaccination.select_people() method which determines who to vaccinate each timestep.
- Added cv.vaccinate_num() as an alternate way to allocate vaccines. This intervention specifies the order in which to vaccinate people, and the number of doses to distribute each day.
- Renamed cv.vaccinate() to cv.vaccinate_prob(), but added cv.vaccinate() as an alias that can be used (more or less) interchangeably with cv.vaccinate_prob().
- Updated NAb kinetics so that the NAb level no longer exceeds the peak NAb value after the second dose, and updated nab_growth_decay so that the NAb level no longer increases in the second decay phase (i.e. after 250 days by default). Note: we are in the process of changing the functional form for the NAb waning, so this will likely change again in version 3.0.7.
- Vaccine parameters for simulations with multiple different vaccines are now correctly handled. Previously only the first vaccine’s parameters were used.
• Added a new `fit_args` argument to the `Calibration` class, allowing arguments to be passed to `sim.compute_fit()`. Also added a `par_samplers` argument, allowing different Optuna samplers to be specified.

• *Regression information:* `cv.vaccination` has been renamed to `cv.vaccinate_prob` (however, `cv.vaccinate()` is retained as an alias to `cv.vaccinate_prob()`), so user code should not break). The correction to the NAb decay implementation means results in simulations with vaccines and a long duration (e.g., >250 days) may differ – vaccines are expected to be slightly less effective.

• *GitHub info:* PR 1088

1.4.3.3 Version 3.0.5 (2021-05-26)

• Changed all reference to variants from `strain` to `variant`. For example, `cv.strain()` is now `cv.variant()`, `cv.Sim(strains=...)` is now `cv.Sim(variants=...)`, etc. See this article for the rationale behind the change.

• Changed the `nab_to_efficacy` function based on a joint estimation of the marginal vaccine efficacies and inferred conditional efficacies.

• Changed the parameters provided to `nab_to_efficacy` function.

• Updated some strain parameters to be based on studies and not modeled inferences.

• *Regression information:* All instances of `strain` should be renamed `variant`. A find-and-replace should be sufficient for updating most scripts. Parameter values and functional forms have also been updated, so results using waning immunity will differ.

• *GitHub info:* PR 1069

1.4.3.4 Version 3.0.4 (2021-05-19)

• Fixed a bug that prevented simulations from being run without prognoses by age.

• Fixed an array length mismatch for single-dose vaccines.

• The default antibody kinetics are now a 3-part curve, with a 14-day growth, 250 day exp decay and then another exponential decay with an exponentially decaying decay parameter. This is captured in the new NAb functional form, `nab_growth_decay`. To align with this change, NAbs are now initialized at the time of infection, so that individuals build immunity over the course of infection.

• Some strain parameter changes based on https://www.eurosurveillance.org/content/10.2807/1560-7917.ES.2021.26.16.2100348

• Added strain to the infection log

• Removed the `rel_imm_strain` parameter; self-immunity is now always 1.0.

• Updated vaccine and strain parameter values based on fits to empirical data.

• Merged multisims now use the labels from each multisim, rather than the sim labels, for plotting.

• *Regression information:* Parameter values have been updated, so results using waning immunity will differ.

• *GitHub info:* PR 1058
1.4.3.5 Version 3.0.3 (2021-05-17)

- Added a new class, `cv.Calibration`, that can perform automatic calibration. Simplest usage is `sim.calibrate(calib_pars)`.
  Note: this requires Optuna, which is not installed by default; please install separately via `pip install optuna`. See the updated calibration tutorial for more information.
- Added a new result, `known_deaths`, which counts only deaths among people who have been diagnosed.
- Updated several vaccine and variant parameters (e.g., B1.351 and B117 cross-immunity).
- `sim.compute_fit()` now returns the fit by default, and creates `sim.fit` (previously, this was stored in `sim.results.fit`).
- Regression information: Calls to `sim.results.fit` should be replaced with `sim.fit`. The output parameter for `sim.compute_fit()` has been removed since it now always outputs the Fit object.
- GitHub info: PR 1047

1.4.3.6 Version 3.0.2 (2021-04-26)

- Added Novavax as one of the default vaccines.
- If `use_waning=True`, people will now become *undiagnosed* when they recover (so they are not incorrectly marked as diagnosed if they become reinfected).
- Added a new method, `sim.to_df()`, that exports results to a pandas dataframe.
- Added `people.lock()` and `people.unlock()` methods, so you do not need to set `people._lock` manually.
- Added extra parameter checking to `people.set_pars(pars)`, so `pop_size` is guaranteed to be an integer.
- Flattened `sim['immunity']` to no longer have separate axes for susceptible, symptomatic, and severe.
- Fixed a bug in `cv.sequence()`, introduced in version 2.1.2, that meant it would only ever trigger the last intervention.
- Fixed a bug where if subtargeting was used with `cv.vaccinate()`, it would trigger on every day.
- Fixed `msim.compare()` to be more careful about not converting all results to integers.
- Regression information: If you are using waning, `sim.people.diagnosed` no longer refers to everyone who has ever been diagnosed, only those still infectious. You can use `sim.people.defined('date_diagnosed')` in place of `sim.people.true('diagnosed')` (before these were identical).
- GitHub info: PR 1020

1.4.3.7 Version 3.0.1 (2021-04-16)

- Immunity and vaccine parameters have been updated.
- The `People` class has been updated to remove parameters that were copied into attributes; thus there is no longer both `people.pars['pop_size']` and `people.pop_size`; only the former. Recommended practice is to use `len(people)` to get the number of people.
- Loaded population files can now be used with more than one strain; arrays will be resized automatically. If there is a mismatch in the number of people, this will not be automatically resized.
- A bug was fixed with the `rescale` argument to `cv.strain()` not having any effect.
• Dead people are no longer eligible to be vaccinated.

• Regression information: Any user scripts that call sim.people.pop_size should be updated to call len(sim.people) (preferred), or sim.n, sim['pop_size'], or sim.people.pars['pop_size'].

• GitHub info: PR 999

1.4.3.8 Version 3.0.0 (2021-04-13)

This version introduces fully featured vaccines, variants, and immunity. Note: These new features are still under development; please use with caution and email us at covasim@idmod.org if you have any questions or issues. We expect there to be several more releases over the next few weeks as we refine these new features.

Highlights

• Model structure: The model now follows an “SEIS”-type structure, instead of the previous “SEIR” structure. This means that after recovering from an infection, agents return to the “susceptible” compartment. Each agent in the simulation has properties sus_imm, trans_imm and prog_imm, which respectively determine their immunity to acquiring an infection, transmitting an infection, or developing a more severe case of COVID-19. All these immunity levels are initially zero. They can be boosted by either natural infection or vaccination, and thereafter they can wane over time or remain permanently elevated.

• Multi-strain modeling: Model functionality has been extended to allow for modeling of multiple different co-circulating strains with different properties. This means you can now do e.g. b117 = cv.strain('b117', days=1, n_imports=20) followed by sim = cv.Sim(strains=b117) to import strain B117. Further examples are contained in tests/test_immunity.py and in Tutorial 8.

• New methods for vaccine modeling: A new cv.vaccinate() intervention has been added, which allows more flexible modeling of vaccinations. Vaccines, like natural infections, are assumed to boost agents’ immunity.

• Consistency: By default, results from Covasim 3.0.0 should exactly match Covasim 2.1.2. To use the new features, you will need to manually specify cv.Sim(use_waning=True).

• Still TLDR? Here’s a quick showcase of the new features:

```python
import covasim as cv

pars = dict(  
    use_waning = True, # Use the new immunity features  
    n_days = 180, # Set the days, as before  
    n_agents = 50e3, # New alias for pop_size  
    scaled_pop = 200e3, # New alternative to specifying pop_scale  
    strains = cv.strain('b117', days=20, n_imports=20), # Introduce B117  
    interventions = cv.vaccinate('astrazeneca', days=80), # Create a vaccine
)

cv.Sim(pars).run().plot('strain') # Create, run, and plot strain results
```

Immunity-related parameter changes

• A new control parameter, use_waning, has been added that controls whether to use new waning immunity dynamics (“SEIS” structure) or the old dynamics where post-infection immunity was perfect and did not wane (“SEIR” structure). By default, use_waning=False.
• A subset of existing parameters have been made strain-specific, meaning that they are allowed to differ by strain. These include: rel_beta, which specifies the relative transmissibility of a new strain compared to the wild strain; rel_symp_prob, rel_severe_prob, rel_crit_prob, and the newly-added immunity parameters rel_imm (see next point). The list of parameters that can vary by strain is specified in defaults.py.

• The parameter n_strains is an integer that specifies how many strains will be in circulation at some point during the course of the simulation.

• Seven new parameters have been added to characterize agents’ immunity levels:
  – The parameter nab_init specifies a distribution for the level of neutralizing antibodies that agents have following an infection. These values are on log2 scale, and by default they follow a normal distribution.
  – The parameter nab_decay is a dictionary specifying the kinetics of decay for neutralizing antibodies over time.
  – The parameter nab_kin is constructed during sim initialization, and contains pre-computed evaluations of the nab decay functions described above over time.
  – The parameter nab_boost is a multiplicative factor applied to a person’s nab levels if they get reinfected.
  – The parameter cross_immunity. By default, infection with one strain of SARS-CoV-2 is assumed to grant 50% immunity to infection with a different strain. This default assumption of 50% cross-immunity can be modified via this parameter (which will then apply to all strains in the simulation), or it can be modified on a per-strain basis using the immunity parameter described below.
  – The parameter immunity is a matrix of size total_strains by total_strains. Row i specifies the immunity levels that people who have been infected with strain i have to other strains.
  – The parameter rel_imm is a dictionary with keys asymp, mild and severe. These contain scalars specifying the relative immunity levels for someone who had an asymptomatic, mild, or severe infection. By default, values of 0.98, 0.99, and 1.0 are used.

• The parameter strains contains information about any circulating strains that have been specified as additional to the default strain. This is initialized as an empty list and then populated by the user.

Other parameter changes

• The parameter frac_susceptible will initialize the simulation with less than 100% of the population to be susceptible to COVID (to represent, for example, a baseline level of population immunity). Note that this is intended for quick explorations only, since people are selected at random, whereas in reality higher-risk people will typically be infected first and preferentially be immune. This is primarily designed for use with use_waning=False.

• The parameter scaled_pop, if supplied, can be used in place of pop_scale or pop_size. For example, if you specify cv.Sim(pop_size=100e3, scaled_pop=550e3), it will automatically calculate pop_scale=5.5.

• Aliases have been added for several parameters: pop_size can also be supplied as n_agents, and pop_infected can also be supplied as init_infected. This only applies when creating a sim; otherwise, the default names will be used for these parameters.
Changes to states and results

- Several new states have been added, such as `people.naive`, which stores whether or not a person has ever been exposed to COVID before.
- New results have been added to store information by strain, as well as population immunity levels. In addition to new entries in `sim.results`, such as `pop_nabs` (population level neutralizing antibodies) and `new_reinfections`, there is a new set of results `sim.results.strain: cum_infections_by_strain, cum_infectious_by_strain, new_infections_by_strain, new_infected_by_strain, prevalence_by_strain, incidence_by_strain`.

New functions, methods and classes

- The newly-added file `immunity.py` contains functions, methods, and classes related to calculating immunity. This includes the `strain` class (which uses lowercase convention like Covasim interventions, which are also technically classes).
- A new `cv.vaccinate()` intervention has been added. Compared to the previous `vaccine` intervention (now renamed `cv.simple_vaccine()`), this new intervention allows vaccination to boost agents’ immunity against infection, transmission, and progression.
- There is a new `sim.people.make_nonnaive()` method, as the opposite of `sim.people.make_naive()`.
- New functions `cv.iundefined()` and `cv.iundefinedi()` have been added for completeness.
- A new function `cv.demo()` has been added as a shortcut to `cv.Sim().run().plot()`.
- There are now additional shortcut plotting methods, including `sim.plot('strain')` and `sim.plot('all')`.

Renamed functions and methods

- `cv.vaccine()` is now called `cv.simple_vaccine()`.
- `cv.get_sim_plots()` is now called `cv.get_default_plots(); cv.get_scen_plots()` is now `cv.get_default_plots(kind='scen')`.
- `sim.people.make_susceptible()` is now called `sim.people.make_naive()`.

Bugfixes

- `n_imports` now scales correctly with population scale (previously they were unscaled).
- `cv.ifalse()` and related functions now work correctly with non-boolean arrays (previously they used the `~` operator instead of `np.logical_not()`, which gave incorrect results for int or float arrays).
- Interventions and analyzers are now deep-copied when supplied to a sim; this means that the same ones can be created and then used in multiple sims. Scenarios also now deep-copy their inputs.

Regression information

- As noted above, with `cv.Sim(use_waning=False)` (the default), results should be the same as Covasim 2.1.2, except for new results keys mentioned above (which will mostly be zeros, since they are only populated with immunity turned on).
• Scripts using `cv.vaccine()` should be updated to use `cv.simple_vaccine()`.
• Scripts calling `sim.people.make_susceptible()` should now call `sim.people.make_naive()`.
• *GitHub info:* PR 927

1.4.4 Versions 2.x (2.0.0 – 2.1.2)

1.4.4.1 Version 2.1.2 (2021-03-31)

• Interventions and analyzers now accept a function as an argument to `days` or e.g. `start_day`. For example, instead of defining `start_day=30`, you can define a function (with the intervention and the sim object as arguments) that calculates and returns a start day. This allows interventions to be dynamically triggered based on the state of the sim. See [Tutorial 5](https://docs.idmod.org/projects/covasim/en/latest/tutorials/t05.html) for a new section on how to use this feature.
• Added a `finalize()` method to interventions and analyzers, to replace the `if sim.t == sim.npts-1:` blocks in `apply()` that had been being used to finalize.
• Changed setup instructions from `python setup.py develop` to `pip install -e .`, and unpinned `lineProfiler`.
• *Regression information:* If you have any scripts/workflows that have been using `python setup.py develop`, please update them to `pip install -e .`. Likewise, `python setup.py develop` is now `pip install -e .` [full].
• *GitHub info:* PR 897

1.4.4.2 Version 2.1.1 (2021-03-29)

• **Duration updates:** All duration parameters have been updated from the literature. While most are similar to what they were before, there are some differences: in particular, durations of severe and critical disease (either to recovery or death) have increased; for example, duration from symptom onset to death has increased from $15.8\pm3.8$ days to $18.8\pm7.2$ days.
• **Performance updates:** The innermost loop of Covasim, `cv.compute_infections()`, has been refactored to make more efficient use of array indexing. The observed difference will depend on the nature of the simulation (e.g., network type, interventions), but runs may be up to 1.5x faster now.
• **Graphs:** People, contacts, and contacts layers now have a new method, `to_graph()`, that will return a networkx graph (requires networkx to be installed, of course). For example, `nx.draw(cv.Sim(pop_size=100).run().people.to_graph())` will draw all connections between 100 default people. See `cv.Sim.people.to_graph()` for full documentation.
• A bug was fixed with `cv.TransTree.animate()` failing in some cases.
• `cv.date_formatter()` now takes `interval`, `start`, and `end` arguments.
• Temporarily pinned `lineProfiler` to version 3.1 due to this issue.
• *Regression information:* Parameters can be restored by using the `version` argument when creating a sim. Specifically, the parameters for the following distributions (all lognormal) have been changed as follows:

<table>
<thead>
<tr>
<th>Distribution</th>
<th>New Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>exp2inf</code></td>
<td>$\mu = 4.6 \rightarrow 4.5, \sigma = 4.8 \rightarrow 1.5$</td>
</tr>
<tr>
<td><code>inf2sym</code></td>
<td>$\mu = 1.0 \rightarrow 1.1, \sigma = 0.9 \rightarrow 0.9$</td>
</tr>
<tr>
<td><code>sev2crit</code></td>
<td>$\mu = 3.0 \rightarrow 1.5, \sigma = 7.4 \rightarrow 2.0$</td>
</tr>
<tr>
<td><code>sev2rec</code></td>
<td>$\mu = 14.0 \rightarrow 18.1, \sigma = 2.4 \rightarrow 6.3$</td>
</tr>
</tbody>
</table>

(continues on next page)
crit2rec: $\mu = 14.0 \rightarrow 18.1$, $\sigma = 2.4 \rightarrow 6.3$
crit2die: $\mu = 6.2 \rightarrow 10.7$, $\sigma = 1.7 \rightarrow 4.8$

- GitHub info: PR 887

1.4.4.3 Version 2.1.0 (2021-03-23)

Highlights

- Updated lognormal distributions: Lognormal distributions had been inadvertently using the variance instead of the standard deviation as the second parameter, resulting in too small variance. This has been fixed. This has a small but nonzero impact on the results (e.g. with default parameters, the time to peak infections is about 5-10% sooner now).

- Expanded plotting features: You now have much more flexibility with passing arguments to `sim.plot()` and other plotting functions, such as to temporarily set global Matplotlib options (such as DPI), modify axis styles and limits, etc. For example, you can now do things like this: `cv.Sim().run().plot(dpi=150, rotation=30, start_day='2020-03-01', end_day=55, interval=7)`.

- Improved analyzers: Transmission trees can be computed 20 times faster, Fit objects are more forgiving for data problems, and analyzers can now be exported to JSON.

Bugfixes

- Previously, the lognormal distributions were unintentionally using the variance of the distribution, instead of the standard deviation, as the second parameter. This makes a small difference to the results (slightly higher transmission due to the increased variance). Old simulations that are loaded will automatically have their parameters updated so they give the same results; however, new simulations will now give slightly different results than they did previously. (Thanks to Ace Thompson for identifying this.)

- If a results object has low and high values, these are now exported to JSON (and also to Excel).

- MultiSim and Scenarios `run()` methods now return themselves, as Sim does. This means that just as you can do `sim.run().plot()`, you can also now do `msim.run().plot()`.

Plotting and options

- Standard plots now accept keyword arguments that will be passed around to all available subfunctions. For example, if you specify `dpi=150`, Covasim knows that this is a Matplotlib setting and will configure it accordingly; likewise things like `bottom` (only for axes), `frameon` (only for legends), etc. If you pass an ambiguous keyword (e.g. `alpha`, which is used for line and scatter plots), it will only be used for the first one.

- There is a new keyword argument, `date_args`, that will format the x-axis: options include `dateformat` (e.g. `%Y-%m-%d`), `rotation` (to avoid label collisions), and `start_day` and `end_day`.

- Default plotting styles have updated, including less intrusive lines for interventions.

Other changes

- MultiSims now have `to_json()` and `to_excel()` methods, which are shortcuts for calling these methods on the base sim.
• If no label is supplied to an analyzer or intervention, it will use its class name (e.g., the default label for `cv.change_beta` is 'change_beta').

• Analyzers now have a `to_json()` method.

• The `cv.Fit` and `cv.TransTree` classes now derive from `Analyzer`, giving them some new methods and attributes.

• `cv.sim.compute_fit()` has a new keyword argument, `die`, that will print warnings rather than raise exceptions if no matching data is found. Exceptions are now caught and helpful error messages are provided (e.g., if dates don’t match).

• The algorithm for `cv.TransTree` has been rewritten, and now runs 20x as fast. The detailed transmission tree, in `tt.detailed`, is now a pandas dataframe rather than a list of dictionaries. To restore something close to the previous version, use `tt.detailed.to_dict('records')`.

• A data file with an integer rather than date “date” index can now be loaded; these will be counted relative to the simulation’s start day.

• `cv.load()` has two new keyword arguments, `update` and `verbose`, than are passed to `cv.migrate()`.

• `cv.options` has new a `get_default()` method which returns the value of that parameter when Covasim was first loaded.

Documentation and testing

• An extra tutorial has been added on “Deployment”, covering how to use it with Dask and for using Covasim with interactive notebooks and websites.

• Tutorials 7 and 10 have been updated so they work on Windows machines.

• Additional unit tests have been written to check the statistical properties of the sampling algorithms.

Regression information

• To restore previous behavior for a simulation (i.e., using variance instead of standard deviation for lognormal distributions), call `cv.misc.migrate_lognormal(sim)`. This is done automatically when loading a saved sim from disk. To undo a migration, type `cv.misc.migrate_lognormal(sim, revert=True)`. What this function does is loop over the duration parameters and replace `par2` with its square root. If you have used lognormal distributions elsewhere, you will need to update them manually.

• Code that was designed to parse transmission trees will likely need to be revised. The object `tt.detailed` is now a dataframe; calling `tt.detailed.to_dict('records')` will bring it very close to what it used to be, with the exception that for a given row, 't' and 's' used to be nested dictionaries, whereas now they are prefixes. For example, whereas before the 45th person’s source’s “is quarantined” state would have been `tt.detailed[45]['s']['is_quarantined']`, it is now `tt.detailed.iloc[45]['src_is_quarantined']`.

• GitHub info: PR 859

1.4.4.4 Version 2.0.4 (2021-03-19)

• Added a new analyzer, `cv.daily_age_stats()`, which will compute statistics by age for each day of the simulation (compared to `cv.age_histogram()`, which only looks at particular points in time).

• Added a new function, `cv.date_formatter()`, which may be useful in quickly formatting axes using dates.
• Removed the need for `self._store_args()` in interventions; now custom interventions only need to implement `super().__init__(**kwargs)` rather than both.
• Changed how custom interventions print out by default (a short representation rather than the jsonified version used by built-in interventions).
• Added an `update()` method to `Layer`, to allow greater flexibility for dynamic updating.
• *GitHub info:* PR 854

### 1.4.4.5 Version 2.0.3 (2021-03-11)

• Previously, the way a sim was printed (e.g. `print(sim)`) depended on what the global `verbose` parameter was set to (e.g. `cv.options.set(verbos=0.1)`), which used `sim.brief()` if verbosity was 0, or `sim.disp()` otherwise. This has been changed to always use the `sim.brief()` representation regardless of verbosity. To restore the previous behavior, use `sim.disp()` instead of `print(sim)`.
• `sim.run()` now returns a pointer to the sim object rather than either nothing (the current default) or the `sim.results` object. This means you can now do e.g. `sim.run().plot()` or `sim.run().results` rather than `sim.run(do_plot=True)` or `sim.run(output=True)`.
• `sim.get_interventions()` and `sim.get_analyzers()` have been changed to return all interventions/analyzers if no arguments are supplied. Previously, they would return only the last intervention. To restore the previous behavior, call `sim.get_intervention()` or `sim.get_analyzer()` instead.
• The `Fit` object (and `cv.compute_gof()`) have been updated to allow a custom goodness-of-fit estimator to be supplied.
• Two new results have been added, `n_preinfectious` and `n_removed`, corresponding to the E and R compartments of the SEIR model, respectively.
• A new shortcut plotting option has been introduced, `sim.plot(to_plot='seir')`.
• Plotting colors have been revised to have greater contrast.
• The `numba_parallel` option has been updated to include a “safe” option, which parallelizes as much as it can without disrupting the random number stream. For large sims (>100,000 people), this increases performance by about 10%. The previous `numba_parallel=True` option now corresponds to `numba_parallel='full'`, which is about 20% faster but means results are non-reproducible. Note that for sims smaller than 100,000 people, Numba parallelization has almost no effect on performance.
• A new option has been added, `numba_cache`, which controls whether or not Numba functions are cached. They are by default to save compilation time, but if you change Numba options (especially `numba_parallel`), with caching you may also need to delete the `__pycache__` folder for changes to take effect.
• A frozen list of `pip` requirements, as well as test requirements, has been added to the `tests` folder.
• The testing suite has been revamped, with defensive code skipped, bringing code coverage to 90%.
• *Regression information:* Calls to `sim.run(do_plot=True, **kwargs)` should be changed to `sim.run().plot(**kwargs)`. Calls to `sim.get_interventions()/sim.get_analyzers()` (with no arguments) should be changed to `sim.get_intervention()/sim.get_analyzer()`. Calls to `results = sim.run(output=True)` should be replaced with `results = sim.run().results`.
• *GitHub info:* PR 788
1.4.4.6 Version 2.0.2 (2021-02-01)

- Added a new option to easily turn on/off interactive plotting: e.g., simply set `cv.options.set(interactive=False)` to turn off interactive plotting. This meta-option sets the other options `show`, `close`, and `backend`.
- Changed the logic of `do_show`, such that `do_show=False` will never show a plot, even if `cv.options.show` is True.
- Added a new method, `cv.diff_sims()`, that allows the differences in results between two simulations to be quickly calculated.
- Removed the `keys` argument from `cv.daily_stats()`, since non-default keys are had to validate.
- Fixed a bug that prevented prognoses parameters from being correctly set to those from an earlier version.
- Added an R usage example to the `examples` folder (matching the one in the FAQ).
- Added additional tests, increasing test coverage from 72% to 88%.
- *GitHub info:* PR 779

1.4.4.7 Version 2.0.1 (2021-01-31)

- Pinned `xlrd` version to 1.2.0 due to a bug introduced in the 2.0.1 version of `xlrd` (see here for details).
- Fixed a bug that prevented a function from being supplied as `subtarget` for `cv.test_prob()`.
- Fixed a bug that prevented regression parameters (e.g. `cv.Sim(version='1.7.5')`) from working when Covasim was installed via pip.
- Fixed typos in docstrings and tutorials.
- *GitHub info:* PR 775

1.4.4.8 Version 2.0.0 (2020-12-05)

This version contains a number of major updates. Note: this version requires Sciris 1.0, so when upgrading to this version, you may also need to upgrade Sciris (`pip install sciris --upgrade`).

**Highlights**

- **Parameters:** Default infection fatality ratio estimates have been updated in line with the latest literature.
- **Plotting:** Plotting defaults have been updated to support a wider range of systems, and users now have greater control over plotting and options.
- **New functions:** New methods have been added to display objects in different levels of detail; new methods have also been added for working with data, adding contacts, and analyzing multisims.
- **Webapp:** The webapp has been moved to a separate Python package, `covasim_webapp` (available here).
- **Documentation:** A comprehensive set of tutorials has been added, along with a glossary and FAQ; see https://docs.covasim.org or look in the `docs/tutorials` folder.
Parameter updates

- The infection fatality rate has been updated to use O’Driscoll et al. (https://www.nature.com/articles/s41586-020-2918-0). We also validated against other estimates, most notably Brazeau et al. (https://www.imperial.ac.uk/mrc-global-infectious-disease-analysis/covid-19/report-34-ifr). The new estimates have similar estimates for older ages, but tend to be lower for younger ages, especially the 60–70 age category.

- While we have not made any updates to the hospitalization rate, viral load distribution, or infectious durations at this time, we are currently reviewing the literature on these parameters and may be making updates relatively soon.

- A new version option has been added to sims, to use an earlier version of parameters if desired. For example, you can use Covasim version 2.0 but with default parameters from the previous version (1.7.6) via `sim = cv.Sim(version='1.7.6')`. If you wish to load and inspect parameters without making a sim, you can use e.g. `cv.get_version_pars('1.7.6')`.

- A `cv.migration()` function has also been added. Covasim sims and multisims are “migrated” (updated to have the right structure) automatically if loading old versions. However, you may wish to call this function explicitly if you’re migrating a custom saved object (e.g., a list of sims).

Plotting and options

- Plotting defaults have been updated to work better on a wider variety of systems.

- Almost all plotting functions now take both `fig` and `ax` keywords, which let you pass in existing figures/axes to be used by the plot.

- A new options module has been added that lets the user specify plotting and run options; see `cv.options.help()` for a list of the options.

- Plot options that were previously set on a per-figure basis (e.g. font size, font family) are now set globally via the options module, e.g. `cv.options.set(font_size=18)`.

- If plots are too small, you can increase the DPI (default 100), e.g. `cv.options.set(dpi=200)`. If they are too large, you can decrease it, e.g. `cv.options.set(dpi=50)`.

- In addition, you can also change whether Covasim uses 32-bit or 64-bit arithmetic. To use 64-bit (which is about 20% slower and uses about 40% more memory), use `cv.options.set(precision=64)`.

- Options can also now be set via environment variables. For example, you can set `COVASIM_DPI` to change the default DPI, and `COVASIM_VERBOSE` to set the default verbosity. For example, `export COVASIM_VERBOSE=0` is equivalent to `cv.options.set(VERBOSE=0)`. See `cv.options.help()` for the full list.

- The built-in intervention plotting method was renamed from `plot()` to `plot_intervention()`, allowing the user to define custom plotting functions that do something different.

Webapp

- The webapp has been moved to a separate repository and `pip` package, in order to improve installation and load times of Covasim.

- The `docker` and `.platform` folders have been moved to `covasim_webapp`.

- Since web dependencies are no longer included, installing and importing Covasim both take half as much time as they did previously.
Bugfixes

- The `quar_period` argument is now correctly passed to the `cv.contact_tracing()` intervention. (Thanks to Scott McCrae for finding this bug.)
- If the user supplies an incorrect type to `cv.Layer.find_contacts()`, this is now caught and corrected. (Thanks to user sba5827 for finding this bug.)
- Non-string Layer keys no longer raise an exception.
- The `sim.compute_r_eff()` error message now gives correct instructions (contributed by Andrea Cattaneo).
- Parallelization in Jupyter notebooks (e.g. `msim.run()`) should now work without crashing.
- If parallelization (e.g. `msim.run()`) is called outside a main block on Windows, this leads to a cryptic error. This error is now caught more elegantly.
- Interventions now print out with their actual name (previously they all printed out as InterventionDict).
- The keyword argument `test_sensitivity` for `cv.test_prob()` has been renamed `sensitivity`, for consistency with `cv.test_num()`.

New functions and methods

- Sims, multisims, scenarios, and people objects now have `disp()`, `summarize()`, and `brief()` methods, which display full detail, moderate detail, and very little detail about each. If `cv.options.verbose` is 0, then `brief()` will be used to display objects; otherwise, `disp()` will be used.
- Two new functions have been added, `sim.get_intervention()` and `sim.get_analyzer()`. These act very similarly to e.g. `sim.get_interventions()`, except they return the last matching intervention/analyzer, rather than returning a list of interventions/analyzers.
- MultiSims now have a `shrink()` method, which shrinks both the base sim and the other sims they contain.
- MultiSims also provide options to compute statistics using either the mean or the median; this can be done via the `msim.reduce(use_mean=True)` method. Two convenience methods, `msim.mean()` and `msim.median()`, have also been added as shortcuts.
- Scenarios now have a `scens.compare()` method, which (like the multisim equivalent) creates a dataframe comparing results across scenarios.
- Contacts now have new methods for handling layers, `sim.people.contacts.add_layer()` and `sim.people.contacts.pop_layer()`. Additional validation on layers is also performed.
- There is a new function, `cv.data.show_locations()`, that lists locations for which demographic data are available. You can also now edit the data dictionaries directly, by modifying e.g. `cv.data.country_age_data.data` (suggested by Andrea Cattaneo).

Other changes

- There is a new verbose option for sims: `cv.Sim(verbos='brief').run()` will print a single line of output when the sim finishes (namely, `sim.brief()`).
- The argument `n_cpus` can now be supplied directly to `cv.multirun()` and `msim.run()`.
- The types `cv.default_float` and `cv.default_int` are now available at the top level (previously they had to be accessed by e.g. `cv.defaults.default_float`).
• Transmission trees now contain additional output; after `tt = sim.make_transtree()`, a dataframe of key results is contained in `tt.df`.

• The default number of seed infections has been changed from 10 to 20 for greater numerical stability. (Note that this placeholder value should be overridden for all actual applications.)

• `sim.run()` no longer returns the results object by default (if you want it, set `output=True`).

• A migrations module has been added (in `misc.py`). Objects are now automatically migrated to the current version of Covasim when loaded. The function `cv.migrate()` can also be called explicitly on objects if needed.

Documentation

• A glossary, FAQ, and tutorials have been added. All are available from https://docs.covasim.org.

Regression information

• To restore previous default parameters for simulations, use e.g. `sim = cv.Sim(version='1.7.6')`. Note that this does not affect saved sims (which store their own parameters).

• Any scripts that specify the `test_sensitivity` keyword for the `test_prob` intervention will need to rename that variable to `sensitivity`.

• Any scripts that used `results = sim.run()` will need to be updated to `results = sim.run(output=True)`.

• Any scripts that passed formatting options directly to plots should set these as options instead; e.g. `sim.plot(font_size=18)` should now be `cv.options.set(font_size=18); sim.plot()`.

• Any custom interventions that defined a custom `plot()` method should use `plot_interventions()` instead.

• **GitHub info:** PRs 738, 740

1.4.5 Versions 1.7.x (1.7.0 – 1.7.6)

1.4.5.1 Version 1.7.6 (2020-10-23)

• Added additional flexibility to `cv.People`, `cv.make_people()`, and `cv.make_synthpop()` to allow easier modification of different types of people (e.g. the raw output of SynthPops, the popdict, and the `People` object).

• **GitHub info:** PR 712

1.4.5.2 Version 1.7.5 (2020-10-13)

• Added extra convenience methods to `Layer` objects:

  - `Layer.members` returns an array of all people with interactions in the layer
  - `__contains__` is implemented so `uid in layer` can be used

• `cv.sequence.apply()` passes on the underlying intervention’s return value rather than always returning None

• **GitHub info:** PR 709
1.4.5.3 Version 1.7.4 (2020-10-02)

- Refactored `cv.contact_tracing()` so that derived classes can extend individual parts of contact tracing without having to re-implement the entire intervention
- Moved `people.trace` to `contact_tracing` so that the tracing step can be extended via custom interventions
- **Regression information:** Custom interventions calling `people.trace` should inherit from `cv.contact_tracing` instead and use `contact_tracing.identify_contacts` and `contact_tracing.notify_contacts` to replace `people.trace`. In most cases however, it would be possible to overload one of the contact tracing steps rather than `contact_tracing.apply`, which thus eliminates the need to call `people.trace` entirely.
- **GitHub info:** PR 702

1.4.5.4 Version 1.7.3 (2020-09-30)

- Changed `test_prob.apply()` and `test_num.apply()` to return the indices of people that were tested
- `cvm.date(None)` returns `None` instead of an empty list. Both `cvm.date()` and `cvm.day()` no longer raise errors if the list of inputs includes `None` entries.
- **GitHub info:** PR 699

1.4.5.5 Version 1.7.2 (2020-09-24)

- Changed the intervention validation introduced in version 1.7.1 from an exception to a printed warning, to accommodate for custom-defined interventions.
- Docstrings were clarified to indicate that usage guidance is a recommendation, not a requirement.
- **GitHub info:** PR 693

1.4.5.6 Version 1.7.1 (2020-09-23)

- Added two new methods, `sim.get_interventions()` and `sim.get_analyzers()`, which return interventions or analyzers based on the index, label, or type.
- Added a new analyzer, `cv.daily_stats()`, which can print out and plot detailed information about the state of the simulation on each day.
- MultiSims can now be run without parallelization; use `msim.run(parallel=False)`. This can be useful for debugging, or for parallelizing across rather than within MultiSims (since multiprocessing calls cannot be nested).
- `sim.people.not_defined()` has been renamed `sim.people.undefined()`, and `sim.people.quarantine()` has been renamed `sim.people.schedule_quarantine()`, since it does not actually place people in quarantine.
- New helper functions have been added: `cv.maximize()` maximizes the current figure, and `cv.get_rows_cols()` converts a number (usually a number of plots) into the required number of rows and columns. Both will eventually be moved to Sciris.
- The transmission tree plot has been corrected to account for people who have left quarantine. The definition of “quarantine end” for the sake of testing (`quar_policy='end'` for `cv.test_num()` and `cv.test_prob()`) has also been shifted up by a day (since by `date_end_quarantine`, people are no longer in quarantine by the end of the day, so tests were not being counted as happening in quarantine).
• Additional validation is done on intervention order to ensure that testing interventions are defined before tracing interventions.

• Code has been moved between `sim.py`, `people.py`, and `base.py` to better reflect the division between “the simulation” (the first two files) and “the housekeeping” (the last file).

• Regression information: Scripts that used `quar_policy='end'` may now provide stochastically different results. User scripts that explicitly call `sim.people.not_defined()` or `sim.people.quarantine()` should be updated to call `sim.people.undefined()` and `sim.people.schedule_quarantine()` instead.

• GitHub info: PR 690

1.4.5.7 Version 1.7.0 (2020-09-20)

• The way in which `test_num` handles rescaling has changed, taking into account the non-modeled population. It now behaves more consistently throughout the dynamic rescaling period. In addition, it previously used sampling with replacement, whereas now it uses sampling without replacement. While this does not affect results in most cases, it can make a difference if certain subgroups (e.g. people with severe disease) have very high testing rates.

• Two new results have been added: `n_alive` (total number of people minus deaths) and `rel_test_yield` (the proportion of tests that are positive relative to a random sample from the population). In addition, the `n_susceptible` calculation has been updated for simulations with dynamic rescaling to reflect the number of people rather than the number of agents.

• There are additional options for the quarantine policy in the `test_prob` intervention. For example, you can now test people on entry and 5 days into quarantine by specifying `quar_policy=[0,5]`.

• A new method `cv.randround()` has been introduced which will probabilistically round a float to an integer – for example, 3.2 will be rounded up 20% of the time and rounded down 80% of the time. This is used to ensure accurate mean values for small numbers.

• `cv.check_version()` can now take a comparison, e.g. `cv.check_version('>='1.7.0')`.

• A `People` object can now be created with a single number, representing the number of people. However, to be fully initialized, it still needs the other model parameters. This change lets the people and their connections be created first, and then inserted into a `sim` later.

• Additional checking is performed on interventions to ensure they are in the correct order (i.e., testing before tracing).

• The `Result` object used to have several scaling options, but now it simply has `True` (corresponding to the previous 'dynamic') and `False`. The `static` scaling option has been removed since it is no longer used by any result types.

• Regression information: sims that used `test_num` may now produce different results, given the changes for sample-without-replacement and dynamic rescaling. Previous behavior had the effect of artificially inflating the effectiveness of `test_num` before and during dynamic rescaling, since all tests were assigned to the modeled subpopulation. As a result, to get comparable results as before, test efficacy (loosely parameterized by `symp_test`) should increase. Although there is not an exact relationship, to give an example, a simulation with `symp_test=7` and `pop_scale=10` previously may correspond to `symp_test=25` now. This change means that `symp_test` behaves consistently across the simulation period, so whereas previously this parameter may have needed to change over time, it should now be possible to use a single value (typically the last one used).

• GitHub info: PR 684, head bfb9f66

1.4. What’s new
1.4.6 Versions 1.6.x (1.6.0 – 1.6.1)

1.4.6.1 Version 1.6.1 (2020-09-13)

- Unpinned `numba` from version 0.48. Version 0.49 changed the seed used for `np.random.choice()`, meaning that results from versions >=0.49 will differ from versions <=0.48. Version 0.49 was also significantly slower for some operations, which is why the switch was not made at the time, but this no longer appears to impact Covasim.
- `People.person()` now populates the contacts dictionary when returning a person, so that e.g. `sim.people[0].contacts` is no longer None.
- There is a new `story()` method for `People` that prints a history of an individual person, e.g. `sim.people.story(35)`.
- The baseline test in `test_baseline.py` has been updated to include contact tracing, giving greater code coverage for regression changes.
- Regression information: No changes to the Covasim codebase were made; however, new installations of Covasim (or if you update Numba manually) will have a different random number stream. To return previous results, use the previous version of Numba: `pip install numba==0.48.0`.
- GitHub info: PRs 669, 677, head 756e8eab

1.4.6.2 Version 1.6.0 (2020-09-08)

- There is a new `cv.vaccine()` intervention, which can be used to implement vaccination for subgroups of people. Vaccination can affect susceptibility, symptomaticity, or both. Multiple doses (optionally with diminishing efficacy) can be delivered.
- `cv.Layer` objects have a new highly optimized `find_contacts()` method, which reduces time required for the contact tracing by a factor of roughly 2. This method can also be used directly to find the matching contacts for a set of indices, e.g. `sim.people.contacts['h'].find_contacts([12, 144, 2048])` will find all contacts of the three people listed.
- The method `sim.compute_summary()` has been removed; `sim.summarize()` now serves both purposes. This function previously always took the last time point in the results arrays, but now can take any time point.
- A new `reset` keyword has been added to `sim.initialize()`, which will overwrite `sim.people` even if it already exists. Similarly, both interventions and analyzers are preserved after a sim run, unless `sim.initialize()` is called again (previously, analyzers were preserved but interventions were reset). This is to support storing data in interventions, as used by `cv.vaccine()`.
- `sim.date()` can now handle strings or date objects (previously, it could only handle integers).
- Data files in formats `.json` and `.xls` can now be loaded, in addition to the `.csv` and `.xlsx` formats supported previously.
- Additional flexibility has been added to plotting, including user-specified colors for data; custom sim labels; and reusing existing axes for plots.
- Metadata now saves correctly to PDF and SVG images via `cv.savefig()`. An issue with `cv.check_save_version()` using the wrong calling frame was also fixed.
- The field `dateExposed` has been added to transmission trees.
- The result “Effective reproductive number” has been renamed “Effective reproduction number”.
- Analyzers now have additional validation to avoid out-of-bounds dates, as well as additional test coverage.
• **Regression information:** No major backwards incompatibilities are introduced by this version. Instances of `sim.compute_summary()` should be replaced by `sim.summarize()`, and results dependent on the original state of an intervention post-simulation should use `sim._orig_pars['interventions']` (or perform `sim.initialize()` prior to using them) instead of `sim['interventions']`.

• **GitHub info:** PR 664, head e902cdff

### 1.4.7 Versions 1.5.x (1.5.0 – 1.5.3)

#### 1.4.7.1 Version 1.5.3 (2020-09-01)

• An `AlreadyRunError` is now raised if `sim.run()` is called in such a way that no timesteps will be taken. This error is a distinct type so that it can be safely caught and ignored if required, but it is anticipated that most of the time, calling `run()` and not taking any timesteps, would be an inadvertent error.

• If the simulation has reached the end, `sim.run()` (and `sim.step()`) will now raise an `AlreadyRunError`.

• `sim.run()` now only validates parameters as part of initialization. Parameters will always be validated in the normal workflow where `sim.initialize()` is called via `sim.run()`. However, the use case for modifying parameters during a split run or otherwise modifying parameters after initialization suggests that the user should have maximum control over the parameters at this point, so in this specialist workflow, the user is responsible for setting the parameter values correctly and in return, `sim.run()` is guaranteed not to change them.

• Added a `sim.complete` attribute, which is `True` if all timesteps have been executed. This is independent of finalizing results, since if `sim.step()` is being called externally, then finalizing the results may happen separately.

• **GitHub info:** PR 654, head d84b5f97

#### 1.4.7.2 Version 1.5.2 (2020-08-18)

• Modify `cv.People.quarantine()` to allow it schedule future quarantines, and allow quarantines of varying duration.

• Update the quarantine pipeline so that `date_known_contact` is not removed when someone goes into quarantine.

• Fixed bug where people identified as known contacts while on quarantine would be re-quarantined at the end of their quarantine for the entire quarantine duration. Now if a quarantine is requested while someone is already on quarantine, their existing quarantine will be correctly extended where required. For example, if someone is quarantined for 14 days on day 0 so they are scheduled to leave quarantine on day 14, and they are then subsequently identified as a known contact of a separate person on day 6 requiring 14 days quarantine, in previous versions of Covasim they would be released from quarantine on day 15, and then immediately quarantined on day 16 until day 30. With this update, their original quarantine would now be extended, so they would be released from quarantine on day 20.

• Quarantine duration via `cv.People.trace()` is now based on time since tracing, not time since notification, as people are typically instructed to isolate for a period after their last contact with the confirmed case, whenever that was. This results in an overall decrease in time spent in quarantine when the `trace_time` is greater than 0.

• **Regression information:**
Scripts that called `cv.People.quarantine()` directly would have also had to manually update `sim.results['new_quarantined']`. This is no longer required, and those commands should now be removed as they will otherwise be double counted.

Results are expected to differ slightly because the handling of quarantines being extended has been improved, and because quarantine duration is now reduced by the `trace_time`.

- *GitHub info:* PR 624, head 9041157f

1.4.7.3 Version 1.5.1 (2020-08-17)

- Modify `cv.BasePeople.__getitem__()` to retrieve a person if the item is an integer, so that `sim.people[5]` will return a `cv.Person` instance
- Modify `cv.BasePeople.__iter__` so that iterating over `people` e.g. `for person in sim.people:` iterates over `cv.Person` instances
- *Regression information:* To restore previous behavior of `for idx in sim.people:` use `for idx in range(len(sim.people))`; instead
- *GitHub info:* PR 623, head aaa4d7c1

1.4.7.4 Version 1.5.0 (2020-07-01)

- Based on calibrations to Seattle-King County data, default parameter values have been updated to have higher dispersion and smaller differences between layers.
- Keywords for computing goodness-of-fit (e.g. `use_frac`) can now be passed to the `Fit()` object.
- The overview plot (`to_plot='overview'`) has been updated with more plots.
- Subtargeting of testing interventions is now more flexible: values can now be specified per person.
- Issues with specifying DPI and for saving calling function information via `cv.savefig()` have been addressed.
- Several minor plotting bugs were fixed.
- A new function, `cv.undefined()`, can be used to find indices for which a quantity is *not* defined (e.g., `cv.undefined(sim.people.date_diagnosed)` returns the indices of everyone who has never been diagnosed).
- *Regression information:* To restore previous behavior, use the following parameter changes:

```python
pars['beta_dist'] = {'dist':'lognormal', 'par1':0.84, 'par2':0.3}
pars['beta_layer'] = dict(h=7.0, s=0.7, w=0.7, c=0.14)
pars['iso_factor'] = dict(h=0.3, s=0.0, w=0.0, c=0.1)
pars['quar_factor'] = dict(h=0.8, s=0.0, w=0.0, c=0.3)
```

- *GitHub info:* PR 596, head 775cf358

1.4.8 Versions 1.4.x (1.4.0 – 1.4.8)

1.4.8.1 Version 1.4.8 (2020-06-11)

- Prerelease version of 1.5.0, including the layer and beta distribution changes.
- *GitHub info:* head 2cb21846
1.4.8.2 Version 1.4.7 (2020-06-02)

- Added `quar_policy` argument to `cv.test_num()` and `cv.test_prob()`: by default, people are only tested upon entering quarantine ('start'); other options are to test people as they leave quarantine, both as they enter and leave, and every day they are in quarantine (which was the previous default behavior).
- Requirements have been tidied up; `python setup.py develop nowebapp` now only installs minimal packages. In a future version, this may become the default.
- Fixed intervention export and import from JSON.
- **Regression information**: To restore previous behavior (not recommended) with using contact tracing, add `quar_policy='daily'` to `cv.test_num()` and `cv.test_prob()` interventions.
- **GitHub info**: PR 593, head 4d8016fa

1.4.8.3 Version 1.4.6 (2020-06-01)

- Implemented continuous rescaling: dynamic rescaling can now be used with an arbitrarily small `rescale_factor`. The amount of rescaling on a given timestep is now either `rescale_factor` or the factor that would be required to bring the population below the threshold, whichever is larger.
- **Regression information**: Results should not be affected unless a simulation was run with too small of a rescaling factor. This change corrects this issue.
- **GitHub info**: PR 588, head f7ef0fa5

1.4.8.4 Version 1.4.5 (2020-05-31)

- Added `cv.date_range()`.
- Changed `cv.day()` and `cv.date()` to assume a start day of 2020-01-01 if not supplied.
- Added the option to add custom data to a `Fit` object, e.g. age histogram data.
- **GitHub info**: PR 585, head 4cabddc3

1.4.8.5 Version 1.4.4 (2020-05-31)

- Improved transmission tree histogram plotting, including allowing start and end days, and renamed `plot_histograms()`.
- Added functions for negative binomial distributions, allowing easier exploration of overdispersion effects: see `cv.make_random_contacts()`, and, most importantly, `pars['beta_dist']`.
- Renamed `cv.multinomial()` to `cv.n_multinomial()`.
- Added a `build_docs` script.
- **GitHub info**: PR 582, head 8bb8b82e

1.4.8.6 Version 1.4.3 (2020-05-30)

- Added `swab_delay` to `cv.test_prob()`, which behaves the same way as for `cv.test_num()` (to set the delay between experiencing symptoms and receiving a test).
- Allowed weights for a `Fit` to be specified as a time series.
- **GitHub info**: PR 573, head d84ffeef

1.4. What's new
1.4.8.7 Version 1.4.2 (2020-05-30)

- Renamed `cv.check_save_info()` to `cv.check_save_version()`, and allowed the die argument to be passed.
- Allowed verbose to be a float instead of an int; if between 0 and 1, during a model run, it will print out once every 1/verbose days, e.g. verbose = 0.2 will print an update once every 5 days.
- Updated the default number of household contacts from 2.7 to 2.0 for hybrid, and changed `cv.poisson()` to no longer cast to an integer. These two changes cancel out, so default behavior has not changed.
- Updated the calculation of contacts from household sizes (now uses household size - 1, to remove self-connections).
- Added `cv.MultiSim.load()`.
- Added Numba caching to `compute_viral_load()`, reducing overall Covasim load time by roughly 50%.
- Added an option for parallel execution of Numba functions (see `utils.py`); although this significantly improves performance (20-30%), it results in non-deterministic results, so is disabled by default.
- Changed `People` to use its own contact layer keys rather than those taken from the parameters.
- Improved plotting and corrected minor bugs in age histogram and model fit analyzers.

*Regression information:*
  - Replace `cv.check_save_info()` with `cv.check_save_version()`.
  - If you used a non-integer number of contacts, round down to the nearest integer (e.g., change 2.7 to 2.0).
  - If you loaded a household size distribution (e.g. `cv.Sim(location='nigeria')`), add one to the number of household contacts (but then round down).
- *GitHub info: PR 577, head 5569b88a

1.4.8.8 Version 1.4.1 (2020-05-29)

- Added `sim.people.plot()`, which shows the age distribution, and distribution of contacts by age and layer.
- Added `sim.make_age_histogram()`, as well as the ability to call `cv.age_histogram(sim)`, as an alternative to adding these as analyzers to a sim.
- Updated `cv.make_synthpop()` to pass a random seed to SynthPops (note: requires SynthPops version 0.7.1 or later).
- `cv.set_seed()` now also resets `random.seed()`, to ensure reproducibility among functions that use this (e.g., NetworkX).
- Corrected `sim.run()` so `sim.t` is left at the last timestep (instead of one more).
- *GitHub info: PR 574, head a828d29b

1.4.8.9 Version 1.4.0 (2020-05-28)

This version contains a large number of changes, including two new classes, Analyzer and Fit, for performing simulation analyses and fitting the model to data, respectively. These changes are described below.
Analysis

- Added a new class, `Analyzer`, to perform analyses on a simulation.
- Added a new parameter, `sim['analyzers']`, that operates like `interventions`: it accepts a list of functions or `Analyzer` objects.
- Added two analyzers: `cv.age_hist` records age histograms of infections, diagnoses, and deaths; `cv.snapshot` makes copies of the `People` object at specified points in time.

Fitting

- Added a new class, `cv.Fit()`, that stores information about the fit between the model and the data. “Likelihood” is no longer automatically calculated, but instead “mismatch” can be calculated via `fit = sim.compute_fit()`.
- The Poisson test that was previously used for the “likelihood” calculation has been deprecated; the new default mismatch is based on normalized absolute error.
- For a plot of how the mismatch is being calculated, use `fit.plot()`.

MultiSims

- Added `multisim.init_sims()`, which is not usually necessary, but can be helpful if you want to create the `Sim` objects without running them straight away.
- Added `multisim.split()`, easily allowing a merged multisim to be split back into its constituent parts (non-merged multisims can also be split). This can be used for example to create several multisims, merge them together, run them all at the same time in parallel, and then split the back for analysis.

Display functions

- Added `sim.summarize()`, which shows a short review of key sim results (cumulative counts).
- Added `sim.brief()`, which shows a one-line summary of the sim.
- Added `multisim.summarize()`, which prints a brief summary of all the constituent sims.

Parameter changes

- Removed the parameter `interv_func`; instead, intervention functions can now be appended to `sim['interventions']`.
- Changed the default for the `rescale` parameter from `False` to `True`. To return to previous behavior, define `sim['rescale'] = False` explicitly.

Other changes

- Added `cv.day()` convenience function to convert a date to an integer number of days (similar to `cv.daydiff()`); also modified `cv.date()` to be able to handle input more flexibly. While `sim.day()` and `sim.date()` are still the recommended functions, the same functionality is now also available without a `Sim` object available.
Covasim, Release 3.1.1

- Allowed `cv.load_data()` to accept non-time-series inputs.
- Added cumulative diagnoses to default plots.
- Moved `sweeps` (Weights & Biases) to `examples/wandb`.
- Refactored cruise ship example to work again.
- Various bugfixes (e.g. to plotting arguments, data scrapers, etc.).
- **Regression information:** To migrate an old parameter set `pars` to this version and to restore previous behavior, use:

```python
pars['analyzers'] = None  # Add the new parameter key
interv_func = pars.pop('interv_func', None)  # Remove the deprecated key
if interv_func:
    pars['interventions'] = interv_func  # If no interventions
    pars['interventions'].append(interv_func)  # If other interventions are present
pars['rescale'] = pars.pop('rescale', False)  # Change default to False
```

- **GitHub info:** PR 569, head 2dcf6ad8

### 1.4.9 Versions 1.3.x (1.3.0 – 1.3.5)

#### 1.4.9.1 Version 1.3.5 (2020-05-28)

- Added `swab_delay` argument to `cv.test_num()`, allowing a distribution of times between when a person develops symptoms and when they go to be tested (i.e., receive a swab) to be specified.
  - **GitHub info:** PR 566, head 19dcfdd7

#### 1.4.9.2 Version 1.3.4 (2020-05-26)

- Allowed data to be loaded from a dataframe instead of from file.
- Fixed data scrapers to use correct column labels.
  - **GitHub info:** PR 568, head 8b157a26

#### 1.4.9.3 Version 1.3.3 (2020-05-26)

- Fixed issue with a loaded population being reloaded when a simulation is re-initialized.
- Fixed issue with the argument `dateformat` not being passed to the right plotting routine.
- Fixed issue with MultiSim plotting appearing in separate panels when run in a Jupyter notebook.
- Fixed issue with `cv.git_info()` failing to write to file when the calling function could not be found.
  - **GitHub info:** PR 567, head d1b2bc40

#### 1.4.9.4 Version 1.3.2 (2020-05-25)

- `People` and `popdict` objects can now be supplied directly to the sim instead of a file name.
- `git_info()` and `check_save_info()` now include information from the calling script (not just Covasim). They also now include a comments field to optionally store additional information.
  - **GitHub info:** PR 562, head a943bb9e
1.4.9.5 Version 1.3.1 (2020-05-25)

• Modified calculation of $R_{eff}$ to include a longer integration period at the beginning, and restored previous method of creating seed infections.

• Updated default plots to include number of active infections, and removed recoveries.

• GitHub info: PR 561, head 6c91a32c

1.4.9.6 Version 1.3.0 (2020-05-24)

• Changed the default number of work contacts in hybrid from 8 to 16, and halved beta from 1.4 to 0.7, to better capture superspreading events. Regression information: To restore previous behavior, set sim['beta_layer']['w'] = 0.14 and sim['contacts']['w'] = 8.

• Initial infections now occur at a distribution of dates instead of all at once; this fixes the artificial spike in $R_{eff}$ that occurred at the very beginning of a simulation. Regression information: This change affects results, but was reverted in the next version (1.3.1).

• Changed the definition of age bins in prognoses to be lower limits rather than upper limits. Added an extra set of age bins for 90+.

• Changed population loading and saving to be based on People objects, not popdicts (syntax is exactly the same, although it is recommended to use .ppl instead of .pop for these files).

• Added additional random seed resets to population initialization and just before the run so that populations loaded from disk produce identical results to newly created ones. Regression information: This affects results by changing the random number stream. In most cases, previous behavior can typically be restored by setting sim.run(reset_seed=False).

• Added a new convenience method, cv.check_save_info(), which can be put at the top of a script to check the Covasim version and automatically save the Git info to file.

• Added additional methods to People to retrieve different types of keys: e.g., sim.people.state_keys() returns all the different states a person can be in (e.g., symptomatic).

• GitHub info: PR 557, head 32c5e1e3

1.4.10 Versions 1.2.x (1.2.0 – 1.2.3)

1.4.10.1 Version 1.2.3 (2020-05-23)

• Added cv.savefig(), which is an alias to Matplotlib’s savefig() function, but which saves additional metadata in the figure file. This metadata can be loaded with the new cv.get_png_metadata() function.

• Major changes to MultiSim plotting, incorporating all the flexibility of both simulation and scenario plotting. By default, with a small number of runs (<= 5), it defaults to scenario-style plotting; else, it defaults to simulation-style plotting.

• Default scenario plotting options were updated (e.g., showing deaths instead of hospitalizations).

• You may merge multiple multisims more merrily now, with e.g. msim = cv.MultiSim.merge(msim1, msim2).

• Test scripts (e.g. tests/run_tests) have been updated to use pytest-parallel, reducing wall-clock time by a factor of 5.

• GitHub info: PR 552, head 3c1ca8b3

1.4. What’s new
### 1.4.10.2 Version 1.2.2 (2020-05-22)

- Changed the syntax of `cv.clip_edges()` to match `cv.change_beta()`. The old format of intervention `cv.clip_edges(start_day=d1, end_day=d2, change=c)` should now be written as `cv.clip_edges(days=[d1, d2], changes=[c, 1.0])`.
- Changed the syntax for the transmission tree: it now takes the `Sim` object rather than the `People` object, and typical usage is now `tt = sim.make_transtree()`.
- Plots now default to a maximum of 4 rows; this can be overridden using the `n_cols` argument, e.g. `sim.plot(to_plot='overview', n_cols=2)`.
- Various bugs with `MultiSim` plotting were fixed.

**GitHub info**: PR 551, head 28bf02b5

### 1.4.10.3 Version 1.2.1 (2020-05-21)

- Added influenza-like illness (ILI) symptoms to testing interventions. If nonzero, this reduces the effectiveness of symptomatic testing, because you cannot distinguish between people who are symptomatic with COVID and people with other ILI symptoms.
- Removed an unneeded `copy()` in `single_run()` because multiprocessing always produces copies of objects via the pickling process.

**GitHub info**: PR 541, head 07009eb9

### 1.4.10.4 Version 1.2.0 (2020-05-20)

- Since parameters can be modified during the run, previously, the sim could not be rerun with the guarantee that the results would be the same. `sim.run()` now has a `restore_pars` argument (default true), which makes a copy of the parameters just prior to the run to ensure reproducibility.
- In plotting, by default, data points are now slightly transparent and behind the lines to improve visibility of the model curve.
- Interventions now have a `label` attribute, which can be helpful for finding them if many are used, e.g. `[interv for interv in sim['interventions'] if interv.label=='Close schools']`. There is also a new method, `intervention.disp()`, which prints out detailed information about an intervention object.
- Subtargeting of particular people in testing interventions can now be done via a function that gets called dynamically, avoiding the need to initialize the sim prior to creating the intervention.
- Layer keys are now stored inside the `popdict`, for greater consistency handling loaded populations. Layer key handling has been simplified and made more robust.
- Loading and saving a population is now controlled by the `Sim` object, not by the `sim.initialize()` method. Instead of `sim = cv.Sim(); sim.initialize(save_pop=True)`, you can now simply do `sim = cv.Sim(save_pop=True)`, and it will save when the sim is initialized.
- Added prevalence and incidence as results.
- Added `sim.scaled_pop_size`, which is the population size (the number of agents) times the population scale factor. This corresponds to the "actual" population size being modeled.
- Removed the numerical artifact at the beginning and end of the $R_{eff}$ calculation due to the smoothing kernel, and confirmed that the spike in $R_{eff}$ often seen at the beginning is due to the way the seed infectious progress from exposed to infectious, and not from a bug.
• Added more flexibility to plotting, including a new show_args keyword, allowing particular aspects of plotting (e.g., the data or interventions) to be turned on or off.

• Moved the cruise ship code from the core folder into the examples folder.

• GitHub info: PR 538, head 9b2dbfba

1.4.11 Versions 1.1.x (1.1.0 – 1.1.7)

1.4.11.1 Version 1.1.7 (2020-05-19)

• Diagnoses are now reported on the day the test was conducted, not the day the person gets their diagnosis. This is to better align with data (which is reported this way), and to avoid a bug in which test yield could be >100%. A new attribute, date_pos_test, was added to the sim.people object in order to track the date on which a person is given the test which will (after test_delay days) come back positive.

• An “overview” plotting feature has been added for sims and scenarios: simply use sim.plot(to_plot='overview') to use. This plots almost all of the simulation outputs on one screen.

• It is now possible to set pop_type = None if you are supplying a custom population.

• Population creation functions (including the People class) have been tidied up with additional docstrings added.

• Duplication between pre- and post-step state checking has been removed.

• GitHub info: PR 537, head 451f4100

1.4.11.2 Version 1.1.6 (2020-05-19)

• Created an analysis.py file to support different types of analysis.

• Moved transtree from sim.people into its own class: thus instead of sim.people.make_detailed_transtree(), the new syntax is tt = cv.TransTree(sim.people).

• GitHub info: PR 531, head 2d55c380

1.4.11.3 Version 1.1.5 (2020-05-18)

• Added extra flexibility for targeting interventions by index of a person, for example, by age.

• GitHub info: head fda4cc17

1.4.11.4 Version 1.1.4 (2020-05-18)

• Added a new hospital bed capacity constraint and renamed health system capacity parameters. To migrate an older set of parameters to this version, set:

  ```
  pars['no_icu_factor'] = pars.pop('OR_no_treat')
  pars['n_beds_icu'] = pars.pop('n_beds')
  pars['no_hosp_factor'] = 1.0
  pars['n_beds_hosp'] = None
  ```

• Removed the bed_capacity result.

• GitHub info: PR 510, head 81261f90
1.4.11.5 Version 1.1.3 (2020-05-18)

- Improved the how “layer parameters” (e.g., beta_layer) are initialized.
- Allowed arbitrary arguments to be passed to SynthPops via cv.make_synthpop.
- GitHub info: head 0f6d48c0

1.4.11.6 Version 1.1.2 (2020-05-18)

- Added a new result, test_yield, which is the number of diagnoses divided by the number of cases each day.
- Minor improvements to date handling and plotting.
- GitHub info: head 6f2f0455

1.4.11.7 Version 1.1.1 (2020-05-13)

- Refactored the contact tracing and quarantining functions, to fixed a bug (introduced in v1.1.0) in which some people who went into quarantine never came out of quarantine.
- Changed initialization so seed infections are now sampled randomly from the population, rather than the first pop_infected agents. Since hybrid also uses consecutive indices for constructing households, this was causing some households to be fully infected on initialization, while all other households had no infections.
- Updated the default rescale_factor from 2.0 to 1.2, since large amounts of rescaling cause noticeable “blips” in inhomogeneous networks (e.g., a population where some households are 100% infected and most are 0% infected).
- Added ability to pass plotting arguments to intervention.plot().
- Removed default noise in scenarios (restore previous behavior by setting metapars = dict(noise=0.1)).
- Refactored and renamed computed results (e.g., summary stats) in the Sim class.
- GitHub info: PR 513, head 2332c319

1.4.11.8 Version 1.1.0 (2020-05-12)

- Renamed the parameter diag_factor to iso_factor, and converted it to a dictionary by layer.
- Renamed the parameter quar_eff to quar_factor (but otherwise left it unchanged).
- Added the option for presumptive isolation and quarantine in testing interventions.
- Fixed a bug whereby people who had been in quarantine and were then diagnosed had both diagnosis and quarantine factors applied.
- GitHub info: PR 502, head 973801a6

1.4.12 Versions 1.0.x (1.0.0 – 1.0.3)

1.4.12.1 Version 1.0.3 (2020-05-11)

- Added an extra output of make_microstructured_contacts() to store each person’s cluster identifier. Currently, this is only supported for the hybrid population type, but in future versions, synthpops will also be supported.
• Removed the directed argument from population creation functions since it is no longer supported in the model.
  • GitHub info: head 57f58480

1.4.12.2 Version 1.0.2 (2020-05-10)
  • Added uncertainty to the plot_result() method of MultiSims.
  • Added documentation and webapp links to the paper.
  • GitHub info: head 6811bc59

1.4.12.3 Version 1.0.1 (2020-05-09)
  • Added argument as_date for sim.date() to return a datetime object instead of a string.
  • Fixed plotting of interventions in the webapp.
  • Removed default 1-hour time limit for simulations.
  • GitHub info: PR 490, head 1e08cc9a

1.4.12.4 Version 1.0.0 (2020-05-08)
  • Official release of Covasim.
  • Made scenario and simulation plotting more flexible: to_plot can now simply be a list of results keys, e.g. cum_deaths.
  • Added additional tests, increasing test coverage from 67% to 92%.
  • Fixed bug in cv.save().
  • Added reset() to MultiSim that undoes a reduce() or combine() call.
  • General code cleaning: made exceptions raised more consistent, removed unused functions, etc.
  • GitHub info: PR 487, head 9a6c23b

1.4.13 Prerelease versions (0.27.0 – 0.32.1)

1.4.13.1 Version 0.32.1 (2020-05-06)
  • Allow until to be a date, e.g. sim.run(until='2020-05-06').
  • Added ipywidgets dependency since otherwise the webapp breaks due to a bug with the latest Plotly version (4.7).
  • GitHub info: head c8ca32d

1.4.13.2 Version 0.32.0 (2020-05-05)
  • Changed the edges of the contact network from being directed to undirected, halving the amount of memory required and making contact tracing and edge clipping more realistic.
  • Added comorbidities to the prognoses parameters.
1.4.13.3 Version 0.31.0 (2020-05-05)

- Added age-susceptible odds ratios, and modified severe and critical progression probabilities. To compensate, default beta has been increased from 0.015 to 0.016. To restore previous behavior (which was based on the Imperial paper), set beta=0.015 and set the following values in sim.pars['prognoses']:

```python
sus_ORs[:] = 1.0
severe_probs = np.array([0.00100, 0.00100, 0.01100, 0.03400, 0.04300, 0.08200, 0.11800, 0.16600, 0.18400])
crit_probs = np.array([0.00004, 0.00011, 0.00050, 0.00123, 0.00214, 0.00800, 0.02750, 0.06000, 0.10333])
```

- Relative susceptibility and transmissibility (i.e., sim.people.rel_sus) are now set when the population is initialized (before, they were modified dynamically when a person became infected or recovered). This means that modifying them before a simulation starts, or during a simulation, should be more robust.

- Reordered results dictionary to start with cumulative counts.

- `sim.export_pars()` now accepts a filename to save to.

- Added a tests/regression folder with previous versions of default parameter values.

- Changed `pars['n_beds']` to interpret 0 or None as no bed constraint.

- GitHub info: PR 482

1.4.13.4 Version 0.30.4 (2020-05-04)

- Changed the detailed transmission tree (`sim.people.transtree.detailed`) to include much more information.

- Added animation method to transmission tree: `sim.people.transtree.animate()`.

- Added support to generate populations on the fly in SynthPops.

- Adjusted the default arguments for `test_prob` and fixed a bug with `test_num` not accepting date input.

- Added `tests/devtests/intervention_showcase.py`, using and comparing all available interventions.

1.4.13.5 Version 0.30.3 (2020-05-03)

- Fixed bugs in dynamic scaling; see `tests/devtests/dev_test_rescaling.py`. When using `pop_scale>1`, the recommendation is now to use `rescale=True`.

- In `cv.test_num()`, renamed argument from `sympt_test` to `symp_test` for consistency.

- Added `plot_compare()` method to MultiSim.

- Added `labels` arguments to plotting methods, to allow custom labels to be used.
1.4.13.6 Version 0.30.2 (2020-05-02)

- Updated \texttt{r\_eff} to use a new method based on daily new infections. The previous version, where infections were counted from when someone recovered or died, is available as \texttt{sim.compute_r\_eff(method='outcome')}, while the traditional method, where infections are counted from the day someone becomes infectious, is available via \texttt{sim.compute_r\_eff(method='infectious')}.

1.4.13.7 Version 0.30.1 (2020-05-02)

- Added \texttt{end\_day} as a parameter, allowing an end date to be specified instead of a number of days.
- \texttt{Sim.run()} now displays the date being simulated.
- Added a \texttt{par\_args} argument to \texttt{multi\_run()}, allowing arguments (e.g. \texttt{ncpus}) to be passed to \texttt{sc.parallelize()}.
- Added a \texttt{compare()} method to multisims and stopped people from being saved by default.
- Fixed bug whereby intervention were not getting initialized if they were added to a sim after it was initialized.

1.4.13.8 Version 0.30.0 (2020-05-02)

- Added new \texttt{MultiSim} class for plotting a single simulation with uncertainty.
- Added \texttt{low} and \texttt{high} attributes to the \texttt{Result} object.
- Refactored plotting to increase consistency between \texttt{sim.plot()}, \texttt{sim.plot\_result()}, \texttt{scens.plot()}, and \texttt{multisim.plot()}.
- Doubling time calculation defaults have been updated to use a window of 3 days and a maximum of 30 days.
- Added an \texttt{until} argument to \texttt{sim.run()}, to make it easier to run a partially completed sim and then resume. See \texttt{tests/devtests/test_run\_until.py}.
- Fixed a bug whereby \texttt{cv.clip\_edges()} with no end day specified resulted in large sim files when saved.

1.4.13.9 Version 0.29.9 (2020-04-28)

- Fixed bug in which people who had been tested and since recovered were not being diagnosed.
- Updated definition of “Time to die” parameter in the webapp.

1.4.13.10 Version 0.29.8 (2020-04-28)

- Updated webapp UI with more detail on and control over interventions.

1.4.13.11 Version 0.29.7 (2020-04-27)

- New functions \texttt{cv.date()} and \texttt{cv.daydiff()} have been added, to ease handling of dates of different formats.
- Defaults are now functions rather than dictionaries, specifically: \texttt{cv.default_sim\_plots} is now \texttt{cv.get\_sim\_plots(); cv.default\_scen\_plots} is now \texttt{cv.get\_scen\_plots();} and \texttt{cv.default\_colors} is now \texttt{cv.get\_colors();}
- Interventions now have a \texttt{do\_plot} kwarg, which if \texttt{False} will disable their plotting.
• The example scenario (examples/run_scenario.py) has been rewritten to include a test-trace-quarantine example.

1.4.13.12 Version 0.29.6 (2020-04-27)

• Updated to use Sciris v0.17.0, to fix JSON export issues and improve KeyError messages.

1.4.13.13 Version 0.29.5 (2020-04-26)

• Fixed bug whereby layer betas were applied twice, and updated default values.
• Includes individual-level viral load (to use previous results, set pars['beta_dist'] = {'dist': 'lognormal', 'par1':1.0, 'par2':0.0} and pars['viral_dist'] = {'frac_time':0.0, 'load_ratio':1, 'high_cap':0}).
• Updated parameter values (mostly durations) based on revised literature review.
• Added sim.export_pars() and sim.export_results() methods.
• Interventions can now be converted to/from JSON – automatically when loading a parameters dictionary into a sim, or manually using cv.InterventionDict().
• Improvements to transmission trees: can now make a detailed tree with sim.people.make_detailed_transintree() (replacing sim.people.transtree.make_detailed(sim.people)), and can plot via sim.people.transtree.plot().
• Improved date handling, so most functions are now agnostic as to whether a date string, datetime object, or number of days is provided; new functions: sim.day() converts dates to days, sim.date() (formerly sim.inds2dates()) converts days to dates, and sim.daydiff() computes the number of days between two dates.

1.4.13.14 Version 0.28.8 (2020-04-24)

• Includes data on household sizes from various countries.
• Includes age data on US states.
• Changes to interventions to include end as well as start days, and plotting as a default option.
• Adds version checks to loading and introduces a new function cv.load() to replace e.g. cv.Sim.load().
• Major layout and functionality changes to the webapp, including country selection (disabled by default).
• Provided access to Plotly graphs via the backend.
• Moved relative probabilities (e.g. rel_death_prob) from population creation to loop so can be modified dynamically.
• Introduced cv.clip_edges() intervention, similar to cv.change_beta() but removes contacts entirely.

1.4.13.15 Version 0.28.1 (2020-04-19)

• Major refactor of transmission trees, including additional detail via sim.people.transtree.make_detailed().
• Counting of diagnoses before and after interventions on each timestep (allowing people to go into quarantine on the same day).
• Improved saving of people in scenarios, and updated keyword for sims(sim.save(keep_people=True)).
1.4.13.16 Version 0.28.0 (2020-04-19)

- Includes dynamic per-person viral load.
- Refactored data types.
- Changed how populations are handled, including adding a `dynam_layer` parameter to specify which layers are dynamic.
- Disease progression duration parameters were updated to be longer.
- Fixed bugs with quarantine.
- Fixed bug with hybrid school and work contacts.
- Changed contact tracing to be only for contacts with nonzero transmission.

1.4.13.17 Version 0.27.12 (2020-04-17)

- Caches Numba functions, reducing load time from 2.5 to 0.5 seconds.
- Pins Numba to 0.48, which is 10x faster than 0.49.
- Fixed issue with saving populations in scenarios.
- Refactored how populations are handled, removing `use_layers` parameter (use `pop_type` instead).
- Removed layer key from layer object, reducing total sim memory footprint by 3x.
- Improved handling of mismatches between loaded population layers and simulation parameters.
- Added custom key errors to handle multiline error messages.
- Fix several issues with probability-based testing.
- Changed how layer betas are applied (inside the sim rather than statically).
- Added more detail to the transmission tree.
- Refactored random population calculation, speeding up large populations (>100k) by a factor of 10.
- Added documentation.

1.4.13.18 Version 0.27.0 (2020-04-16)

- Refactor calculations to be vector-based rather than object based.
- Include factors for per-person viral load (transmissibility) and susceptibility.
- Started a changelog (needless to say).

1.5 Parameters

This file describes each of the input parameters in Covasim. Note: the overall infection rate can be explored using `sim.results['doubling_time']` and `sim.results['r_eff']` (a higher infection rate means lower doubling times and higher $R_{eff}$), as well as by simply looking at the epidemic curves.
1.5.1 Population parameters

- `pop_size` = Number of agents, i.e., people susceptible to SARS-CoV-2
- `pop_infected` = Number of initial infections
- `pop_type` = What type of population data to use – ‘random’ (fastest), ‘synthpops’ (best), ‘hybrid’ (compromise)
- `location` = What location to load data from – default Seattle

1.5.2 Simulation parameters

- `start_day` = Start day of the simulation
- `end_day` = End day of the simulation
- `n_days` = Number of days to run, if end_day isn’t specified
- `rand_seed` = Random seed, if None, don’t reset
- `verbose` = Whether or not to display information during the run – options are 0 (silent), 0.1 (some; default), 1 (more), 2 (everything)

1.5.3 Rescaling parameters

- `pop_scale` = Factor by which to scale the population – e.g. 1000 with pop_size = 10e3 means a population of 10m
- `scaled_pop` = The total scaled population, i.e. the number of agents times the scale factor; alternative to pop_scale
- `rescale` = Enable dynamic rescaling of the population
- `rescale_threshold` = Fraction susceptible population that will trigger rescaling if rescaling
- `rescale_factor` = Factor by which we rescale the population

1.5.4 Basic disease transmission

- `beta` = Beta per symptomatic contact; absolute
- `n_imports` = Average daily number of imported cases (actual number is drawn from Poisson distribution)
- `beta_dist` = Distribution to draw individual level transmissibility; see https://wellcomeopenresearch.org/articles/5-67
- `viral_dist` = The time varying viral load (transmissibility); estimated from Lescure 2020, Lancet, https://doi.org/10.1016/S1473-3099(20)30200-0
- `asymp_factor` = Multiply beta by this factor for asymptomatic cases; no statistically significant difference in transmissibility: https://www.sciencedirect.com/science/article/pii/S1201971220302502

1.5.5 Network parameters

- `contacts` = The number of contacts per layer
- `dynam_layer` = Which layers are dynamic
• beta_layer = Transmissibility per layer

1.5.6 Multi-strain parameters

• n_imports = Average daily number of imported cases (actual number is drawn from Poisson distribution)
• n_strains = The number of strains circulating in the population

1.5.7 Immunity parameters

• use_waning = Whether to use dynamically calculated immunity
• nab_init = Parameters for the distribution of the initial level of log2(nab) following natural infection, taken from fig1b of https://doi.org/10.1101/2021.03.09.21252641
• nab_decay = Parameters describing the kinetics of decay of nab over time, taken from fig3b of https://doi.org/10.1101/2021.03.09.21252641
• nab_kin = Constructed during sim initialization using the nab_decay parameters
• nab_boost = Multiplicative factor applied to a person’s nab levels if they get reinfected. # TODO: add source
• nab_eff = Parameters to map nab to efficacy
• rel_imm_symp = Relative immunity from natural infection varies by symptoms
• immunity = Matrix of immunity and cross-immunity factors, set by init_immunity() in immunity.py

1.5.8 Strain-specific parameters

• rel_beta = Relative transmissibility varies by strain
• rel_imm_strain = Relative own-immunity varies by strain

1.5.9 Time for disease progression

• exp2inf = Duration from exposed to infectious; see Lauer et al., https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7081172/, subtracting inf2sym duration
• inf2sym = Duration from infectious to symptomatic; see Linton et al., https://doi.org/10.3390/jcm9020538
• sym2sev = Duration from symptomatic to severe symptoms; see Linton et al., https://doi.org/10.3390/jcm9020538Duration from severe symptoms to requiring ICU; see Wang et al., https://jamanetwork.com/journals/jama/fullarticle/2761044Duration from severe symptoms to requiring ICU

1.5.10 Time for disease recovery

• asym2rec = Duration for asymptomatic people to recover; see Wölfel et al., https://www.nature.com/articles/s41586-020-2196-x
• mild2rec = Duration for people with mild symptoms to recover; see Wölfel et al., https://www.nature.com/articles/s41586-020-2196-x
• sev2rec = Duration for people with severe symptoms to recover, 22.6 days total; see Verity et al., https://www.medrxiv.org/content/10.1101/2020.03.09.20033357v1.full.pdf
• crit2rec = Duration for people with critical symptoms to recover, 22.6 days total; see Verity et al., https://www.medrxiv.org/content/10.1101/2020.03.09.20033357v1.full.pdf
• crit2die = Duration from critical symptoms to death, 17.8 days total; see Verity et al., https://www.medrxiv.org/content/10.1101/2020.03.09.20033357v1.full.pdf

1.5.11 Severity parameters

• rel_symp_prob = Scale factor for proportion of symptomatic cases
• rel_severe_prob = Scale factor for proportion of symptomatic cases that become severe
• rel_crit_prob = Scale factor for proportion of severe cases that become critical
• rel_death_prob = Scale factor for proportion of critical cases that result in death
• prog_by_age = Whether to set disease progression based on the person’s age
• prognoses = The actual arrays of prognoses by age; this is populated later

1.5.12 Efficacy of protection measures

• iso_factor = Multiply beta by this factor for diagnosed cases to represent isolation; set below
• quar_factor = Quarantine multiplier on transmissibility and susceptibility; set below
• quar_period = Number of days to quarantine for; assumption based on standard policies

1.5.13 Events and interventions

• interventions = The interventions present in this simulation; populated by the user
• analyzers = Custom analysis functions; populated by the user
• timelimit = Time limit for the simulation (seconds)
• stopping_func = A function to call to stop the sim partway through

1.5.14 Health system parameters

• n_beds_hosp The number of hospital (adult acute care) beds available for severely ill patients (default is no constraint)
• n_beds_icu The number of ICU beds available for critically ill patients (default is no constraint)
• no_hosp_factor Multiplier for how much more likely severely ill people are to become critical if no hospital beds are available
• no_icu_factor Multiplier for how much more likely critically ill people are to die if no ICU beds are available

1.6 Data scrapers

These scripts pull data from various sources for use in Covasim. To run all scrapers, simply type
1.6.1 1. Corona Data Scraper

To quote the Corona Data Scraper web page,

Corona Data Scraper pulls COVID-19 Coronavirus case data from verified sources.

These are scraped by the loader below, and placed in the `data/epi_data/corona-data-scraper-project` directory. The data is in CSV format.

Here is a sample of the data.
<table>
<thead>
<tr>
<th>key</th>
<th>population</th>
<th>aggr</th>
<th>cum positives</th>
<th>cum tests</th>
<th>cum hospitalized</th>
<th>cum discharged</th>
<th>cumulative_date</th>
<th>dance</th>
<th>positives</th>
<th>death</th>
<th>tests</th>
<th>discharged</th>
<th>recovered</th>
<th>active</th>
</tr>
</thead>
<tbody>
<tr>
<td>5708</td>
<td>Roane County, Tennessee, United States</td>
<td>County</td>
<td>0</td>
<td>0.0</td>
<td>2020-03-21</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5709</td>
<td>Roane County, Tennessee, United States</td>
<td>County</td>
<td>0</td>
<td>0.0</td>
<td>2020-03-22</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5709</td>
<td>Roane County, Tennessee, United States</td>
<td>County</td>
<td>0</td>
<td>0.0</td>
<td>2020-03-23</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5709</td>
<td>Roane County, Tennessee, United States</td>
<td>County</td>
<td>0</td>
<td>0.0</td>
<td>2020-03-24</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5709</td>
<td>Roane County, Tennessee, United States</td>
<td>County</td>
<td>0</td>
<td>0.0</td>
<td>2020-03-25</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5709</td>
<td>Roane County, Tennessee, United States</td>
<td>County</td>
<td>0</td>
<td>0.0</td>
<td>2020-03-26</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5709</td>
<td>Roane County, Tennessee, United States</td>
<td>County</td>
<td>0</td>
<td>0.0</td>
<td>2020-03-27</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5709</td>
<td>Roane County, Tennessee, United States</td>
<td>County</td>
<td>0</td>
<td>0.0</td>
<td>2020-03-28</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5709</td>
<td>Roane County, Tennessee, United States</td>
<td>County</td>
<td>0</td>
<td>0.0</td>
<td>2020-03-29</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5709</td>
<td>Roane County, Tennessee, United States</td>
<td>County</td>
<td>0</td>
<td>0.0</td>
<td>2020-03-30</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5709</td>
<td>Roane County, Tennessee, United States</td>
<td>County</td>
<td>0</td>
<td>0.0</td>
<td>2020-03-31</td>
<td>88.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5709</td>
<td>Roane County, Tennessee, United States</td>
<td>County</td>
<td>0</td>
<td>0.0</td>
<td>2020-04-01</td>
<td>91.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5709</td>
<td>Roane County, Tennessee, United States</td>
<td>County</td>
<td>0</td>
<td>0.0</td>
<td>2020-04-02</td>
<td>121.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5709</td>
<td>Roane County, Tennessee, United States</td>
<td>County</td>
<td>0</td>
<td>0.0</td>
<td>2020-04-03</td>
<td>150.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Updating: To update the Corona Data Scraper data,

```
python data/load_corona_data_scraper_data.py
```

As of April 4, 2020, there are apparently 3874 data sets.

### 1.6.2 2. European Centre for Disease Prevention and Control

To quote the [European Centre for Disease Prevention and Control](https://www.ecdc.europa.eu/en) web page,

> Since the beginning of the coronavirus pandemic, ECDC’s Epidemic Intelligence team has been collecting the number of COVID-19 cases and deaths, based on reports from health authorities worldwide. This comprehensive and systematic process is carried out on a daily basis. To insure the accuracy and reliability of the data, this process is being constantly refined. This helps to monitor and interpret the dynamics of the COVID-19 pandemic not only in the European Union (EU), the European Economic Area (EEA), but also worldwide.

The data is stored in CSV format in `data/epi_data/european-centre-for-disease-prevention-and-control`

Here is a sample of the data:

<table>
<thead>
<tr>
<th>day</th>
<th>new_positives</th>
<th>new_death</th>
<th>key</th>
<th>population</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>3960</td>
<td>0</td>
<td>0</td>
<td>Greenland</td>
<td>56025.0</td>
<td>2020-03-20</td>
</tr>
<tr>
<td>3959</td>
<td>1</td>
<td>0</td>
<td>Greenland</td>
<td>56025.0</td>
<td>2020-03-21</td>
</tr>
<tr>
<td>3958</td>
<td>2</td>
<td>0</td>
<td>Greenland</td>
<td>56025.0</td>
<td>2020-03-22</td>
</tr>
<tr>
<td>3957</td>
<td>3</td>
<td>0</td>
<td>Greenland</td>
<td>56025.0</td>
<td>2020-03-23</td>
</tr>
<tr>
<td>3956</td>
<td>4</td>
<td>2</td>
<td>Greenland</td>
<td>56025.0</td>
<td>2020-03-24</td>
</tr>
<tr>
<td>3955</td>
<td>5</td>
<td>0</td>
<td>Greenland</td>
<td>56025.0</td>
<td>2020-03-25</td>
</tr>
<tr>
<td>3954</td>
<td>6</td>
<td>1</td>
<td>Greenland</td>
<td>56025.0</td>
<td>2020-03-26</td>
</tr>
<tr>
<td>3953</td>
<td>7</td>
<td>1</td>
<td>Greenland</td>
<td>56025.0</td>
<td>2020-03-27</td>
</tr>
<tr>
<td>3952</td>
<td>8</td>
<td>3</td>
<td>Greenland</td>
<td>56025.0</td>
<td>2020-03-28</td>
</tr>
<tr>
<td>3951</td>
<td>9</td>
<td>1</td>
<td>Greenland</td>
<td>56025.0</td>
<td>2020-03-29</td>
</tr>
<tr>
<td>3950</td>
<td>10</td>
<td>0</td>
<td>Greenland</td>
<td>56025.0</td>
<td>2020-03-30</td>
</tr>
<tr>
<td>3949</td>
<td>11</td>
<td>0</td>
<td>Greenland</td>
<td>56025.0</td>
<td>2020-03-31</td>
</tr>
<tr>
<td>3948</td>
<td>12</td>
<td>0</td>
<td>Greenland</td>
<td>56025.0</td>
<td>2020-04-01</td>
</tr>
<tr>
<td>3947</td>
<td>13</td>
<td>0</td>
<td>Greenland</td>
<td>56025.0</td>
<td>2020-04-02</td>
</tr>
</tbody>
</table>

Updating: To update the Corona Data Scraper data,

```
python data/load_ecdp_data.py
```


### 1.6.3 3. The COVID Tracking Project

The COVID Tracking Project “obtains, organizes, and publishes high-quality data required to understand and respond to the COVID-19 outbreak in the United States.” The project website is [https://covidtracking.com](https://covidtracking.com)

We transform this data for use in the Covasim parameter format. It is stored in CSV-format in the `ata/epi_data/covid-tracking-project` directory.
<table>
<thead>
<tr>
<th></th>
<th>date</th>
<th>key</th>
<th>cum_hospitalized</th>
<th>death</th>
<th>num_icu</th>
<th>num_on_ventilator</th>
<th>num_tests</th>
<th>num_positives</th>
<th>num_negatives</th>
<th>new_positives</th>
<th>new_tests</th>
<th>new_hospitalized</th>
<th>new_in_icu</th>
<th>new_on_ventilator</th>
<th>new_death</th>
<th>new_tests</th>
<th>new_negatives</th>
<th>new_hospitalized</th>
<th>new_death</th>
</tr>
</thead>
<tbody>
<tr>
<td>2210</td>
<td>2020-03-04</td>
<td>NY</td>
<td>0.0</td>
<td>0.0</td>
<td>28.0</td>
<td>16.0</td>
<td>44.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>2191</td>
<td>2020-03-05</td>
<td>NY</td>
<td>0.0</td>
<td>0.0</td>
<td>28.0</td>
<td>16.0</td>
<td>44.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>2163</td>
<td>2020-03-06</td>
<td>NY</td>
<td>0.0</td>
<td>0.0</td>
<td>16.0</td>
<td>11.0</td>
<td>27.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>2122</td>
<td>2020-03-07</td>
<td>NY</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>43.0</td>
<td>43.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>2071</td>
<td>2020-03-08</td>
<td>NY</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>29.0</td>
<td>29.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>2020</td>
<td>2020-03-09</td>
<td>NY</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>37.0</td>
<td>37.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>1969</td>
<td>2020-03-10</td>
<td>NY</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>31.0</td>
<td>31.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>1918</td>
<td>2020-03-11</td>
<td>NY</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>43.0</td>
<td>43.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>1867</td>
<td>2020-03-12</td>
<td>NY</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>1816</td>
<td>2020-03-13</td>
<td>NY</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>2687.0</td>
<td>205.0</td>
<td>2892.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>1765</td>
<td>2020-03-14</td>
<td>NY</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>103.0</td>
<td>103.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>1714</td>
<td>2020-03-15</td>
<td>NY</td>
<td>3.0</td>
<td>3.0</td>
<td>0.0</td>
<td>1764.0</td>
<td>205.0</td>
<td>1969.011</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>1661</td>
<td>2020-03-16</td>
<td>NY</td>
<td>7.0</td>
<td>4.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>221.0</td>
<td>221.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>1605</td>
<td>2020-03-17</td>
<td>NY</td>
<td>7.0</td>
<td>0.0</td>
<td>0.0</td>
<td>963.0</td>
<td>750.0</td>
<td>1713.013</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Updating:** To update the COVID Tracking Project data,

```
python data/load_covid_tracking_project_data.py
```

### 1.6.4 4. Demographic data scraper

To scrape demographic data, run
1.7 Glossary

This document contains a list of commonly used terms, both in Covasim and for COVID-19 more generally. If there are others you’d like to see included, please email us at info@covasim.org. Cross-references are listed in italics.

- **agent**: The basic unit of the model. Typically, one agent in the simulation corresponds to one person in the population, but the population may also be subsampled, such that a larger population is represented by a smaller number of agents. This depends on the assumption that the full population behaves the same way that a subset of the population behaves. See also dynamic rescaling.

- **agent-based model**: The modeling approach that tracks individual agents in the population. Compared to compartmental model, agent-based models are typically slower, but allow much greater detail in simulations. Covasim is an agent-based model.

- **back-tracking**: The process of finding the person who infected a person who has just been diagnosed.

- **basic reproduction number**: See reproduction number.

- **beta**: The probability of transmission from an infectious person to a susceptible person, also known as infectiousness or transmissibility. The overall transmission probability depends on the network layer two people are connected by, as well as the infected person’s viral load, the susceptible person’s age, and other factors.

- **case**: A person who has tested positive (i.e., been diagnosed) with COVID-19.

- **case detection ratio** (CDR): The proportion of all infections that have been diagnosed. For example, if 10,000 people have been infected and 1,500 have been diagnosed, the CDR will be 15%.

- **case fatality ratio** (CFR): The proportion of people who have been diagnosed who eventually die. This is typically much higher than the infection fatality ratio since more severe cases are more likely to be diagnosed. Typical CFRs are 5-10%.

- **compartmental model**: The modeling approach that does not track individual agents, but instead considers compartments with rates of transfer between them (e.g., the total number of susceptible people in the compartment S, and their rate of transfer into the exposed compartment E). Compared to agent-based models, compartmental models are typically faster but rely more heavily on assumptions and approximations.

- **contact tracing**: The program for getting in touch with people who have been exposed to contacts of a known positive case.

- **COVID-19**: Technically defined as the disease caused by infection with the virus SARS-CoV-2, but informally used to refer to both the infection and the disease.

- **dynamic rescaling**: A method by which a smaller number of agents is used to represent a larger number of people. For example, 10,000 agents may be used to represent 100,000 people, with an initial 1:1 ratio at the beginning of the simulation, and as the number of infections increases during the simulation, scaling up to a 1:10 ratio.

- **effective reproduction number**: See reproduction number.

- **exposed**: The state of a person who has been infected with SARS-CoV-2, but before they become infectious infectious. One of the four states of the SEIR model. Also known as infected, but the term “exposed” is used to avoid confusion in the acronym SEIR.

- **generation time**: The time between the infection of the primary infection and one or more secondary infections. Compare serial interval.

- **index case**: See primary case.
• **infection fatality ratio** (IFR): The proportion of people who become infected who eventually die. Typical IFRs are 0.3-1.5%, depending on the age distribution and other factors.

• **infectious**: The state of a person who is capable of passing infection on to others. One of the four states in the SEIR model.

• **infectiousness**: See beta.

• **intervention**: Any program or policy implemented to limit the spread of COVID-19: these might including testing or tracing programs, physical distancing, mobility restrictions, vaccination, and other programs. See also non-pharmaceutical intervention.

• **isolation**: The behavior change that occurs after a person has received a positive diagnosis. See also quarantine.

• **layer**: The way in which two people are connected. Also called a contact layer or network layer. Examples include households, schools, workplaces, communities, and LTCFs.

• **long-term care facilities** (LTCFs): Facilities to care for people over long periods of time, also known as assisted living facilities, aged-care facilities, or nursing homes.

• **neutralizing antibodies** (NAbs): The antibodies that offer protection against SARS-CoV-2 infection and COVID-19 disease severity, which may be due to prior infection (natural immunity) or from vaccination.

• **non-pharmaceutical intervention**: Any intervention to reduce the spread or severity of COVID-19 other than therapeutics or vaccination. Examples including distancing, hand washing, mask wearing, etc.

• **parameter**: One or more values (usually numbers) that define how the simulation runs. For example, “probability of developing severe disease” could be considered a parameter (i.e., list of values by age), as could “probability of developing severe disease for people aged 60-70” (i.e., a single number). Parameters can also be qualitative values (e.g., population type) or true/false values (e.g., whether or not to dynamically rescale the population).

• **primary case**: The earliest-infected person diagnosed in a cluster of infections, i.e. the person who is the source of other infections in that cluster. Usually, but not always, this person is also the earliest person in the cluster to be diagnosed. (If back-tracing is used, the primary case may be diagnosed after one or more of the secondary cases.) See also secondary infections.

• **primary infection**: Same as primary case, but not necessarily diagnosed.

• **program**: See intervention.

• **quarantine**: The behavior change that occurs when a person has been notified that they have been in contact with a person who has tested positive. See also isolation.

• **R₀**, **R_e**, **R_eff**: See reproduction number.

• **recovered**: A person who has been infected with SARS-CoV-2 and since recovered. They are usually considered to be immune (reinfection is not considered), and thus removed from the model.

• **removed**: In the context of SEIR modeling, refers to someone who has either recovered from infection or has died; i.e. they are no longer susceptible or infected, so act as if they have been “removed” from the simulation.

• **reproduction number**: The average number of secondary infections caused by each each primary infection. In the absence of interventions, this is called the basic reproduction number, or R₀. Otherwise, it is usually called the effective reproduction number, abbreviated R_e or R_eff. If R_e > 1, then the epidemic is (usually) growing; if R_e < 1, the epidemic is (usually) shrinking.

• **random seed**: The starting point for a given simulation used to convert probabilities (e.g., 10% probability of infection) to events (e.g., a person actually being infected). Two simulations will produce identical results if (and usually only if) they have identical parameters as well as the same random seed.

• **rescaling**: When one agent does not necessarily correspond to one person in the population; for example, 10,000 agents may be used to represent 100,000 people in the population. See also dynamic rescaling.
• **SARS-CoV-2**: The virus responsible for causing COVID-19. Informally, the two terms are used interchangeably.

• **secondary case**: The people who are infected by the *index case* who have been diagnosed.

• **secondary infection**: Same as *secondary case*, but not necessarily diagnosed.

• **SEIR**: A common type of epidemic model, of which Covasim is an example. It stands for Susceptible - Exposed - Infectious - Removed (or “recovered”), referring to the four different states that agents may have. Both agent-based models and compartmental models may have SEIR structure.

• **serial interval**: The time between when the *primary case* develops symptoms and when secondary cases develop infections. Usually used in reference to symptomatic and diagnosed infections (which is a subset of all infections). While the *generation time* is of more interest, the serial interval is easier to measure, so is often used as a proxy for it.

• **simulation**: A single realization of the model, consisting of: the model itself (i.e., Covasim), along with the random seed and other parameters. Running the same simulation is expected to produce the same results every time.

• **susceptible**: The state of a person who has not been infected with SARS-CoV-2, and can become infected. One of the four states of the SEIR model. Also sometimes referred to simply as uninfected.

• **susceptibility**: The probability of a person becoming infected after being exposed to an infectious person. This typically depends on age, as well as whether or not a person has been vaccinated.

• **testing**: The program for diagnosing people with COVID-19. Most typically reverse to polymerase chain reaction (PCR) tests, but can also refer to antigen tests.

• **transmissibility**: See beta.

• **transmission tree**: The network of infections in the model; so named because when plotted, it resembles a tree, with the “trunk” consisting of the initial infections, and the “twigs” consisting of the most recent infections.

• **viral load**: The amount of virus in an infected person’s body. Typically it is assumed that infectiousness is proportional to viral load, although the two are not necessarily linearly proportional.

### 1.8 Code of conduct

#### 1.8.1 Our pledge

We believe that a diverse, equitable, and inclusive environment is essential for producing the best quality software. In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in Covasim development and the Covasim community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

#### 1.8.2 Our standards

Examples of behavior that contributes to creating a positive environment include:

• Using welcoming and inclusive language

• Being respectful of differing viewpoints and experiences

• Gracefully accepting constructive criticism

• Focusing on what is best for the community
• Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

• The use of sexualized language or imagery and unwelcome sexual attention or advances
• Trolling, insulting/derogatory comments, and personal or political attacks
• Public or private harassment
• Publishing others’ private information, such as a physical or electronic address, without explicit permission
• Other conduct which could reasonably be considered inappropriate in a professional setting

1.8.3 Our responsibilities

Covasim maintainers are responsible for clarifying the standards of acceptable behavior and will take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Covasim maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

1.8.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing Covasim or its community. Examples of representing the Covasim project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

1.8.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at covasim@idmod.org. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The Covasim team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Covasim maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of Covasim’s leadership.

1.8.6 Attribution


For answers to common questions about this code of conduct, see the FAQ.

1.9 Contributing to Covasim

Welcome! We are thrilled you are interested in contributing to Covasim. This document will help you get started.

Please see the guidelines below to provide feedback or contribute to Covasim code. Note that we make no representations that the code works as intended or that we will provide support, address issues that are found, or accept pull requests.
1.9.1 Getting started

Contributions to this project are released to the public under the project’s open source license.

Note that this project is released with a Contributor Code of Conduct. By participating in this project you agree to abide by its terms.

Review the documentation before you begin. Additionally, this content is also included in the following README files:

- Project README: Setup and project overview.
- Covasim README: Simulation parameters. application. docker setup.
- Tests README: Running tests locally.

1.9.2 Contribution types

This is a fast-moving project with many opportunities to contribute across the project. We welcome the following types of contributions:

1. Issues:
   - Bug reports.
   - Feature requests.

2. Pull requests:
   - Tests reproducing an issue.
   - Bug fixes.
   - Code to resolve open approved issues.
   - Documentation improvements in the docs folder.

All external communication about these contribution efforts is currently occurring on GitHub.

1.9.3 Request new features or report bugs

If you notice unexpected behavior or a limitation in Covasim, follow the steps below before requesting a new feature or reporting a bug.

1. First, review the Covasim documentation to see if there is already functionality that supports what you want to do. In particular, check Parameters or this README to see if the behavior you observed might be expected and if configuration options are available to provide you with the desired behavior.

2. Search the existing issues to see if there is already one that contains your feedback. If there is, add a thumbs up reaction to convey your interest in the issue being addressed. This helps the team gauge interest without the noise of comments, which trigger notifications to all watchers. Comments should be used only if you have new and useful information to share.

1.9.3.1 Open a feature request

When opening an issue to request a new feature, do the following:

1. Provide a clear and descriptive title for the issue.
2. Include as many details as possible in the body. Fully explain your use case, the problems you’re hitting, and the solutions you’d like to see to address those problems.

### 1.9.3.2 Report a bug

When opening an issue to report a bug, explain the problem and include additional details to help us reproduce the problem:

1. Describe the specific steps that led to the problem you encountered with as many details as possible. Don’t just say what you did, but explain how you did it.

2. Provide specific examples to demonstrate the steps, such as links to files or projects, code snippets, or screen shots. Please use Markdown code blocks for code snippets.

3. Describe the behavior you observed after following the steps and point out exactly what the problem with that behavior is, including explaining what you expected to see instead and why.

### 1.9.4 Submit a pull request

To contribute directly to Covasim code, do the following:

1. Fork and clone the Covasim repository.

2. Install Covasim on your machine. See the project README.

3. Create a new branch:

   ```bash
   git checkout -b my-branch-name
   ```

4. Make your code changes, including a descriptive commit message.

5. Push to your fork and submit a pull request.

Although we make no guarantees that a submitted pull request will be accepted, PRs that meet the following criteria are more likely to be merged:

- Up-to-date with master with no merge conflicts
- Self-contained
- Fix a demonstrable limitation of bug
- Follow the current code style
- If the PR introduces a new feature, it has complete Google style docstrings and comments, and a test demonstrating its functionality
- Otherwise, sample code demonstrating old and new behavior (this can be in the PR comment on GitHub, not necessarily committed in the repo)

If you have additional questions or comments, contact covasim@idmod.org.

### 1.9.5 Resources

- LitCovid
- MIDAS network
- How to contribute to open source
- Using pull requests
• GitHub help

1.10 API reference

Initialize Covasim by importing all the modules

Convention is to use “import covasim as cv”, and then to use all functions and classes directly, e.g. cv.Sim() rather than cv.sim.Sim().

1.10.1 Subpackages

1.10.1.1 covasim.data package

Submodules

covasim.data.country_age_data module

This is the following file:
https://github.com/neherlab/covid19_scenarios/blob/master/src/assets/data/country_age_distribution.json
expressed as a Python file.

covasim.data.household_size_data module

This is the following file:
https://population.un.org/household/exceldata/population_division_UN_Houseshold_Size_and_Composition_2019.xlsx
expressed as a function.

covasim.data.loaders module

Load data

get_country_aliases ()
  Define aliases for countries with odd names in the data

map_entries (json, location)
  Find a match between the JSON file and the provided location(s).

  Parameters
  • json (list or dict) – the data being loaded
  • location (list or str) – the list of locations to pull from

show_locations (location=None, output=False)
  Print a list of available locations.

  Parameters
  • location (str) – if provided, only check if this location is in the list
  • output (bool) – whether to return the list (else print)
Examples:

```python
# Print a list of valid locations
cv.data.show_locations()

# Check if Lithuania is a valid location
cv.data.show_locations('lithuania')

# Check if Viet-Nam is a valid location
cv.data.show_locations('Viet-Nam')
```

**get_age_distribution** *(location=None)*

Load age distribution for a given country or countries.

**Parameters**

- location *(str or list)* – name of the country or countries to load the age distribution for

**Returns**

Numpy array of age distributions, or dict if multiple locations

**Return type**

age_data (array)

**get_household_size** *(location=None)*

Load household size distribution for a given country or countries.

**Parameters**

- location *(str or list)* – name of the country or countries to load the age distribution for

**Returns**

Size of household, or dict if multiple locations

**Return type**

house_size (float)

covasim.data.state_age_data module

This data is translated from the US Census CSV to JSON/Python format.

### 1.10.2 Submodules

#### 1.10.2.1 covasim.analysis module

Additional analysis functions that are not part of the core Covasim workflow, but which are useful for particular investigations.

**class** Analyzer *(label=None)*

**Bases:** sciris.sc_utils.prettyobj

Base class for analyzers. Based on the Intervention class. Analyzers are used to provide more detailed information about a simulation than is available by default – for example, pulling states out of sim.people on a particular timestep before it gets updated in the next timestep.

To retrieve a particular analyzer from a sim, use sim.get_analyzer().

**Parameters**

- label *(str)* – a label for the Analyzer (used for ease of identification)

**initialize** *(sim=None)*

Initialize the analyzer, e.g. convert date strings to integers.

**finalize** *(sim=None)*

Finalize analyzer

This method is run once as part of sim.finalize() enabling the analyzer to perform any final operations after the simulation is complete (e.g. rescaling)

**apply** *(sim)*

Apply analyzer at each time point. The analyzer has full access to the sim object, and typically stores data/results in itself. This is the core method which each analyzer object needs to implement.
Parameters `sim` – the Sim instance

`shrink (in_place=False)`
Remove any excess stored data from the intervention; for use with `sim.shrink()`.

Parameters `in_place` (bool) – whether to shrink the intervention (else shrink a copy)

`to_json()`
Return JSON-compatible representation

Custom classes can’t be directly represented in JSON. This method is a one-way export to produce a JSON-compatible representation of the intervention. This method will attempt to JSONify each attribute of the intervention, skipping any that fail.

Returns JSON-serializable representation

class `snapshot (days, *args, die=True, **kwargs)`
Bases: `covasim.analysis.Analyzer`
Analyzer that takes a “snapshot” of the sim.people array at specified points in time, and saves them to itself. To retrieve them, you can either access the dictionary directly, or use the `get()` method.

Parameters

• `days (list)` – list of ints/strings/date objects, the days on which to take the snapshot
• `args (list)` – additional day(s)
• `die (bool)` – whether or not to raise an exception if a date is not found (default true)
• `kwargs (dict)` – passed to `Analyzer()`

Example:

```python
sim = cv.Sim(analyzers=cv.snapshot('2020-04-04', '2020-04-14'))
sim.run()
snapshot = sim['analyzers'][0]
people = snapshot.snapshots[0]  # Option 1
people = snapshot.snapshots['2020-04-04']  # Option 2
people = snapshot.get('2020-04-14')  # Option 3
people = snapshot.get(34)  # Option 4
people = snapshot.get()  # Option 5
```

`initialize (sim)`

`apply (sim)`

`finalize (sim)`

`get (key=None)`
Retrieve a snapshot from the given key (int, str, or date)

class `age_histogram (days=None, states=None, edges=None, datafile=None, sim=None, die=True, **kwargs)`
Bases: `covasim.analysis.Analyzer`
Calculate statistics across age bins, including histogram plotting functionality.

Parameters

• `days (list)` – list of ints/strings/date objects, the days on which to calculate the histograms (default: last day)
• `states (list)` – which states of people to record (default: exposed, tested, diagnosed, dead)
• **edges** (*list*) – edges of age bins to use (default: 10 year bins from 0 to 100)
• **datafile** (*str*) – the name of the data file to load in for comparison, or a dataframe of data (optional)
• **sim** (*Sim*) – only used if the analyzer is being used after a sim has already been run
• **die** (*bool*) – whether to raise an exception if dates are not found (default true)
• **kwargs** (*dict*) – passed to Analyzer()

**Examples:**

```python
sim = cv.Sim(analyzers=cv.age_histogram())
sim.run()

agehist = sim.get_analyzer()
agehist = cv.age_histogram(sim=sim) # Alternate method
agehist.plot()

from_sim(sim)
  Create an age histogram from an already run sim
initialize(sim)
apply(sim)
finalize(sim)
get(key=None)
  Retrieve a specific histogram from the given key (int, str, or date)
compute_windows()
  Convert cumulative histograms to windows
plot(windows=False, width=0.8, color='#F8A493', fig_args=None, axis_args=None, data_args=None)
  Simple method for plotting the histograms.
```

---

**class daily_age_stats** (*states=None, edges=None, **kwargs*)

Bases: *covasim.analysis.Analyzer*

Calculate daily counts by age, saving for each day of the simulation. Can plot either time series by age or a histogram over all time.

**Parameters**

• **states** (*list*) – which states of people to record (default: ['diagnoses', 'deaths', 'tests', 'severe'])
• **edges** (*list*) – edges of age bins to use (default: 10 year bins from 0 to 100)
• **kwargs** (*dict*) – passed to Analyzer()
Examples:

```python
sim = cv.Sim(analyzers=cv.daily_age_stats())
sim = cv.Sim(pars, analyzers=daily_age)
sim.run()
daily_age = sim.get_analyzer()
daily_age.plot()
daily_age.plot(total=True)
```

```python
initialize(sim)
apply(sim)
to_df()
    Create dataframe totals for each day
to_total_df()
    Create dataframe totals across days
plot(total=False, do_show=None, fig_args=None, axis_args=None, plot_args=None,
     dateformat='%b-%d', width=0.8, color='#F8A493', data_args=None)
Plot the results.
```

**Parameters**

- `total` (bool) – whether to plot the total histograms rather than time series
- `do_show` (bool) – whether to show the plot
- `fig_args` (dict) – passed to plt.figure()
- `axis_args` (dict) – passed to plt.subplots_adjust()
- `plot_args` (dict) – passed to plt.plot()
- `dateformat` (str) – the format to use for the x-axes (only used for time series)
- `width` (float) – width of bars (only used for histograms)
- `color` (hex/rgb) – the color of the bars (only used for histograms)

```python
class daily_stats(days=None, verbose=True, reporter=None, save_inds=False, **kwargs)
Bases: covasim.analysis.Analyzer
```

Print out daily statistics about the simulation. Note that this analyzer takes a considerable amount of time, so should be used primarily for debugging, not in production code. To keep the intervention but toggle it off, pass an empty list of days.

To show the stats for a day after a run has finished, use e.g. `daily_stats.report('2020-04-04')`.

**Parameters**

- `days` (list) – days on which to print out statistics (if None, assume all)
- `verbose` (bool) – whether to print on each timestep
- `reporter` (func) – if supplied, a custom parser of the stats object into a report (see make_report() function for syntax)
- `save_inds` (bool) – whether to save the indices of every infection at every timestep (also recoverable from the infection log)

**Example:**
```python
sim = cv.Sim(analyzers=cv.daily_stats())
sim.run()
sim['analyzers'][0].plot()
```

**initialize** *(sim)*

**intersect** (*args*)
Compute the intersection between arrays of indices, handling either keys to precomputed indices or lists of indices. With two array inputs, simply performs np.intersect1d(arr1, arr2).

**apply** *(sim)*

**report** *(day=None)*
Print out one or all reports – take a date string or an int

**make_report** *(sim, stats, show_empty='count')*
Turn the statistics into a report

**transpose** *(keys=None)*
Transpose the data from a list-of-dicts-of-dicts to a dict-of-dicts-of-lists

**plot** *(fig_args=None, axis_args=None, plot_args=None, do_show=None)*
Plot the daily statistics recorded. Some overlap with e.g. `sim.plot(to_plot='overview')`.

**Parameters**

- **fig_args** *(dict)* – passed to `plt.figure()`
- **axis_args** *(dict)* – passed to `plt.subplots_adjust()`
- **plot_args** *(dict)* – passed to `plt.plot()`
- **do_show** *(bool)* – whether to show the plot

### nab_histogram

#### Class `nab_histogram`

**Bases:** `covasim.analysis.Analyzer`

Store histogram of log_{10}(NAb) distribution

**Parameters**

- **days** *(list)* – days on which calculate the NAb histogram (if None, assume last day)
- **edges** *(list)* – log10 bin edges for histogram

**Example:**

```python
sim = cv.Sim(analyzers=cv.nab_histogram())
sim.run()
sim.get_analyzer().plot()
```

New in version 3.1.0.

**initialize** *(sim)*

**apply** *(sim)*

**plot** *(fig_args=None, axis_args=None, plot_args=None, do_show=None)*
Plot the results

### Fit

#### Class `Fit`

**Bases:** `covasim.analysis.Analyzer`

A class for calculating the fit between the model and the data. Note the following terminology is used here:
• fit: nonspecific term for how well the model matches the data
• difference: the absolute numerical differences between the model and the data (one time series per result)
• goodness-of-fit: the result of passing the difference through a statistical function, such as mean squared error
• loss: the goodness-of-fit for each result multiplied by user-specified weights (one time series per result)
• mismatches: the sum of all the losses (a single scalar value per time series)
• mismatch: the sum of the mismatches – this is the value to be minimized during calibration

Parameters

- sim (Sim) – the sim object
- weights (dict) – the relative weight to place on each result (by default: 10 for deaths, 5 for diagnoses, 1 for everything else)
- keys (list) – the keys to use in the calculation
- custom (dict) – a custom dictionary of additional data to fit; format is e.g. {'my_output': {'data':[1,2,3], 'sim':[1,2,4], 'weights':2.0}}
- compute (bool) – whether to compute the mismatch immediately
- verbose (bool) – detail to print
- die (bool) – whether to raise an exception if no data are supplied
- label (str) – the label for the analyzer
- kwargs (dict) – passed to cv.compute_gof() – see this function for more detail on goodness-of-fit calculation options

Example:

```python
sim = cv.Sim(datafile='my-data-file.csv')
sim.run()
fit = sim.compute_fit()
fit.plot()
```

`compute()`
Perform all required computations

`reconcile_inputs()`
Find matching keys and indices between the model and the data

`compute_diffs (absolute=False)`
Find the differences between the sim and the data

`compute_gofs (**kwargs)`
Compute the goodness-of-fit

`compute_losses()`
Compute the weighted goodness-of-fit

`compute_mismatch (use_median=False)`
Compute the final mismatch

`plot (keys=None, width=0.8, fig_args=None, axis_args=None, plot_args=None, date_args=None, do_show=None, fig=None)`
Plot the fit of the model to the data. For each result, plot the data and the model; the difference; and the loss (weighted difference). Also plots the loss as a function of time.
Parameters

- **keys** *(list)* – which keys to plot (default, all)
- **width** *(float)* – bar width
- **fig_args** *(dict)* – passed to `pl.figure()`
- **axis_args** *(dict)* – passed to `pl.subplots_adjust()`
- **plot_args** *(dict)* – passed to `pl.plot()`
- **date_args** *(dict)* – passed to `cv.plotting.reset_ticks()` (handle date format, rotation, etc.)
- **do_show** *(bool)* – whether to show the plot
- **fig** *(fig)* – if supplied, use this figure to plot in

Returns

Figure object

class Calibration(sim, calib_pars=None, fit_args=None, custom_fn=None, par_samplers=None, n_trials=None, n_workers=None, total_trials=None, name=None, db_name=None, keep_db=None, storage=None, label=None, die=False, verbose=True)

Bases: covasim.analysis.Analyzer

A class to handle calibration of Covasim simulations. Uses the Optuna hyperparameter optimization library (optuna.org), which must be installed separately (via pip install optuna).

Note: running a calibration does not guarantee a good fit! You must ensure that you run for a sufficient number of iterations, have enough free parameters, and that the parameters have wide enough bounds. Please see the tutorial on calibration for more information.

Parameters

- **sim** *(Sim)* – the simulation to calibrate
- **calib_pars** *(dict)* – a dictionary of the parameters to calibrate of the format `dict(key1=[best, low, high])`
- **fit_args** *(dict)* – a dictionary of options that are passed to `sim.compute_fit()` to calculate the goodness-of-fit
- **par_samplers** *(dict)* – an optional mapping from parameters to the Optuna sampler to use for choosing new points for each; by default, `suggest_uniform`
- **custom_fn** *(func)* – a custom function for modifying the simulation; receives the sim and `calib_pars` as inputs, should return the modified sim
- **n_trials** *(int)* – the number of trials per worker
- **n_workers** *(int)* – the number of parallel workers (default: maximum)
- **total_trials** *(int)* – if `n_trials` is not supplied, calculate by dividing this number by `n_workers`
- **name** *(str)* – the name of the database (default: ‘covasim_calibration’)
- **db_name** *(str)* – the name of the database file (default: ‘covasim_calibration.db’)
- **keep_db** *(bool)* – whether to keep the database after calibration (default: false)
- **storage** *(str)* – the location of the database (default: sqlite)
- **label** *(str)* – a label for this calibration object
- **die** *(bool)* – whether to stop if an exception is encountered (default: false)
• **verbose** *(bool)* – whether to print details of the calibration
• **kwargs** *(dict)* – passed to cv.Calibration()

Returns  A Calibration object

**Example:**

```python
sim = cv.Sim(datafile='data.csv')
calib_pars = dict(beta=[0.015, 0.010, 0.020])
calib = cv.Calibration(sim, calib_pars, total_trials=100)
calib.calibrate()
calib.plot()
```

New in version 3.0.3.

**run_sim**(calib_pars, label=None, return_sim=False)

Create and run a simulation

**run_trial**(trial)

Define the objective for Optuna

**worker()**

Run a single worker

**run_workers()**

Run multiple workers in parallel

**remove_db()**

Remove the database file if keep_db is false and the path exists.

New in version 3.1.0.

**make_study()**

Make a study, deleting one if it already exists

**calibrate**(calib_pars=None, verbose=True, **kwargs)

Actually perform calibration.

**Parameters**

• **calib_pars** *(dict)* – if supplied, overwrite stored calib_pars
• **verbose** *(bool)* – whether to print output from each trial
• **kwargs** *(dict)* – if supplied, overwrite stored run_args (n_trials, n_workers, etc.)

**summarize()**

Print out results from the calibration

**parse_study()**

Parse the study into a data frame – called automatically

**to_json**(filename=None)

Convert the data to JSON.

New in version 3.1.1.

**plot_sims**(**kwargs)

Plot sims, before and after calibration.

New in version 3.1.1: renamed from plot() to plot_sims().

**plot_trend**(best_thresh=2)

Plot the trend in best mismatch over time.
New in version 3.1.1.

```python
plot_all()
```
Plot every point in the calibration. Warning, very slow for more than a few hundred trials.

New in version 3.1.1.

```python
plot_best(best_thresh=2)
```
Plot only the points with lowest mismatch. New in version 3.1.1.

```python
plot_stride(npts=200)
```
Plot a fixed number of points in order across the results.

New in version 3.1.1.

```python
class TransTree(sim, to_networkx=False, **kwargs)
```
A class for holding a transmission tree. There are several different representations of the transmission tree: “infection_log” is copied from the people object and is the simplest representation. “detailed h” includes additional attributes about the source and target. If NetworkX is installed (required for most methods), “graph” includes an NX representation of the transmission tree.

**Parameters**

- `sim` (Sim) – the sim object
- `to_networkx` (bool) – whether to convert the graph to a NetworkX object

**Example:**

```python
sim = cv.Sim().run()
sim.run()
tt = sim.make_transtree()
tt.plot()
tt.plot_histograms()
```

New in version 2.1.0: `tt.detailed` is a dataframe rather than a list of dictionaries; for the latter, use `tt.detailed.to_dict('records')`.

```python
def day(day=None, which=None)
```
Convenience function for converting an input to an integer day

```python
def count_targets(start_day=None, end_day=None)
```
Count the number of targets each infected person has. If start and/or end days are given, it will only count the targets of people who got infected between those dates (it does not, however, filter on the date the target got infected).

**Parameters**

- `start_day` (int/str) – the day on which to start counting people who got infected
- `end_day` (int/str) – the day on which to stop counting people who got infected

```python
def count_transmissions()
```
Iterable over edges corresponding to transmission events

This excludes edges corresponding to seeded infections without a source

```python
def make_detailed(people, reset=False)
```
Construct a detailed transmission tree, with additional information for each person

```python
def r0(recovered_only=False)
```
Return average number of transmissions per person
This doesn’t include seed transmissions. By default, it also doesn’t adjust for length of infection (e.g. people infected towards the end of the simulation will have fewer transmissions because their infection may extend past the end of the simulation, these people are not included). If ‘recovered_only=True’ then the downstream transmissions will only be included for people that recover before the end of the simulation, thus ensuring they all had the same amount of time to transmit.

**plot** *(fig_args=None, plot_args=None, do_show=None, fig=None)*
Plot the transmission tree.

**Parameters**
- `fig_args` *(dict)* – passed to `pl.figure()`
- `plot_args` *(dict)* – passed to `pl.plot()`
- `do_show` *(bool)* – whether to show the plot
- `fig` *(fig)* – if supplied, use this figure

**animate** *(*args, **kwargs)*
Animate the transmission tree.

**Parameters**
- `animate` *(bool)* – whether to animate the plot (otherwise, show when finished)
- `verbose` *(bool)* – print out progress of each frame
- `markersize` *(int)* – size of the markers
- `sus_color` *(list)* – color for susceptibles
- `fig_args` *(dict)* – arguments passed to `pl.figure()`
- `axis_args` *(dict)* – arguments passed to `pl.subplots_adjust()`
- `plot_args` *(dict)* – arguments passed to `pl.plot()`
- `delay` *(float)* – delay between frames in seconds
- `colors` *(list)* – color of each person
- `cmap` *(str)* – colormap for each person (if colors is not supplied)
- `fig` *(fig)* – if supplied, use this figure

**Returns** the figure object

**Return type** fig

**plot_histograms** *(start_day=None, end_day=None, bins=None, width=0.8, fig_args=None, fig=None)*
Plots a histogram of the number of transmissions.

**Parameters**
- `start_day` *(int/str)* – the day on which to start counting people who got infected
- `end_day` *(int/str)* – the day on which to stop counting people who got infected
- `bins` *(list)* – bin edges to use for the histogram
- `width` *(float)* – width of bars
- `fig_args` *(dict)* – passed to `pl.figure()`
- `fig` *(fig)* – if supplied, use this figure
1.10.2.2 covasim.base module

Base classes for Covasim. These classes handle a lot of the boilerplate of the People and Sim classes (e.g. loading, saving, key lookups, etc.), so those classes can be focused on the disease-specific functionality.

```python
class ParsObj(pars)
    Bases: covasim.base.FlexPretty

    A class based around performing operations on a self.pars dict.

    update_pars(pars=None, create=False)

    Update internal dict with new pars.

    Parameters
    • pars (dict) – the parameters to update (if None, do nothing)
    • create (bool) – if create is False, then raise a KeyNotFoundError if the key does not already exist
```

```python
class Result(name=None, npts=None, scale=True, color=None, n_variants=0)
    Bases: object

    Stores a single result – by default, acts like an array.

    Parameters
    • name (str) – name of this result, e.g. new_infections
    • npts (int) – if values is None, precreate it to be of this length
    • scale (bool) – whether or not the value scales by population scale factor
    • color (str/arr) – default color for plotting (hex or RGB notation)
    • n_variants (int) – the number of variants the result is for (0 for results not by variant)

    Example:

    ```python
    import covasim as cv
    r1 = cv.Result(name='test1', npts=10)
    r1[:5] = 20
    print(r1.values)
    ```
```

```python
class BaseSim(*args, **kwargs)
    Bases: covasim.base.ParsObj

    The BaseSim class stores various methods useful for the Sim that are not directly related to simulating the epidemic. It is not used outside of the Sim object, so the separation of methods into the BaseSim and Sim classes is purely to keep each one of manageable size.

    update_pars(pars=None, create=False, **kwargs)

    Ensure that metaparameters get used properly before being updated

    set_metadata(simfile)

    Set the metadata for the simulation – creation time and filename

    set_seed(seed=-1)

    Set the seed for the random number stream from the stored or supplied value

    Parameters seed (None or int) – if no argument, use current seed; if None, randomize; otherwise, use and store supplied seed
```
Count the number of people – if it fails, assume none

Get the total population size, i.e. the number of agents times the scale factor – if it fails, assume none

Count the number of time points

Create a time vector

Create a vector of dates

Returns Array of datetime instances containing the date associated with each simulation time step

Convert a string, date/datetime object, or int to a day (int).

Parameters day (str, date, int, or list) – convert any of these objects to a day relative to the simulation’s start day

Returns the day(s) in simulation time

Return type days (int or str)

Example:
sim.day('2020-04-05') # Returns 35

Convert one or more integer days of simulation time to a date/list of dates – by default returns a string, or returns a datetime Date object if as_date is True. See also cv.date(), which provides a partly overlapping set of date conversion features.

Parameters

• ind (int, list, or array) – the index day(s) in simulation time (NB: strings and date objects are accepted, and will be passed unchanged)

• args (list) – additional day(s)

• dateformat (str) – the format to return the date in

• as_date (bool) – whether to return as a datetime date instead of a string

Returns the date(s) corresponding to the simulation day(s)

Return type dates (str, Date, or list)

Examples:
sim = cv.Sim()
sim.date(34) # Returns '2020-04-04'
sim.date([34, 54]) # Returns ['2020-04-04', '2020-04-24']
sim.date([34, '2020-04-24']) # Returns ['2020-04-04', '2020-04-24']
sim.date(34, 54, as_date=True) # Returns [datetime.date(2020, 4, 4), datetime.date(2020, 4, 24)]

Get the actual results objects, not other things stored in sim.results.
If which is ‘main’, return only the main results keys. If ‘variant’, return only variant keys. If ‘all’, return all keys.

**copy()**

Returns a deep copy of the sim

**export_results**(for_json=True, filename=None, indent=2, *args, **kwargs)

Convert results to dict – see also to_json().

The results written to Excel must have a regular table shape, whereas for the JSON output, arbitrary data shapes are supported.

**Parameters**

- for_json (bool) – if False, only data associated with Result objects will be included in the converted output
- filename (str) – filename to save to; if None, do not save
- indent (int) – indent (int): if writing to file, how many indents to use per nested level
- args (list) – passed to savejson()
- kwargs (dict) – passed to savejson()

**Returns** dictionary representation of the results

**Return type** resdict (dict)

**export_pars**(filename=None, indent=2, *args, **kwargs)

Return parameters for JSON export – see also to_json().

This method is required so that interventions can specify their JSON-friendly representation.

**Parameters**

- filename (str) – filename to save to; if None, do not save
- indent (int) – indent (int): if writing to file, how many indents to use per nested level
- args (list) – passed to savejson()
- kwargs (dict) – passed to savejson()

**Returns** a dictionary containing all the parameter values

**Return type** pardict (dict)

**to_json**(filename=None, keys=None, tostring=False, indent=2, verbose=False, *args, **kwargs)

Export results and parameters as JSON.

**Parameters**

- filename (str) – if None, return string; else, write to file
- keys (str or list) – attributes to write to json (default: results, parameters, and summary)
- tostring (bool) – if not writing to file, whether to write to string (alternative is sanitized dictionary)
- indent (int) – if writing to file, how many indents to use per nested level
- verbose (bool) – detail to print
- args (list) – passed to savejson()
- kwargs (dict) – passed to savejson()
**Returns** A unicode string containing a JSON representation of the results, or writes the JSON file to disk

**Examples:**
```python
json = sim.to_json()
sim.to_json('results.json')
sim.to_json('summary.json', keys='summary')
```

**to_df**(date_index=False)
Export results to a pandas dataframe

**Parameters**
- **date_index**(bool) – if True, use the date as the index

**to_excel**(filename=None, skip_pars=None)
Export parameters and results as Excel format

**Parameters**
- **filename**(str) – if None, return string; else, write to file
- **skip_pars**(list) – if provided, a custom list parameters to exclude

**Returns** An sc.Spreadsheet with an Excel file, or writes the file to disk

**shrink**(skip_attrs=None, in_place=True)
“Shrinks” the simulation by removing the people and other memory-intensive attributes (e.g., some interventions and analyzers), and returns a copy of the “shrunk” simulation. Used to reduce the memory required for RAM or for saved files.

**Parameters**
- **skip_attrs**(list) – a list of attributes to skip (remove) in order to perform the shrinking; default “people”
- **in_palce**(bool) – whether to perform the shrinking in place (default), or return a shrunk copy instead

**Returns** a Sim object with the listed attributes removed

**Return type** shrunk (Sim)

**save**(filename=None, keep_people=None, skip_attrs=None, **kwargs)
Save to disk as a gzipped pickle.

**Parameters**
- **filename**(str or None) – the name or path of the file to save to; if None, uses stored
- **kwargs** – passed to sc.makefilepath()

**Returns** the validated absolute path to the saved file

**Return type** filename (str)

**Example:**
```python
sim.save()  # Saves to a .sim file with the date and time of creation by default
```

**static load**(filename, *args, **kwargs)
Load from disk from a gzipped pickle.

**Parameters**
• **filename** (*str*) – the name or path of the file to load from

• **kwargs** – passed to `cv.load()`

Returns the loaded simulation object

Return type `sim (Sim)`

Example:

```python
sim = cv.Sim.load('my-simulation.sim')
```

### get_interventions(*label=None, partial=False, as_inds=False*)

Find the matching intervention(s) by label, index, or type. If None, return all interventions. If the label provided is "summary", then print a summary of the interventions (index, label, type).

Parameters

- **label** (*str, int, Intervention, list*) – the label, index, or type of intervention to get; if a list, iterate over one of those types
- **partial** (*bool*) – if true, return partial matches (e.g. ‘beta’ will match all beta interventions)
- **as_inds** (*bool*) – if true, return matching indices instead of the actual interventions

Examples:

```python
tp = cv.test_prob(symp_prob=0.1)
cb1 = cv.change_beta(days=5, changes=0.3, label='NPI')
cb2 = cv.change_beta(days=10, changes=0.3, label='Masks')
sim = cv.Sim(interventions=[tp, cb1, cb2])
cb1, cb2 = sim.get_interventions(cv.change_beta)
tp, cb2 = sim.get_interventions([0,2])
ind = sim.get_interventions(cv.change_beta, as_inds=True)  # Returns [1,2]
sim.get_interventions('summary')  # Prints a summary
```

### get_intervention(*label=None, partial=False, first=False, die=True*)

Like `get_interventions()`, find the matching intervention(s) by label, index, or type. If more than one intervention matches, return the last by default. If no label is provided, return the last intervention in the list.

Parameters

- **label** (*str, int, Intervention, list*) – the label, index, or type of intervention to get; if a list, iterate over one of those types
- **partial** (*bool*) – if true, return partial matches (e.g. ‘beta’ will match all beta interventions)
- **first** (*bool*) – if true, return first matching intervention (otherwise, return last)
- **die** (*bool*) – whether to raise an exception if no intervention is found

Examples:

```python
tp = cv.test_prob(symp_prob=0.1)
cb = cv.change_beta(days=5, changes=0.3, label='NPI')
sim = cv.Sim(interventions=[tp, cb])
cb = sim.get_intervention('NPI')
cb = sim.get_intervention('NP', partial=True)  
```

(continues on next page)
cb = sim.get_intervention(1)
cb = sim.get_intervention()
tp = sim.get_intervention(first=True)

get_analyzers (label=None, partial=False, as inds=False)
Same as get_interventions(), but for analyzers.

get_analyzer (label=None, partial=False, first=False, die=True)
Same as get_intervention(), but for analyzers.

class BasePeople

Bases: covasim.base.FlexPretty

A class to handle all the boilerplate for people – note that as with the BaseSim vs Sim classes, everything interesting happens in the People class, whereas this class exists to handle the less interesting implementation details.

lock ()
Lock the people object to prevent keys from being added

unlock ()
Unlock the people object to allow keys to be added

summarize (output=False)
Print a summary of the people – same as brief

set (key, value, die=True)
Ensure sizes and dtypes match

gt (key)
Convenience method – key can be string or list of strings

ttrue (key)
Return indices matching the condition

false (key)
Return indices not matching the condition

defined (key)
Return indices of people who are not-nan

undefined (key)
Return indices of people who are nan

count (key)
Count the number of people for a given key

count_by_variant (key, variant)
Count the number of people for a given key

count_not (key)
Count the number of people who do not have a property for a given key

set_pars (pars=None)
Re-link the parameters stored in the people object to the sim containing it, and perform some basic validation.

keys ()
Returns keys for all properties of the people object

person_keys ()
Returns keys specific to a person (e.g., their age)
**state_keys()**
Returns keys for different states of a person (e.g., symptomatic)

**date_keys()**
Returns keys for different event dates (e.g., date a person became symptomatic)

**dur_keys()**
Returns keys for different durations (e.g., the duration from exposed to infectious)

**layer_keys()**
Get the available contact keys – try contacts first, then beta_layer

**indices()**
The indices of each people array

**validate** (*die=True, verbose=False*)

**to_df()**
Convert to a Pandas dataframe

**to_arr()**
Return as numpy array

**person** (*ind*)
Method to create person from the people

**to_people()**
Return all people as a list

**from_people** (*people, resize=True*)
Convert a list of people back into a People object

**to_graph()**
Convert all people to a networkx MultiDiGraph, including all properties of the people (nodes) and contacts (edges).

**Example:**

```python
import covasim as cv
import networkx as nx
sim = cv.Sim(pop_size=50, pop_type='hybrid', contacts=dict(h=3, s=10, w=10, c=5)).run()
G = sim.people.to_graph()
nodes = G.nodes(data=True)
edges = G.edges(keys=True)
node_colors = [n['age'] for i,n in nodes]
layer_map = dict(h='#37b', s='#e11', w='#4a4', c='#a49')
edge_colors = [layer_map[G[i][j][k]['layer']] for i,j,k in edges]
edge_weights = [G[i][j][k]['beta']*5 for i,j,k in edges]
xn.draw(G, node_color=node_colors, edge_color=edge_colors, width=edge_weights, alpha=0.5)
```

**init_contacts** (*reset=False*)
Initialize the contacts dataframe with the correct columns and data types

**add_contacts** (*contacts, lkey=None, beta=None*)
Add new contacts to the array. See also contacts.add_layer().

**make_edgelist** (*contacts*)
Parse a list of people with a list of contacts per person and turn it into an edge list.

**static remove_duplicates** (*df*)
Sort the dataframe and remove duplicates – note, not extensively tested
**class Person** *(pars=None, uid=None, age=-1, sex=-1, contacts=None)*

Bases: `sciris.sc_utils.prettyobj`

Class for a single person. Note: this is largely deprecated since `sim.people` is now based on arrays rather than being a list of people.

**class FlexDict**

Bases: `dict`

A dict that allows more flexible element access: in addition to `obj['a']`, also allow `obj[0]`. Lightweight implementation of the Sciris odict class.

  - `keys()`
  - `values()`
  - `items()`

**class Contacts** *(layer_keys=None, data=None)*

Bases: `covasim.base.FlexDict`

A simple (for now) class for storing different contact layers.

  - `add_layer(**kwargs)`

Small method to add one or more layers to the contacts. Layers should be provided as keyword arguments.

Example:

```
hospitals_layer = cv.Layer(label='hosp')
sim.people.contacts.add_layer(hospitals=hospitals_layer)
```

  - `pop_layer(*args)`

Remove the layer(s) from the contacts.

Example:

```
sim.people.contacts.pop_layer('hospitals')
```

Note: while included here for convenience, this operation is equivalent to simply popping the key from the contacts dictionary.

  - `to_graph()`

Convert all layers to a networkx MultiDiGraph

Example:

```
import networkx as nx
sim = cv.Sim(pop_size=50, pop_type='hybrid').run()
G = sim.people.contacts.to_graph()
xn.draw(G)
```

**class Layer** *(label=None, **kwargs)*

Bases: `covasim.base.FlexDict`

A small class holding a single layer of contact edges (connections) between people.

The input is typically three arrays: person 1 of the connection, person 2 of the connection, and the weight of the connection. Connections are undirected; each person is both a source and sink.

This class is usually not invoked directly by the user, but instead is called as part of the population creation.

**Parameters**

- `p1(array)` – an array of N connections, representing people on one side of the connection
• *p2* (*array*) – an array of people on the other side of the connection

• *beta* (*array*) – an array of weights for each connection

• *label* (*str*) – the name of the layer (optional)

• *kwargs* (*dict*) – other keys copied directly into the layer

Note that all arguments (except for *label*) must be arrays of the same length, although not all have to be supplied at the time of creation (they must all be the same at the time of initialization, though, or else validation will fail).

**Examples:**

```python
# Generate an average of 10 contacts for 1000 people
n = 10_000
n_people = 1000
p1 = np.random.randint(n_people, size=n)
p2 = np.random.randint(n_people, size=n)
beta = np.ones(n)
layer = cv.Layer(p1=p1, p2=p2, beta=beta, label='rand')

# Convert one layer to another with extra columns
index = np.arange(n)
self_conn = p1 == p2
layer2 = cv.Layer(**layer, index=index, self_conn=self_conn, label=layer.label)
```

**members**
- Return sorted array of all members

**meta_keys()**
- Return the keys for the layer’s meta information – i.e., *p1*, *p2*, *beta*

**validate()**
- Check the integrity of the layer: right types, right lengths

**pop_inds(inds)**
- “Pop” the specified indices from the edgelist and return them as a dict. Returns in the right format to be used with layer.append().

  **Parameters** *inds* (*int*, *array*, *slice*) – the indices to be removed

**append(contacts)**
- Append contacts to the current layer.

  **Parameters** *contacts* (*dict*) – a dictionary of arrays with keys *p1*, *p2*, *beta*, as returned from layer.pop_inds()

**to_df()**
- Convert to dataframe

**from_df(df, keys=None)**
- Convert from a dataframe

**to_graph()**
- Convert to a networkx DiGraph

**Example:**

```python
import networkx as nx
sim = cv.Sim(pop_size=20, pop_type='hybrid').run()
G = sim.people.contacts['h'].to_graph()
xz.draw(G)
```
**find_contacts** *(inds, as_array=True)*

Find all contacts of the specified people

For some purposes (e.g. contact tracing) it's necessary to find all of the contacts associated with a subset of the people in this layer. Since contacts are bidirectional it's necessary to check both P1 and P2 for the target indices. The return type is a Set so that there is no duplication of indices (otherwise if the Layer has explicit symmetric interactions, they could appear multiple times). This is also for performance so that the calling code doesn't need to perform its own unique() operation. Note that this cannot be used for cases where multiple connections count differently than a single infection, e.g. exposure risk.

**Parameters**

- **inds** *(array)* – indices of people whose contacts to return
- **as_array** *(bool)* – if true, return as sorted array (otherwise, return as unsorted set)

**Returns** a set of indices for pairing partners

**Return type** `contact_inds` *(array)*

Example: If there were a layer with - P1 = [1,2,3,4] - P2 = [2,3,1,4] Then `find_contacts([1,3])` would return {1,2,3}

**update** *(people, frac=1.0)*

Regenerate contacts on each timestep.

This method gets called if the layer appears in `sim.pars['dynam_layer']`. The Layer implements the update procedure so that derived classes can customize the update e.g. implementing over-dispersion/other distributions, random clusters, etc.

Typically, this method also takes in the `people` object so that the update can depend on person attributes that may change over time (e.g. changing contacts for people that are severe/critical).

**Parameters**

- **people** *(People)* – the Covasim People object, which is usually used to make new contacts
- **frac** *(float)* – the fraction of contacts to update on each timestep

### 1.10.2.3 covasim.defaults module

Set the defaults across each of the different files.

To change the default precision from 32 bit (default) to 64 bit, use:

```python
cv.options.set(precision=64)
```

**default_float**

alias of `numpy.float32`

**default_int**

alias of `numpy.int32`

**get_default_colors()**

Specify plot colors – used in sim.py.

NB, includes duplicates since stocks and flows are named differently.

**get_default_plots**(which='default', kind='sim', sim=None)

Specify which quantities to plot; used in sim.py.

**Parameters**

- **which** *(str)* – either ‘default’ or ‘overview’
### 1.10.2.4 covasim.immunity module

Defines classes and methods for calculating immunity

**class variant** *(variant, days, label=None, n_imports=1, rescale=True)*

Bases: `sciris.sc_utils.prettyobj`

Add a new variant to the sim

**Parameters**

- **variant**(str/dict) – name of variant, or dictionary of parameters specifying information about the variant
- **days**(int/list) – day(s) on which new variant is introduced
- **label**(str) – if variant is supplied as a dict, the name of the variant
- **n_imports**(int) – the number of imports of the variant to be added
- **rescale**(bool) – whether the number of imports should be rescaled with the population

**Example:**

```python
alpha = cv.variant('alpha', days=10) # Make the alpha variant B117 active from day 10
p1 = cv.variant('p1', days=15) # Make variant P1 active from day 15
my_var = cv.variant(variant={'rel_beta': 2.5}, label='My variant', days=20)
sim = cv.Sim(variants=[alpha, p1, my_var]).run() # Add them all to the sim
sim2 = cv.Sim(variants=cv.variant('alpha', days=0, n_imports=20), pop_infected=0).run() # Replace default variant with alpha
```

**parse**(variant=None, label=None)

Unpack variant information, which may be given as either a string or a dict

**initialize**(sim)

Update variant info in sim

**apply**(sim)

Introduce new infections with this variant

### 1.10.2.5 covasim.interventions module

Specify the core interventions available in Covasim. Other interventions can be defined by the user by inheriting from these classes.

**InterventionDict**(which, pars)

Generate an intervention from a dictionary. Although a function, it acts like a class, since it returns a class instance.

**Example:**

```python
interv = cv.InterventionDict(which='change_beta', pars={'days': 30, 'changes': 0.05, 'layers': None})
```

**class Intervention**(label=None, show_label=False, do_plot=None, line_args=None)

Bases: object

Base class for interventions. By default, interventions are printed using a dict format, which they can be recreated from. To display all the attributes of the intervention, use disp() instead.

To retrieve a particular intervention from a sim, use sim.get_intervention().
Parameters

- **label** *(str)* – a label for the intervention (used for plotting, and for ease of identification)
- **show_label** *(bool)* – whether or not to include the label in the legend
- **do_plot** *(bool)* – whether or not to plot the intervention
- **line_args** *(dict)* – arguments passed to `pl.axvline()` when plotting

**disp()**
Print a detailed representation of the intervention

**initialize**(sim=None)
Initialize intervention – this is used to make modifications to the intervention that can’t be done until after the sim is created.

**finalize**(sim=None)
Finalize intervention
This method is run once as part of `sim.finalize()` enabling the intervention to perform any final operations after the simulation is complete (e.g. rescaling)

**apply**(sim)
Apply the intervention. This is the core method which each derived intervention class must implement. This method gets called at each timestep and can make arbitrary changes to the Sim object, as well as storing or modifying the state of the intervention.

**Parameters**

- **sim** – the Sim instance

**Returns**
None

**shrink**(in_place=False)
Remove any excess stored data from the intervention; for use with `sim.shrink()`.

**Parameters**

- **in_place** *(bool)* – whether to shrink the intervention (else shrink a copy)

**plot_intervention**(sim, ax=None, **kwargs)
Plot the intervention
This can be used to do things like add vertical lines on days when interventions take place. Can be disabled by setting `self.do_plot=False`.

Note 1: you can modify the plotting style via the `line_args` argument when creating the intervention.

Note 2: By default, the intervention is plotted at the days stored in `self.days`. However, if there is a `self.plot_days` attribute, this will be used instead.

**Parameters**

- **sim** – the Sim instance
- **ax** – the axis instance
- **kwargs** – passed to `ax.axvline()`

**Returns**
None

**to_json()**
Return JSON-compatible representation
Custom classes can’t be directly represented in JSON. This method is a one-way export to produce a JSON-compatible representation of the intervention. In the first instance, the object dict will be returned. However, if an intervention itself contains non-standard variables as attributes, then its `to_json` method will need to handle those.
Note that simply printing an intervention will usually return a representation that can be used to recreate it.

**Returns** JSON-serializable representation (typically a dict, but could be anything else)

class dynamic_pars(pars=None, **kwargs)

    Bases: covasim.interventions.Intervention

A generic intervention that modifies a set of parameters at specified points in time.

The intervention takes a single argument, pars, which is a dictionary of which parameters to change, with following structure: keys are the parameters to change, then subkeys ‘days’ and ‘vals’ are either a scalar or list of when the change(s) should take effect and what the new value should be, respectively.

You can also pass parameters to change directly as keyword arguments.

    Parameters
    • pars (dict) – described above
    • kwargs (dict) – passed to Intervention()

**Examples**:

```python
interv = cv.dynamic_pars(n_imports=dict(days=10, vals=100))
interv = cv.dynamic_pars({'beta':{'days':[14, 28], 'vals':[0.005, 0.015]}, 'rel_death_prob':{'days':30, 'vals':2.0}}) # Change beta, and make diagnosed people stop transmitting
```

**apply**(sim)

Loop over the parameters, and then loop over the days, applying them if any are found

class sequence(days, interventions, **kwargs)

    Bases: covasim.interventions.Intervention

This is an example of a meta-intervention which switches between a sequence of interventions.

    Parameters
    • days (list) – the days on which to start applying each intervention
    • interventions (list) – the interventions to apply on those days
    • kwargs (dict) – passed to Intervention()

**Example**:

```python
interv = cv.sequence(days=[10, 51], interventions=[
    cv.test_num(n_tests=[100]*npts),
    cv.test_prob(symptomatic_prob=0.2, asymptomatic_prob=0.002),
])
```

initialize(sim)

Fix the dates

**apply**(sim)

Find the matching day, and see which intervention to activate

class change_beta(days, changes, layers=None, **kwargs)

    Bases: covasim.interventions.Intervention

The most basic intervention – change beta (transmission) by a certain amount on a given day or days. This can be used to represent physical distancing (although clip_edges() is more appropriate for overall changes in mobility, e.g. school or workplace closures), as well as hand-washing, masks, and other behavioral changes that affect transmission rates.
Parameters

- **days** (*int/arr*) – the day or array of days to apply the interventions
- **changes** (*float/arr*) – the changes in beta (1 = no change, 0 = no transmission)
- **layers** (*str/list*) – the layers in which to change beta (default: all)
- **kwargs** (*dict*) – passed to Intervention()

Examples:

```python
interv = cv.change_beta(25, 0.3) # On day 25, reduce overall beta by 70% to 0.3
interv = cv.change_beta([14, 28], [0.7, 1], layers='s') # On day 14, reduce beta...
```

**initialize**(sim)

Fix days and store beta

**apply**(sim)

**class clip_edges**(days, changes, layers=None, **kwargs)

Bases: **covasim.interventions.Intervention**

Isolate contacts by removing them from the simulation. Contacts are treated as “edges”, and this intervention works by removing them from `sim.people.contacts` and storing them internally. When the intervention is over, they are moved back. This intervention has quite similar effects as `change_beta()`, but is more appropriate for modeling the effects of mobility reductions such as school and workplace closures. The main difference is that since `clip_edges()` actually removes contacts, it affects the number of people who would be traced and placed in quarantine if an individual tests positive. It also alters the structure of the network – i.e., compared to a baseline case of 20 contacts and a 2% chance of infecting each, there are slightly different statistics for a beta reduction (i.e., 20 contacts and a 1% chance of infecting each) versus an edge clipping (i.e., 10 contacts and a 2% chance of infecting each).

Parameters

- **days** (*int or array*) – the day or array of days to isolate contacts
- **changes** (*float or array*) – the changes in the number of contacts (1 = no change, 0 = no contacts)
- **layers** (*str or list*) – the layers in which to isolate contacts (if None, then all layers)
- **kwargs** (*dict*) – passed to Intervention()

Examples:

```python
interv = cv.clip_edges(25, 0.3) # On day 25, reduce overall contacts by 70% to 0.3
interv = cv.clip_edges([14, 28], [0.7, 1], layers='s') # On day 14, remove 30% of...
```

**initialize**(sim)

**apply**(sim)

**finalize**(sim)

Ensure the edges get deleted at the end

**class test_num**(daily_tests, symp_test=100.0, quar_test=1.0, quar_policy=None, subtarget=None, ili_prev=None, sensitivity=1.0, loss_prob=0, test_delay=0, start_day=0, end_day=None, swab_delay=None, **kwargs)

Bases: **covasim.interventions.Intervention**
Test the specified number of people per day. Useful for including historical testing data. The probability of a given person getting a test is dependent on the total number of tests, population size, and odds ratios. Compare this intervention with cv.test_prob().

**Parameters**

- **daily_tests** *(arr)* – number of tests per day, can be int, array, or dataframe/series; if integer, use that number every day; if ‘data’ or another string, use loaded data
- **symp_test** *(float)* – odds ratio of a symptomatic person testing (default: 100x more likely)
- **quar_test** *(float)* – probability of a person in quarantine testing (default: no more likely)
- **quar_policy** *(str)* – policy for testing in quarantine: options are ‘start’ (default), ‘end’, ‘both’ (start and end), ‘daily’; can also be a number or a function, see get_quar_inds()
- **subtarget** *(dict)* – subtarget intervention to people with particular indices (format: {'ind': array of indices, or function to return indices from the sim, 'vals': value(s) to apply})
- **ili_prev** *(arr)* – prevalence of influenza-like-illness symptoms in the population; can be float, array, or dataframe/series
- **sensitivity** *(float)* – test sensitivity (default 100%, i.e. no false negatives)
- **loss_prob** *(float)* – probability of the person being lost-to-follow-up (default 0%, i.e. no one lost to follow-up)
- **test_delay** *(int)* – days for test result to be known (default 0, i.e. results available instantly)
- **start_day** *(int)* – day the intervention starts (default: 0, i.e. first day of the simulation)
- **end_day** *(int)* – day the intervention ends
- **swab_delay** *(dict)* – distribution for the delay from onset to swab; if this is present, it is used instead of test_delay
- **kwargs** *(dict)* – passed to Intervention()

**Examples:**

```python
interv = cv.test_num(daily_tests=[0.10*n_people]*npts)
interv = cv.test_num(daily_tests=[0.10*n_people]*npts, subtarget={'inds': cv.true(sim.people.age>50), 'vals': 1.2}) # People over 50 are 20% more likely to test
interv = cv.test_num(daily_tests=[0.10*n_people]*npts, subtarget={'inds': lambda sim: cv.true(sim.people.age>50), 'vals': 1.2}) # People over 50 are 20% more likely to test
interv = cv.test_num(daily_tests='data') # Take number of tests from loaded data, using default column name (new_tests)
interv = cv.test_num(daily_tests='swabs_per_day') # Take number of tests from loaded data using a custom column name
```

**initialize** *(sim)*

Fix the dates and number of tests

**finalize** *(sim)*

Ensure variables with large memory footprints get erased

**apply** *(sim)*
class test_prob(symp_prob, asymp_prob=0.0, symp_quar_prob=None, asymp_quar_prob=None, quar_policy=None, subtarget=None, ili_prev=None, sensitivity=1.0, loss_prob=0.0, test_delay=0, start_day=0, end_day=None, swab_delay=None, **kwargs)

Bases: covasim.interventions.Intervention

Assign each person a probability of being tested for COVID based on their symptom state, quarantine state, and other states. Unlike test_num, the total number of tests not specified, but rather is an output.

Parameters

- **symp_prob** (*float*) – probability of testing a symptomatic (unquarantined) person (default: 0)
- **asymp_prob** (*float*) – probability of testing an asymptomatic (unquarantined) person (default: same as symp_prob)
- **symp_quar_prob** (*float*) – probability of testing a symptomatic quarantined person (default: same as symp_quar_prob)
- **asymp_quar_prob** (*float*) – probability of testing an asymptomatic quarantined person (default: same as asymp_prob)
- **quar_policy** (*str*) – policy for testing in quarantine: options are ‘start’ (default), ‘end’, ‘both’ (start and end), ‘daily’; can also be a number or a function, see get_quar_inds()
- **subtarget** (*dict*) – subtarget intervention to people with particular indices (see test_num() for details)
- **ili_prev** (*float/arr*) – prevalence of influenza-like-illness symptoms in the population; can be float, array, or dataframe/series
- **sensitivity** (*float*) – test sensitivity (default 100%, i.e. no false negatives)
- **loss_prob** (*float*) – probability of the person being lost-to-follow-up (default 0%, i.e. no one lost to follow-up)
- **test_delay** (*int*) – days for test result to be known (default 0, i.e. results available instantly)
- **start_day** (*int*) – day the intervention starts (default: 0, i.e. first day of the simulation)
- **end_day** (*int*) – day the intervention ends (default: no end)
- **swab_delay** (*dict*) – distribution for the delay from onset to swab; if this is present, it is used instead of test_delay
- **kwargs** (*dict*) – passed to Intervention()

Examples:

```python
interv = cv.test_prob(symp_prob=0.1, asymp_prob=0.01)  # Test 10% of symptomatics and 1% of asymptomatics
interv = cv.test_prob(symp_quar_prob=0.4)  # Test 40% of those in quarantine with symptoms
```

**initialize**(sim)

Fix the dates

**finalize**(sim)

Ensure variables with large memory footprints get erased

**apply**(sim)

Perform testing
class contact_tracing (trace_probs=None, trace_time=None, start_day=0, end_day=None, presumptive=False, quar_period=None, capacity=None, **kwargs)

Bases: covasim.interventions.Intervention

Contact tracing of people who are diagnosed. When a person is diagnosed positive (by either test_num() or test_prob(); this intervention has no effect if there is not also a testing intervention active), a certain proportion of the index case’s contacts (defined by trace_prob) are contacted after a certain number of days (defined by trace_time). After they are contacted, they are placed into quarantine (with effectiveness quar_factor, a simulation parameter) for a certain period (defined by quar_period, another simulation parameter). They may also change their testing probability, if test_prob() is defined.

Parameters

- trace_probs (float/dict) – probability of tracing, per layer (default: 100%, i.e. everyone is traced)
- trace_time (float/dict) – days required to trace, per layer (default: 0, i.e. no delay)
- start_day (int) – intervention start day (default: 0, i.e. the start of the simulation)
- end_day (int) – intervention end day (default: no end)
- presumptive (bool) – whether or not to begin isolation and contact tracing on the presumption of a positive diagnosis (default: no)
- capacity (int) – optionally specify a maximum number of newly diagnosed people to trace each day
- quar_period (int) – number of days to quarantine when notified as a known contact. Default value is pars['quar_period']
- kwargs (dict) – passed to Intervention()

Example:

```python
tp = cv.test_prob(symp_prob=0.1, asymp_prob=0.01)
ct = cv.contact_tracing(trace_probs=0.5, trace_time=2)
sim = cv.Sim(interventions=[tp, ct])  # Note that without testing, contact tracing → has no effect
```

initialize (sim)
Process the dates and dictionaries

apply (sim)
Trace and notify contacts

Tracing involves three steps that can independently be overloaded or extended by derived classes

- Select which confirmed cases get interviewed by contact tracers
- Identify the contacts of the confirmed case
- Notify those contacts that they have been exposed and need to take some action

select_cases (sim)
Return people to be traced at this time step

identify_contacts (sim, trace_inds)
Return contacts to notify by trace time

In the base class, the trace time is the same per-layer, but derived classes might provide different functionality e.g. sampling the trace time from a distribution. The return value of this method is a dict keyed by trace time so that the Person object can be easily updated in contact_tracing.notify_contacts

Parameters
• **sim** – Simulation object
• **trace_inds** – Indices of people to trace

Returns: \{trace_time: np.array(inds)\} dictionary storing which people to notify

**notify_contacts** *(sim, contacts)*

Notify contacts

This method represents notifying people that they have had contact with a confirmed case. In this base class, that involves

• Setting the ‘known_contact’ flag and recording the ‘date_known_contact’
• Scheduling quarantine

**Parameters**

• **sim** – Simulation object
• **contacts** – \{trace_time: np.array(inds)\} dictionary storing which people to notify

**class simple_vaccine** *(days, prob=1.0, rel_sus=0.0, rel_symp=0.0, subtarget=None, cumulative=False, **kwargs)*

Bases: *covasim.interventions.Intervention*

Apply a simple vaccine to a subset of the population. In addition to changing the relative susceptibility and the probability of developing symptoms if still infected, this intervention stores several types of data:

• **doses**: the number of vaccine doses per person
• **vaccination_dates**: list of dates per person
• **orig_rel_sus**: relative susceptibility per person at the beginning of the simulation
• **orig_symp_prob**: probability of developing symptoms per person at the beginning of the simulation
• **mod_rel_sus**: modifier on default susceptibility due to the vaccine
• **mod_symp_prob**: modifier on default symptom probability due to the vaccine

**Parameters**

• **days** *(int or array)* – the day or array of days to apply the interventions
• **prob** *(float)* – probability of being vaccinated (i.e., fraction of the population)
• **rel_sus** *(float)* – relative change in susceptibility; 0 = perfect, 1 = no effect
• **rel_symp** *(float)* – relative change in symptom probability for people who still get infected; 0 = perfect, 1 = no effect
• **subtarget** *(dict)* – subtarget intervention to people with particular indices (see test_num() for details)
• **cumulative** *(bool)* – whether cumulative doses have cumulative effects (default False); can also be an array for efficacy per dose, with the last entry used for multiple doses; thus True = [1] and False = [1, 0]
• **kwargs** *(dict)* – passed to Intervention()

Note: this intervention is still under development and should be used with caution. It is intended for use with use_waning=False.

**Examples**:
interv = cv.simple_vaccine(days=50, prob=0.3, rel_sus=0.5, rel_symp=0.1)
interv = cv.simple_vaccine(days=[10,20,30,40], prob=0.8, rel_sus=0.5, ...
→cumulative=[1, 0.3, 0.1, 0]) # A vaccine with efficacy up to the 3rd dose

initialize(sim)
   Fix the dates and store the doses

apply(sim)
   Perform vaccination

class BaseVaccination(vaccine, label=None, **kwargs)
Bases: covasim.interventions.Intervention

Apply a vaccine to a subset of the population.

This base class implements the mechanism of vaccinating people to modify their immunity. It does not implement allocation of the vaccines, which is implemented by derived classes such as cv.vaccinate. The idea is that vaccination involves a series of standard operations to modify cv.People and applications will likely need to modify the vaccine parameters and test potentially complex allocation strategies. These should be accounted for by:

• Custom vaccine parameters being passed in as a dictionary to the vaccine intervention
• Custom vaccine allocations being implemented by a derived class overloading BaseVaccination.select_people. Any additional attributes required to manage the allocation can be defined in the derived class. Refer to cv.vaccinate or cv.vaccinate_sequential for an example of how to implement this.

Some quantities are tracked during execution for reporting after running the simulation. These are:

• vaccinated: whether or not a person is vaccinated
• doses: the number of vaccine doses per person
• vaccination_dates: integer; dates of all doses for this vaccine

Parameters

• vaccine(dict/str) – which vaccine to use; see below for dict parameters
• label(str) – if vaccine is supplied as a dict, the name of the vaccine
• booster(boolean) – whether the vaccine is a booster, i.e. whether vaccinated people are eligible
• kwargs(dict) – passed to Intervention()

If vaccine is supplied as a dictionary, it must have the following parameters:
EITHER - nab_init: the initial antibody level (higher = more protection) - nab_boost: how much of a boost being vaccinated on top of a previous dose or natural infection provides OR - target_eff: the target efficacy from which to calculate initial antibody and boosting. must be supplied as a list, where length of list is equal to number of doses - nab_eff: the waning efficacy of neutralizing antibodies at preventing infection - doses: the number of doses required to be fully vaccinated with this vaccine - interval: the interval between doses (integer) - entries for efficacy against each of the variants (e.g. b117)

See parameters.py for additional examples of these parameters.

initialize(sim)

finalize(sim)
   Ensure variables with large memory footprints get erased
**select_people** *(sim)*
Return an array of indices of people to vaccinate. Derived classes must implement this function to determine who to vaccinate at each timestep.

:param sim: A cv.Sim instance

Returns: Array of person indices

**vaccinate** *(sim, vacc_inds, t=None)*
Vaccinate people
This method applies the vaccine to the requested people indices. The indices of people vaccinated is returned. These may be different to the requested indices, because anyone that is dead will be skipped, as well as anyone already fully vaccinated (if booster=False). This could occur if a derived class does not filter out such people in its select_people method.

**Parameters**
- **sim** – A cv.Sim instance
- **vacc_inds** – An array of person indices to vaccinate
- **t** – Optionally override the day on which vaccinations are recorded for historical vaccination

Returns: An array of person indices of people vaccinated

**apply** *(sim)*
Perform vaccination each timestep

**shrink** *(in_place=True)*
Shrink vaccination intervention

**vaccinate** *(args, **kwargs)*
Wrapper function for vaccinate_prob() and vaccinate_num(). If the num_doses argument is used, will call vaccinate_num(); else, calls vaccinate_prob().

**Examples:**
```python
vx1 = cv.vaccinate(vaccine='pfizer', days=30, prob=0.7)
vx2 = cv.vaccinate(vaccine='pfizer', num_doses=100)
```

**class vaccinate_prob**(vaccine, days, label=None, prob=None, subtarget=None, booster=False, **kwargs)
Bases: covasim.interventions.BaseVaccination
Probability-based vaccination
This vaccine intervention allocates vaccines parametrized by the daily probability of being vaccinated.

**Parameters**
- **vaccine** *(dict/str)* – which vaccine to use; see below for dict parameters
- **label** *(str)* – if vaccine is supplied as a dict, the name of the vaccine
- **days** *(int/arr)* – the day or array of days to apply the interventions
- **prob** *(float)* – probability of being vaccinated (i.e., fraction of the population)
- **booster** *(bool)* – whether it’s a booster (i.e. targeted to vaccinated people) or not
- **subtarget** *(dict)* – subtarget intervention to people with particular indices (see test_num() for details)
- **kwargs** *(dict)* – passed to Intervention()

If vaccine is supplied as a dictionary, it must have the following parameters:
• nab_eff: the waning efficacy of neutralizing antibodies at preventing infection
• nab_init: the initial antibody level (higher = more protection)
• nab_boost: how much of a boost being vaccinated on top of a previous dose or natural infection provides
• doses: the number of doses required to be fully vaccinated
• interval: the interval between doses (integer)
• entries for efficacy against each of the strains (e.g. b117)

See parameters.py for additional examples of these parameters.

Example:

```python
pfizer = cv.vaccinate_prob(vaccine='pfizer', days=30, prob=0.7)
cv.Sim(interventions=pfizer, use_waning=True).run().plot()
```

```python
initialize(sim)
select_people(sim)
```

```python
class vaccinate_num(vaccine, num_doses, booster=False, subtarget=None, sequence=None, **kwargs)
```

Bases: covasim.interventions.BaseVaccination

This vaccine intervention allocates vaccines in a pre-computed order of distribution, at a specified rate of doses per day. Second doses are prioritized each day.

Parameters

• **vaccine** (dict/str) – which vaccine to use; see below for dict parameters
• **label** (str) – if vaccine is supplied as a dict, the name of the vaccine
• **booster** (bool) – whether it’s a booster (i.e. targeted to vaccinated people) or not
• **subtarget** (dict) – subtarget intervention to people with particular indices (see test_num() for details)

• **sequence** – Specify the order in which people should get vaccinated. This can be
  – An array of person indices in order of vaccination priority
  – A callable that takes in cv.People and returns an ordered sequence. For example, to vaccinate people in descending age order, `def age_sequence(people): return np.argsort(-people.age)` would be suitable.
  – The shortcut ‘age’, which does prioritization by age (see below for implementation) If not specified, people will be randomly ordered.

• **num_doses** – Specify the number of doses per day. This can take three forms
  – A scalar number of doses per day
  – A dict keyed by day/date with the number of doses e.g. `{2:10000, '2021-05-01':20000}`. Any dates are converted to simulation days in `initialize()` which will also copy the dictionary passed in.
  – A callable that takes in a cv.Sim and returns a scalar number of doses. For example, `def doses(sim): return 100 if sim.t > 10 else 0` would be suitable

• ****kwargs** – Additional arguments passed to cv.BaseVaccination
Example::

    pfizer = cv.vaccinate_num(vaccine='pfizer', sequence='age', num_doses=100)
    cv.Sim(interventions=pfizer, use_waning=True).run().plot()

initialize(sim)

select_people(sim)

prior_immunity(*args, **kwargs)

Wrapper function for historical_wave and historical_vaccinate_prob. If the vaccine keyword is present then historical_vaccinate_prob will be used. Otherwise historical_wave is used.

Examples:

    pim1 = cv.prior_immunity(vaccine='pfizer', days=[-30], prob=0.7)
    pim2 = cv.prior_immunity(120, 0.05)

New in version 3.1.0.

class historical_vaccinate_prob(vaccine, days, label=None, prob=1.0, subtarget=None, compliance=1.0, **kwargs)

Bases: covasim.interventions.BaseVaccination

Probability-based historical vaccination

This vaccine intervention allocates vaccines parametrized by the daily probability of being vaccinated. Unlike cv.vaccinate_prob this function allows vaccination prior to t=0 (and continuing into the simulation).

If any people are infected at the t=0 timestep (e.g. seed infections), this finds those people and will re-infect them at the end of the historical vaccination. Thus you may have breakthrough infections and this might affect other interventions to initialize a population.

Parameters

- **vaccine**(dict/str) – which vaccine to use; see below for dict parameters
- **label**(str) – if vaccine is supplied as a dict, the name of the vaccine
- **days**(int/arr) – the day or array of days to apply the interventions
- **prob**(float) – probability of being vaccinated (i.e., fraction of the population)
- **subtarget**(dict) – subtarget intervention to people with particular indices (see test_num() for details)
- **compliance**(float/arr) – compliance of the person to take each dose (if scalar then applied per dose)
- **kwargs**(dict) – passed to Intervention()

If vaccine is supplied as a dictionary, it must have the following parameters:

- **nab_eff**: the waning efficacy of neutralizing antibodies at preventing infection
- **nab_init**: the initial antibody level (higher = more protection)
- **nab_boost**: how much of a boost being vaccinated on top of a previous dose or natural infection provides
- **doses**: the number of doses required to be fully vaccinated
- **interval**: the interval between doses
- **entries** for efficacy against each of the strains (e.g. b117)
See `parameters.py` for additional examples of these parameters.

Example:

```python
pfizer = cv.historical_vaccinate_prob(vaccine='pfizer', days=np.arange(-30,0), prob=0.007) # 30-day vaccination campaign
cv.Sim(interventions=pfizer).run().plot()
```

New in version 3.1.0.

initialize(sim)

select_people(sim, t=None)

static process_days(sim, days, return_dates=False)

Ensure lists of days are in consistent format. Used by change_beta, clip_edges, and some analyzers. Optionally return dates as well as days. If days is callable, leave unchanged.

static estimate_prob(duration, coverage)

Estimate the per-day probability to achieve desired population coverage for a campaign of fixed duration and fixed per-day probability of a person being vaccinated

Parameters

- `duration` – length of campaign in days
- `coverage` – target coverage of campaign

Example:

```python
prob = historical_vaccinate.estimate_prob(duration=180, coverage=0.70)
```

static NB_cdf(k, p, r=1)

note that the NB distribution shows the fraction

class historical_wave(days_prior, prob, dist=None, subtarget=None, variant=None, **kwargs)

Bases: covasim.interventions.Intervention

Imprint a historical (pre t=0) wave of infections in the population NAbs

Parameters

- `days_prior` (int/str/list) – offset relative to t=0 for the wave (median/par1 value) or median date if a string like “2021-11-15”
- `prob` (float/list) – probability of infection during the wave
- `dist` (dict/list) – passed to covasim.utils.sample to set wave shape (default gaussian with FWHM of 5 weeks)
- `subtarget` (dict/list) – subtarget intervention to people with particular indices (see test_num() for details)
- `variants` (str/list) – name of variant associated with the wave
- `kwargs` (dict) – passed to Intervention()

Example::

```python
cv.Sim(interventions=cv.historical_wave(120, 0.30)).run().plot()
```

New in version 3.1.0.

apply(sim)
1.10.2.6 covasim.misc module

Miscellaneous functions that do not belong anywhere else

date (obj, *args, start_date=None, readformat=None, outformat=None, as_date=True, **kwargs)

Convert any reasonable object – a string, integer, or datetime object, or list/array of any of those – to a date object. To convert an integer to a date, you must supply a start date.

Caution: while this function and readdate() are similar, and indeed this function calls readdate() if the input is a string, in this function an integer is treated as a number of days from start_date, while for readdate() it is treated as a timestamp in seconds. To change

Parameters

• obj (str, int, date, datetime, list, array) – the object to convert
• args (str, int, date, datetime) – additional objects to convert
• start_date (str, date, datetime) – the starting date, if an integer is supplied
• readformat (str/list) – the format to read the date in; passed to sc.readdate()
• outformat (str) – the format to output the date in, if returning a string
• as_date (bool) – whether to return as a datetime date instead of a string

Returns either a single date object, or a list of them (matching input data type where possible)

Return type dates (date or list)

Examples:

```
sc.date('2020-04-05') # Returns datetime.date(2020, 4, 5)
sc.date([[35,36,37], start_date='2020-01-01', as_date=False) # Returns ['2020-02-05 →', '2020-02-06', '2020-02-07']
sc.date(1923288822, readformat='posix') # Interpret as a POSIX timestamp
```

New in version 1.0.0. New in version 1.2.2: “readformat” argument; renamed “dateformat” to “outformat”

day (obj, *args, start_date=None, **kwargs)

Convert a string, date/datetime object, or int to a day (int), the number of days since the start day. See also sc.date() and sc.daydiff(). If a start day is not supplied, it returns the number of days into the current year.

Parameters

• obj (str, date, int, list, array) – convert any of these objects to a day relative to the start day
• args (list) – additional days
• start_date (str or date) – the start day; if none is supplied, return days since (supplied year)-01-01.

Returns the day(s) in simulation time (matching input data type where possible)

Return type days (int or list)

Examples:

```
sc.day(sc.now()) # Returns how many days into the year we are
sc.day(['2021-01-21', '2024-04-04'], start_date='2022-02-22') # Days can be
→ positive or negative
```

New in version 1.0.0. New in version 1.2.2: renamed “start_day” to “start_date”
**daydiff (**args **)**
Convenience function to find the difference between two or more days. With only one argument, calculate days since 2020-01-01.

**Examples:**
```python
diff = sc.daydiff('2020-03-20', '2020-04-05') # Returns 16
diffs = sc.daydiff('2020-03-20', '2020-04-05', '2020-05-01') # Returns [16, 26]
```
New in version 1.0.0.

**date_range (start_date, end_date, inclusive=True, as_date=False, dateformat=None)**
Return a list of dates from the start date to the end date. To convert a list of days (as integers) to dates, use sc.date() instead.

**Parameters**
- **start_date (int/str/date)** – the starting date, in any format
- **end_date (int/str/date)** – the end date, in any format
- **inclusive (bool)** – if True (default), return to end_date inclusive; otherwise, stop the day before
- **as_date (bool)** – if True, return a list of datetime.date objects instead of strings
- **dateformat (str)** – passed to date()

**Example:**
```python
dates = sc.daterange('2020-03-01', '2020-04-04')
```
New in version 1.0.0.

**load_data (datafile, columns=None, calculate=True, check_date=True, verbose=True, start_day=None, **kwargs)**
Load data for comparing to the model output, either from file or from a dataframe.

**Parameters**
- **datafile (str or df)** – if a string, the name of the file to load (either Excel or CSV); if a dataframe, use directly
- **columns (list)** – list of column names (otherwise, load all)
- **calculate (bool)** – whether to calculate cumulative values from daily counts
- **check_date (bool)** – whether to check that a ‘date’ column is present
- **start_day (date)** – if the ‘date’ column is provided as integer number of days, consider them relative to this
- **kwargs (dict)** – passed to pd.read_excel()

**Returns** pandas dataframe of the loaded data

**Return type** data (dataframe)

**load (**args, do_migrate=True, update=True, verbose=True, **kwargs **)**
Convenience method for sc.loadobj() and equivalent to cv.Sim.load() or cv.Scenarios.load().

**Parameters**
- **filename (str)** – file to load
- **do_migrate (bool)** – whether to migrate if loading an old object
Covasim, Release 3.1.1

- **update** (*bool*) – whether to modify the object to reflect the new version
- **verbose** (*bool*) – whether to print migration information
- **args** (*list*) – passed to `sc.loadobj()`
- **kwargs** (*dict*) – passed to `sc.loadobj()`

**Returns** Loaded object

**Examples:**

```python
sim = cv.load('calib.sim')  # Equivalent to `cv.Sim.load('calib.sim')`
sceans = cv.load(filename='school-closures.scens', folder='schools')
```

**save** (*args, **kwargs*)
Convenience method for `sc.saveobj()` and equivalent to `cv.Sim.save()` or `cv.Scenarios.save()`.

**Parameters**

- **filename** (*str*) – file to save to
- **obj** (*object*) – object to save
- **args** (*list*) – passed to `sc.saveobj()`
- **kwargs** (*dict*) – passed to `sc.saveobj()`

**Returns** Filename the object is saved to

**Examples:**

```python
cv.save('calib.sim', sim)  # Equivalent to `sim.save('calib.sim')`
cv.save(filename='school-closures.scens', folder='schools', obj=sceans)
```

**savefig** (*filename=None, comments=None, **kwargs*)
Wrapper for Matplotlib’s `savefig()` function which automatically stores Covasim metadata in the figure. By default, saves (git) information from both the Covasim version and the calling function. Additional comments can be added to the saved file as well. These can be retrieved via `cv.get_png_metadata()`. Metadata can also be stored for SVG and PDF formats, but cannot be automatically retrieved.

**Parameters**

- **filename** (*str*) – name of the file to save to (default, timestamp)
- **comments** (*str*) – additional metadata to save to the figure
- **kwargs** (*dict*) – passed to `savefig()`

**Example:**

```python
cv.Sim().run(do_plot=True)
filename = cv.savefig()
```

**migrate** (*obj, update=True, verbose=True, die=False*)
Define migrations allowing compatibility between different versions of saved files. Usually invoked automatically upon load, but can be called directly by the user to load custom objects, e.g. lists of sims.

Currently supported objects are sims, multisims, scenarios, and people.

**Parameters**

- **obj** (*any*) – the object to migrate
- **update** (*bool*) – whether to update version information to current version after successful migration

**Examples:**

```python
```
• **verbose** (bool) – whether to print warnings if something goes wrong

• **die** (bool) – whether to raise an exception if something goes wrong

**Returns** The migrated object

**Example:**

```python
sims = cv.load('my-list-of-sims.obj')
sims = [cv.migrate(sim) for sim in sims]
```

git_info (filename=None, check=False, comments=None, old_info=None, die=False, indent=2, verbose=True, frame=2, **kwargs)

Get current git information and optionally write it to disk. Simplest usage is `cv.git_info(__file__)`

**Parameters**

• **filename** (str) – name of the file to write to or read from

• **check** (bool) – whether or not to compare two git versions

• **comments** (dict) – additional comments to include in the file

• **old_info** (dict) – dictionary of information to check against

• **die** (bool) – whether or not to raise an exception if the check fails

• **indent** (int) – how many indents to use when writing the file to disk

• **verbose** (bool) – detail to print

• **frame** (int) – how many frames back to look for caller info

• **kwargs** (dict) – passed to sc.loadjson() (if check=True) or sc.savejson() (if check=False)

**Examples:**

```python
cv.git_info()  # Return information
cv.git_info(__file__)  # Writes to disk
cv.git_info('covasim_version.gitinfo')  # Writes to disk
```

```python
cv.check_version('covasim_version.gitinfo', check=True)  # Checks that current version matches saved file
```

**check_version** (expected, die=False, verbose=True)

Get current git information and optionally write it to disk. The expected version string may optionally start with ‘>=’ or ‘<=' (= is implied otherwise), but other operators (e.g. ‘~’=) are not supported. Note that e.g. ‘>’ is interpreted to mean ‘>=’.

**Parameters**

• **expected** (str) – expected version information

• **die** (bool) – whether or not to raise an exception if the check fails

**Example:**

```python
cv.check_version('>=1.7.0', die=True)  # Will raise an exception if an older version is used
```

**check_save_version** (expected=None, filename=None, die=False, verbose=True, **kwargs)

A convenience function that bundles check_version with git_info and saves automatically to disk from the calling file. The idea is to put this at the top of an analysis script, and commit the resulting file, to keep track of which version of Covasim was used.

**Parameters**


• **expected** *(str)* – expected version information
• **filename** *(str)* – file to save to; if None, guess based on current file name
• **kwargs** *(dict)* – passed to git_info(), and thence to sc.savejson()

**Examples:**

```python
cv.check_save_version()
cv.check_save_version('1.3.2', filename='script.gitinfo', comments='This is the main analysis script')
cv.check_save_version('1.7.2', folder='gitinfo', comments={'SynthPops':sc.gitinfo(sp.__file__)})
```

**get_version_pars** *(version, verbose=True)*

Function for loading parameters from the specified version.

Parameters will be loaded for Covasim ‘as at’ the requested version i.e. the most recent set of parameters that is <= the requested version. Available parameter values are stored in the regression folder. If parameters are available for versions 1.3, and 1.4, then this function will return the following

• If parameters for version ‘1.3’ are requested, parameters will be returned from ‘1.3’
• If parameters for version ‘1.3.5’ are requested, parameters will be returned from ‘1.3’, since Covasim at version 1.3.5 would have been using the parameters defined at version 1.3.
• If parameters for version ‘1.4’ are requested, parameters will be returned from ‘1.4’

**Parameters**

- **version** *(str)* – the version to load parameters from

**Returns**

Dictionary of parameters from that version

**get_png_metadata** *(filename, output=False)*

Read metadata from a PNG file. For use with images saved with cv.savefig(). Requires pillow, an optional dependency. Metadata retrieval for PDF and SVG is not currently supported.

**Parameters**

- **filename** *(str)* – the name of the file to load the data from

**Example:**

```python
cv.Sim().run(do_plot=True)
cv.savefig('covasim.png')
cv.get_png_metadata('covasim.png')
```

**get_doubling_time** *(sim, series=None, interval=None, start_day=None, end_day=None, moving_window=None, exp_approx=False, max_doubling_time=100, eps=0.001, verbose=None)*

Alternate method to calculate doubling time (one is already implemented in the sim object).

**Examples:**

```python
cv.get_doubling_time(sim, interval=[3,30]) # returns the doubling time over the given interval (single float)
cv.get_doubling_time(sim, interval=[3,30], moving_window=3) # returns doubling times calculated over moving windows (array)
```

**compute_gof** *(actual, predicted, normalize=True, use_frac=False, use_squared=False, as_scalar='none', eps=1e-09, skestimator=None, estimator=None, **kwargs)*

Calculate the goodness of fit. By default use normalized absolute error, but highly customizable. For example, mean squared error is equivalent to setting normalize=False, use_squared=True, as_scalar='mean'.

**Parameters**
• **actual** *(arr)* – array of actual (data) points
• **predicted** *(arr)* – corresponding array of predicted (model) points
• **normalize** *(bool)* – whether to divide the values by the largest value in either series
• **use_frac** *(bool)* – convert to fractional mismatches rather than absolute
• **use_squared** *(bool)* – square the mismatches
• **as_scalar** *(str)* – return as a scalar instead of a time series: choices are sum, mean, median
• **eps** *(float)* – to avoid divide-by-zero
• **skestimator** *(str)* – if provided, use this scikit-learn estimator instead
• **estimator** *(func)* – if provided, use this custom estimator instead
• **kwargs** *(dict)* – passed to the scikit-learn or custom estimator

**Returns**  
array of goodness-of-fit values, or a single value if as_scalar is True

**Return type**  
gofs (arr)

**Examples:**

```python
x1 = np.cumsum(np.random.random(100))
x2 = np.cumsum(np.random.random(100))

e1 = compute_gof(x1, x2)  # Default, normalized absolute error

e2 = compute_gof(x1, x2, normalize=False, use_frac=False)  # Fractional error

e3 = compute_gof(x1, x2, normalize=False, use_squared=True, as_scalar='mean')  # Mean squared error

e4 = compute_gof(x1, x2, skestimator='mean_squared_error')  # Scikit-learn's MSE

# Protected method

e5 = compute_gof(x1, x2, as_scalar='median')  # Normalized median absolute error -- highly robust
```

### 1.10.2.7 covasim.parameters module

Set the parameters for Covasim.

**make_pars** *(set_prognoses=False, prog_by_age=True, version=None, **kwargs)*

Create the parameters for the simulation. Typically, this function is used internally rather than called by the user; e.g. typical use would be to do sim = cv.Sim() and then inspect sim.pars, rather than calling this function directly.

**Parameters**

• **set_prognoses** *(bool)* – whether or not to create prognoses (else, added when the population is created)

• **prog_by_age** *(bool)* – whether or not to use age-based severity, mortality etc.

• **kwars** *(dict)* – any additional kwars are interpreted as parameter names

• **version** *(str)* – if supplied, use parameters from this Covasim version

**Returns**  
the parameters of the simulation

**Return type**  
pars (dict)
reset_layer_pars (pars, layer_keys=None, force=False)
Helper function to set layer-specific parameters. If layer keys are not provided, then set them based on the population type. This function is not usually called directly by the user, although it can sometimes be used to fix layer key mismatches (i.e. if the contact layers in the population do not match the parameters). More commonly, however, mismatches need to be fixed explicitly.

Parameters

- pars (dict) – the parameters dictionary
- layer_keys (list) – the layer keys of the population, if available
- force (bool) – reset the parameters even if they already exist

get_prognoses (by_age=True, version=None)
Return the default parameter values for prognoses
The prognosis probabilities are conditional given the previous disease state.

Parameters by_age (bool) – whether to use age-specific values (default true)

Returns the dictionary of prognosis probabilities

Return type prog_pars (dict)

get_variant_choices ()
Define valid pre-defined variant names

get_vaccine_choices ()
Define valid pre-defined vaccine names

1.10.2.8 covasim.people module

Defines the People class and functions associated with making people and handling the transitions between states (e.g., from susceptible to infected).

class People (pars, strict=True, **kwargs)
Bases: covasim.base.BasePeople
A class to perform all the operations on the people. This class is usually not invoked directly, but instead is created automatically by the sim. The only required input argument is the population size, but typically the full parameters dictionary will get passed instead since it will be needed before the People object is initialized.

Note that this class handles the mechanics of updating the actual people, while BasePeople takes care of housekeeping (saving, loading, exporting, etc.). Please see the BasePeople class for additional methods.

Parameters

- pars (dict) – the sim parameters, e.g. sim.pars – alternatively, if a number, interpreted as pop_size
- strict (bool) – whether or not to only create keys that are already in self.meta.person; otherwise, let any key be set

- kwargs (dict) – the actual data, e.g. from a popdict, being specified

Examples:

```python
ppl1 = cv.People(2000)
sim = cv.Sim()
ppl2 = cv.People(sim.pars)
```
init_flows()
    Initialize flows to be zero

initialize()
    Perform initializations

set_prognoses()
    Set the prognoses for each person based on age during initialization. Need to reset the seed because viral loads are drawn stochastically.

update_states_pre(t)
    Perform all state updates at the current timestep

update_states_post()
    Perform post-timestep updates

update_contacts()
    Refresh dynamic contacts, e.g. community

check_inds(current, date, filter_inds=None)
    Return indices for which the current state is false and which meet the date criterion

check_infectious()
    Check if they become infectious

check_symptomatic()
    Check for new progressions to symptomatic

check_severe()
    Check for new progressions to severe

check_critical()
    Check for new progressions to critical

check_recovery(inds=None, filter inds=’is_exp’)
    Check for recovery.
    More complex than other functions to allow for recovery to be manually imposed for a specified set of indices.

check_death()
    Check whether or not this person died on this timestep

check_diagnosed()
    Check for new diagnoses. Since most data are reported with diagnoses on the date of the test, this function reports counts not for the number of people who received a positive test result on a day, but rather, the number of people who were tested on that day who are schedule to be diagnosed in the future.

check_quar()
    Update quarantine state

make_naive(inds, reset_vx=False)
    Make a set of people naive. This is used during dynamic resampling.

    Parameters
        * inds (array) – list of people to make naive
        * reset_vx (bool) – whether to reset vaccine-derived immunity

make_nonnaive(inds, set Recovered=False, date_recovered=0)
    Make a set of people non-naive.
This can be done either by setting only susceptible and naive states, or else by setting them as if they have been infected and recovered.

**infect** *(inds, hosp_max=None, icu_max=None, source=None, layer=None, variant=0)*

Infect people and determine their eventual outcomes.

- Every infected person can infect other people, regardless of whether they develop symptoms
- Infected people that develop symptoms are disaggregated into mild vs. severe (=requires hospitalization) vs. critical (=requires ICU)
- Every asymptomatic, mildly symptomatic, and severely symptomatic person recovers
- Critical cases either recover or die
- If the simulation is being run with waning, this method also sets/updates agents’ neutralizing antibody levels

Method also deduplicates input arrays in case one agent is infected many times and stores who infected whom in infection_log list.

**Parameters**

- **inds** (*array*) – array of people to infect
- **hosp_max** (*bool*) – whether or not there is an acute bed available for this person
- **icu_max** (*bool*) – whether or not there is an ICU bed available for this person
- **source** (*array*) – source indices of the people who transmitted this infection (None if an importation or seed infection)
- **layer** (*str*) – contact layer this infection was transmitted on
- **variant** (*int*) – the variant people are being infected by

**Returns** number of people infected

**Return type** count (int)

**test** *(inds, test_sensitivity=1.0, loss_prob=0.0, test_delay=0)*

Method to test people. Typically not to be called by the user directly; see the test_num() and test_prob() interventions.

**Parameters**

- **inds** – indices of who to test
- **test_sensitivity** (*float*) – probability of a true positive
- **loss_prob** (*float*) – probability of loss to follow-up
- **test_delay** (*int*) – number of days before test results are ready

**Schedule_quarantine** *(inds, start_date=None, period=None)*

Schedule a quarantine. Typically not called by the user directly except via a custom intervention; see the contact_tracing() intervention instead.

This function will create a request to quarantine a person on the start_date for a period of time. Whether they are on an existing quarantine that gets extended, or whether they are no longer eligible for quarantine, will be checked when the start_date is reached.

**Parameters**

- **inds** (*int*) – indices of who to quarantine, specified by check_quar()
- **start_date** (*int*) – day to begin quarantine (defaults to the current day, sim.t)
• **period** (*int*) – quarantine duration (defaults to `pars['quar_period']`)

**plot** (*args, **kwargs*)
Plot statistics of the population – age distribution, numbers of contacts, and overall weight of contacts (number of contacts multiplied by beta per layer).

**Parameters**
- **bins** (*arr*) – age bins to use (default, 0-100 in one-year bins)
- **width** (*float*) – bar width
- **font_size** (*float*) – size of font
- **alpha** (*float*) – transparency of the plots
- **fig_args** (*dict*) – passed to `pl.figure()`
- **axis_args** (*dict*) – passed to `pl.subplots_adjust()`
- **plot_args** (*dict*) – passed to `pl.plot()`
- **do_show** (*bool*) – whether to show the plot
- **fig** (*fig*) – handle of existing figure to plot into

**story** (*uid, *args*)
Print out a short history of events in the life of the specified individual.

**Parameters**
- **uid** (*int/list*) – the person or people whose story is being regaled
- **args** (*list*) – these people will tell their stories too

**Example:**
```
sim = cv.Sim(pop_type='hybrid', verbose=0)
sim.run()
sim.people.story(12)
sim.people.story(795)
```

### 1.10.2.9 covasim.plotting module

Core plotting functions for simulations, multisims, and scenarios.

Also includes Plotly-based plotting functions to supplement the Matplotlib based ones that are of the Sim and Scenarios objects. Intended mostly for use with the webapp.

**date_formatter** (*start_day=*, *dateformat=*, *interval=*, *start=*, *end=*, *ax=*, *sim=*)
Create an automatic date formatter based on a number of days and a start day.

Wrapper for Matplotlib's date formatter. Note, start_day is not required if the axis uses dates already. To be used in conjunction with setting the x-axis tick label formatter.

**Parameters**
- **start_day** (*str/date*) – the start day, either as a string or date object
- **dateformat** (*str*) – the date format (default '%b-%d')
- **interval** (*int*) – if supplied, the interval between ticks (must supply an axis also to take effect)
• **start** *(str/int)* – if supplied, the lower limit of the axis

• **end** *(str/int)* – if supplied, the upper limit of the axis

• **ax** *(axes)* – if supplied, automatically set the x-axis formatter for this axis

• **sim** *(Sim)* – if supplied, get the start day from this

Examples:

```python
# Automatically configure the axis with default option
cv.date_formatter(sim=sim, ax=ax)

# Manually configure
ax = pl.subplot(111)
ax.plot(np.arange(60), np.random.random(60))
formatter = cv.date_formatter(start_day='2020-04-04', interval=7, start='2020-05-01', end=50, dateformat='%Y-%m-%d', ax=ax)
ax.xaxis.set_major_formatter(formatter)
```

```
plot_sim(to_plot=None, sim=None, do_save=None, fig_path=None, fig_args=None, plot_args=None, scatter_args=None, axis_args=None, fill_args=None, legend_args=None, date_args=None, show_args=None, mpl_args=None, n_cols=None, grid=False, commaticks=True, setylim=True, log_scale=False, colors=None, labels=None, do_show=None, sep_figs=False, fig=None, ax=None, **kwargs)
```

Plot the results of a single simulation – see Sim.plot() for documentation

```
plot_scens(to_plot=None, scens=None, do_save=None, fig_path=None, fig_args=None, plot_args=None, scatter_args=None, axis_args=None, fill_args=None, legend_args=None, date_args=None, show_args=None, mpl_args=None, n_cols=None, grid=False, commaticks=True, setylim=True, log_scale=False, colors=None, labels=None, do_show=None, sep_figs=False, fig=None, ax=None, **kwargs)
```

Plot the results of a scenario – see Scenarios.plot() for documentation

```
plot_result(key, sim=None, fig_args=None, plot_args=None, axis_args=None, scatter_args=None, date_args=None, mpl_args=None, grid=False, commaticks=True, setylim=True, color=None, label=None, do_show=None, do_save=False, fig_path=None, fig=None, ax=None, **kwargs)
```

Plot a single result – see Sim.plot_result() for documentation

```
plot_compare(df, log_scale=True, fig_args=None, axis_args=None, mpl_args=None, grid=False, commaticks=True, setylim=True, color=None, label=None, do_show=None, fig_path=None, fig=None, ax=None, **kwargs)
```

Plot a MultiSim comparison – see MultiSim.plot_compare() for documentation

```
plot_people(people, bins=None, width=1.0, alpha=0.6, fig_args=None, axis_args=None, plot_args=None, do_show=None, fig=None)
```

Plot statistics of a population – see People.plot() for documentation

```
plotly_sim(sim, do_show=False)
```

Main simulation results – parallel of sim.plot()  

```
plotly_people(sim, do_show=False)
```

Plot a “cascade” of people moving through different states

```
plotly_animate(sim, do_show=False)
```

Plot an animation of each person in the sim

1.10.2.10 covasim.population module

Defines functions for making the population.
*make_people*(sim, popdict=None, save_pop=False, popfile=None, die=True, reset=False, verbose=None, **kwargs)

Make the actual people for the simulation. Usually called via sim.initialize(), but can be called directly by the user.

**Parameters**

- **sim** (Sim) – the simulation object; population parameters are taken from the sim object
- **popdict** (dict) – if supplied, use this population dictionary instead of generating a new one
- **save_pop** (bool) – whether to save the population to disk
- **popfile** (str) – if so, the filename to save to
- **die** (bool) – whether or not to fail if synthetic populations are requested but not available
- **reset** (bool) – whether to force population creation even if self.popdict/self.people exists
- **verbose** (bool) – level of detail to print
- **kwargs** (dict) – passed to make_randpop() or make_synthpop()

**Returns**

people (People)

*make_randpop*(pars, use_age_data=True, use_household_data=True, sex_ratio=0.5, microstructure='random', **kwargs)

Make a random population, with contacts.

This function returns a “popdict” dictionary, which has the following (required) keys:

- **uid**: an array of (usually consecutive) integers of length N, uniquely identifying each agent
- **age**: an array of floats of length N, the age in years of each agent
- **sex**: an array of integers of length N (not currently used, so does not have to be binary)
- **contacts**: list of length N listing the contacts; see make_random_contacts() for details
- **layer_keys**: a list of strings representing the different contact layers in the population; see make_random_contacts() for details

**Parameters**

- **pars** (dict) – the parameter dictionary or simulation object
- **use_age_data** (bool) – whether to use location-specific age data
- **use_household_data** (bool) – whether to use location-specific household size data
- **sex_ratio** (float) – proportion of the population that is male (not currently used)
- **microstructure** (bool) – whether or not to use the microstructuring algorithm to group contacts
- **kwargs** (dict) – passed to contact creation method (e.g., make_hybrid_contacts)

**Returns**

a dictionary representing the population, with the following keys for a population of N agents with M contacts between them:

**Return type** popdict (dict)

*make_random_contacts*(pop_size, n, overshoot=1.2, dispersion=None, mapping=None)

Make random static contacts for a single layer as an edgelist.
Parameters

- **pop_size** (*int*) – number of agents to create contacts between (N)
- **n** (*int*) – the average number of contacts per person for this layer
- **overshoot** (*float*) – to avoid needing to take multiple Poisson draws
- **dispersion** (*float*) – if not None, use a negative binomial distribution with this dispersion parameter instead of Poisson to make the contacts
- **mapping** (*array*) – optionally map the generated indices onto new indices

Returns

Dictionary of two arrays defining UIDs of the edgelist (sources and targets)

New in 3.1.1: optimized and updated arguments.

**make_microstructured_contacts** (*pop_size, cluster_size, mapping=None*)

Create microstructured contacts – i.e. for households.

Parameters

- **pop_size** (*int*) – total number of people
- **cluster_size** (*int*) – the average size of each cluster (Poisson-sampled)

New in version 3.1.1: optimized updated arguments.

**make_hybrid_contacts** (*pop_size, ages, contacts, school_ages=None, work_ages=None*)

Create “hybrid” contacts – microstructured contacts for households and random contacts for schools and workplaces, both of which have extremely basic age structure. A combination of both make_random_contacts() and make_microstructured_contacts().

**make_synthpop** (*sim=None, population=None, layer_mapping=None, community_contacts=None, **kwargs*)

Make a population using SynthPops, including contacts. Usually called automatically, but can also be called manually. Either a simulation object or a population must be supplied; if a population is supplied, transform it into the correct format; otherwise, create the population and then transform it.

Parameters

- **sim** (*Sim*) – a Covasim simulation object
- **population** (*list*) – a pre-generated SynthPops population (otherwise, create a new one)
- **layer_mapping** (*dict*) – a custom mapping from SynthPops layers to Covasim layers
- **community_contacts** (*int*) – if a simulation is not supplied, create this many community contacts on average
- **kwargs** (*dict*) – passed to sp.make_population()

Example:

```python
sim = cv.Sim(pop_type='synthpops')
sim.popdict = cv.make_synthpop(sim)
sim.run()
```

1.10.2.11 **covasim.requirements module**

Check that correct versions of each library are installed, and print warnings or errors if not.
check_sciris()
    Check that Sciris is available and the right version

check_synthpops(verbos=False, die=False)
    Check whether synthpops is available

1.10.2.12 covasim.run module

Functions and classes for running multiple Covasim runs.

make_metapars()
    Create default metaparameters for a Scenarios run

class MultiSim(sims=None, base_sim=None, label=None, initialize=False, **kwargs)
    Bases: covasim.base.FlexPretty

    Class for running multiple copies of a simulation. The parameter n_runs controls how many copies of the
    simulation there will be, if a list of sims is not provided. This is the main class that’s used to run multiple
    versions of a simulation (e.g., with different random seeds).

    Parameters

    • sims (Sim/list) – a single sim or a list of sims
    • base_sim (Sim) – the sim used for shared properties; if not supplied, the first of the sims
      provided
    • label (str) – the name of the multisim
    • initialize (bool) – whether or not to initialize the sims (otherwise, initialize them
      during run)
    • kwargs (dict) – stored in run_args and passed to run()

    Returns
        a MultiSim object

    Return type
        msim

Examples:

```python
sim = cv.Sim()  # Create the sim
msim = cv.MultiSim(sim, n_runs=5)  # Create the multisim
msim.run()  # Run them in parallel
msim.combine()  # Combine into one sim
msim.plot()  # Plot results

sim = cv.Sim()  # Create the sim
msim = cv.MultiSim(sim, n_runs=11, noise=0.1, keep_people=True)  # Set up a
covasim run with noise
msim.run()  # Run
msim.reduce()  # Compute statistics
msim.plot()  # Plot

sims = [cv.Sim(beta=0.015*(1+0.02*i)) for i in range(5)]  # Create sims
for sim in sims: sim.run()  # Run sims in serial
msim = cv.MultiSim(sims)  # Convert to multisim
msim.plot()  # Plot as single sim
```

result_keys()
    Attempt to retrieve the results keys from the base sim
init_sims(**kwargs)
Initialize the sims, but don’t actually run them. Syntax is the same as MultiSim.run(). Note: in most cases you can just call run() directly, there is no need to call this separately.

Parameters

**kwargs (dict) – passed to multi_run()

run(reduce=False, combine=False, **kwargs)
Run the actual sims

Parameters

• **reduce** (bool) – whether or not to reduce after running (see reduce())
• **combine** (bool) – whether or not to combine after running (see combine(), not compatible with reduce)
• **kwargs** (dict) – passed to multi_run(); use run_args to pass arguments to sim.run()

Returns
None (modifies MultiSim object in place)

Examples:

```python
msim.run()
msim.run(run_args=dict(until='2020-0601', restore_pars=False))
```

shrink(**kwargs)
Not to be confused with reduce(), this shrinks each sim in the msim; see sim.shrink() for more information.

Parameters

**kwargs (dict) – passed to sim.shrink() for each sim

reset()
Undo a combine() or reduce() by resetting the base sim, which, and results

reduce(quantiles=None, use_mean=False, bounds=None, output=False)
Combine multiple sims into a single sim statistically: by default, use the median value and the 10th and 90th percentiles for the lower and upper bounds. If use_mean=True, then use the mean and ±2 standard deviations for lower and upper bounds.

Parameters

• **quantiles** (dict) – the quantiles to use, e.g. [0.1, 0.9] or {'low': 0.1, 'high': 0.9}
• **use_mean** (bool) – whether to use the mean instead of the median
• **bounds** (float) – if use_mean=True, the multiplier on the standard deviation for upper and lower bounds (default 2)
• **output** (bool) – whether to return the “reduced” sim (in any case, modify the multisim in-place)

Example:

```python
msim = cv.MultiSim(cv.Sim())
msim.run()
msim.reduce()
msim.summarize()
```

mean(bounds=None, **kwargs)
Alias for reduce(use_mean=True). See reduce() for full description.

Parameters

• **bounds** (float) – multiplier on the standard deviation for the upper and lower bounds (default, 2)
• **kwargs (dict) – passed to reduce()

**median** (quantiles=None, **kwargs)
Alias for reduce(use_mean=False). See reduce() for full description.

**Parameters**

- **quantiles** (list or dict) – upper and lower quantiles (default, 0.1 and 0.9)
- **kwargs** (dict) – passed to reduce()

**combine** (output=False)
Combine multiple sims into a single sim with scaled results.

**Example**:
```python
cv = cv.MultiSim(cv.Sim())
cv.run()
cv.combine()
cv.summarize()
```

**compare** (t=None, sim inds=None, output=False, do_plot=False, **kwargs)
Create a dataframe compare sims at a single point in time.

**Parameters**

- **t** (int/str) – the day (or date) to do the comparison; default, the end
- **sim inds** (list) – list of integers of which sims to include (default: all)
- **output** (bool) – whether or not to return the comparison as a dataframe
- **do_plot** (bool) – whether or not to plot the comparison (see also plot_compare())
- **kwargs** (dict) – passed to plot_compare()

**Returns** a dataframe comparison

**Return type** df (dataframe)

**plot** (to_plot=None, inds=None, plot_sims=False, color_by_sim=None, max_sims=5, colors=None, labels=None, alpha_range=None, plot_args=None, show_args=None, **kwargs)
Plot all the sims – arguments passed to Sim.plot(). The behavior depends on whether or not combine() or reduce() has been called. If so, this function by default plots only the combined/reduced sim (which you can override with plot_sims=True). Otherwise, it plots a separate line for each sim.

Note that this function is complex because it aims to capture the flexibility of both sim.plot() and scens.plot(). By default, if combine() or reduce() has been used, it will resemble sim.plot(); otherwise, it will resemble scens.plot(). This can be changed via color_by_sim, together with the other options.

**Parameters**

- **to_plot** (list) – list or dict of which results to plot; see cv.get_default_plots() for structure
- **inds** (list) – if not combined or reduced, the indices of the simulations to plot (if None, plot all)
- **plot_sims** (bool) – whether to plot individual sims, even if combine() or reduce() has been used
- **color_by_sim** (bool) – if True, set colors based on the simulation type; otherwise, color by result type; True implies a scenario-style plotting, False implies sim-style plotting
- **max_sims** (int) – maximum number of sims to use with color-by-sim; can be overridden by other options
• **colors** *(list)* – if supplied, override default colors for `color_by_sim`

• **labels** *(list)* – if supplied, override default labels for `color_by_sim`

• **alpha_range** *(list)* – a 2-element list/tuple/array providing the range of alpha values to use to distinguish the lines

• **plot_args** *(dict)* – passed to `sim.plot()`

• **show_args** *(dict)* – passed to `sim.plot()`

• **kwargs** *(dict)* – passed to `sim.plot()`

**Returns**  Figure handle

**Return type**  `fig`

**Examples:**

```python
sim = cv.Sim()
msim = cv.MultiSim(sim)
msim.run()
msim.plot()  # Plots individual sims
msim.reduce()
msim.plot()  # Plots the combined sim
```

**plot_result** *(key, colors=None, labels=None, *args, **kwargs)*

Convenience method for plotting – arguments passed to `sim.plot_result()`

**plot_compare** *(t=-1, sim inds=None, log_scale=True, **kwargs)*

Plot a comparison between sims, using bars to show different values for each result. For an explanation of other available arguments, see `Sim.plot()`.

**Parameters**

• **t** *(int)* – index of results, passed to `compare()`

• **sim inds** *(list)* – which sims to include, passed to `compare()`

• **log_scale** *(bool)* – whether to plot with a logarithmic x-axis

• **kwargs** *(dict)* – standard plotting arguments, see `Sim.plot()` for explanation

**Returns**  Figure handle

**Return type**  `fig`

**save** *(filename=None, keep_people=False, **kwargs)*

Save to disk as a gzipped pickle. Load with `cv.load(filename)` or `cv.MultiSim.load(filename)`.

**Parameters**

• **filename** *(str)* – the name or path of the file to save to; if None, uses default

• **keep_people** *(bool)* – whether or not to store the population in the Sim objects (NB, very large)

• **kwargs** *(dict)* – passed to `sc.makefilepath()`

**Returns**  the validated absolute path to the saved file

**Return type**  `str`

**Example:**

```python
msim.save()  # Saves to an .msim file
```
static load (msimfile, *args, **kwargs)
Load from disk from a gzipped pickle.

Parameters

- **msimfile** (**str**): the name or path of the file to load from
- **kwargs**: passed to cv.load()

Returns the loaded MultiSim object

Return type: msim (**MultiSim**)

Example:

```python
msim = cv.MultiSim.load('my-multisim.msim')
```

static merge (*args, base=False)
Convenience method for merging two MultiSim objects.

Parameters

- **args** (**MultiSim**): the MultiSims to merge (either a list, or separate)
- **base** (**bool**): if True, make a new list of sims from the multisim’s two base sims; otherwise, merge the multisim’s lists of sims

Returns a new MultiSim object

Return type: msim (**MultiSim**)

Examples:

```python
mm1 = cv.MultiSim.merge(msim1, msim2, base=True) mm2 = cv.MultiSim.merge([m1, m2, m3, m4], base=False)
```

split (inds=None, chunks=None)
Convenience method for splitting one MultiSim into several. You can specify either individual indices of simulations to extract, via inds, or consecutive chunks of indices, via chunks. If this function is called on a merged MultiSim, the chunks can be retrieved automatically and no arguments are necessary.

Parameters

- **inds** (**list**): a list of lists of indices, with each list turned into a MultiSim
- **chunks** (**int or list**): if an int, split the MultiSim into chunks of that length; if a list return chunks of that many sims

Returns A list of MultiSim objects

Examples:

```python
m1 = cv.MultiSim(cv.Sim(label='sim1'), initialize=True)
m2 = cv.MultiSim(cv.Sim(label='sim2'), initialize=True)
m3 = cv.MultiSim.merge(m1, m2)
m3.run()
m1b, m2b = m3.split()
msim = cv.MultiSim(cv.Sim(), n_runs=6)
msim.run()
m1, m2 = msim.split(inds=[[0, 2, 4], [1, 3, 5]])
mlist1 = msim.split(chunks=[2, 4])  # Equivalent to inds=[[0, 1], [2, 3, 4, 5]]
mlist2 = msim.split(chunks=3)  # Equivalent to inds=[[0, 1, 2], [3, 4, 5]]
```
disp (output=False)
Display a verbose description of a multisim. See also multisim.summarize() (medium length output) and multisim.brief() (short output).

Parameters output (bool) – if true, return a string instead of printing output

Example:
```python
msim = cv.MultiSim(cv.Sim(verbose=0), label='Example multisim')
msim.run()
msim.disp()  # Displays detailed output
```

summarize (output=False)
Print a moderate length summary of the MultiSim. See also multisim.disp() (detailed output) and multisim.brief() (short output).

Parameters output (bool) – if true, return a string instead of printing output

Example:
```python
msim = cv.MultiSim(cv.Sim(verbose=0), label='Example multisim')
msim.run()
msim.summarize()  # Prints moderate length output
```

brief (output=False)
Print a compact representation of the multisim. See also multisim.disp() (detailed output) and multisim.summarize() (medium length output).

Parameters output (bool) – if true, return a string instead of printing output

Example:
```python
msim = cv.MultiSim(cv.Sim(verbose=0), label='Example multisim')
msim.run()
msim.brief()  # Prints one-line output
```

to_json (*args, **kwargs)
Shortcut for base_sim.to_json()

to_excel (*args, **kwargs)
Shortcut for base_sim.to_excel()

class Scenarios (sim=None, metapars=None, scenarios=None, basepars=None, scenfile=None, label=None)
Bases: covasim.base.ParsObj

Class for running multiple sets of multiple simulations – e.g., scenarios. Note that most users are recommended to use MultiSim rather than Scenarios, as it gives more control over run options. Scenarios should be used primarily for quick investigations. See the examples folder for example usage.

Parameters
- sim (Sim) – if supplied, use a pre-created simulation as the basis for the scenarios
- metapars (dict) – meta-parameters for the run, e.g. number of runs; see make_metapars() for structure
- scenarios (dict) – a dictionary defining the scenarios; see examples folder for examples; see below for baseline
- basepars (dict) – a dictionary of sim parameters to be used for the basis of the scenarios (not required if sim is provided)
- **scenfile**(str) – a filename for saving (defaults to the creation date)
- **label**(str) – the name of the scenarios

**Example:**

```python
scens = cv.Scenarios()
```

**Returns** a Scenarios object

**Return type** scens

**result_keys**(which='all')

Attempt to retrieve the results keys from the base sim

**run**(debug=False, keep_people=False, verbose=None, **kwargs)

Run the specified scenarios.

**Parameters**

- **debug**(bool) – if True, runs a single run instead of multiple, which makes debugging easier
- **verbose**(int) – level of detail to print, passed to sim.run()
- **kwargs**(dict) – passed to multi_run() and thence to sim.run()

**Returns** None (modifies Scenarios object in place)

**compare**(t=None, output=False)

Print out a comparison of each scenario.

**Parameters**

- **t**(int/str) – the day (or date) to do the comparison; default, the end
- **output**(bool) – if true, return the dataframe instead of printing output

**Example:**

```python
scenarios = {'base': {'name':'Base', 'pars': {}}, 'beta': {'name': 'Beta', 'pars →': {'beta': 0.020}}
scens = cv.Scenarios(scenarios=scenarios, label='Example scenarios')
scens.run()
scens.compare(t=30) # Prints comparison for day 30
```

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

Plot the results of a scenario. For an explanation of available arguments, see Sim.plot().

**Returns** Figure handle

**Return type** fig

**Example:**

```python
scens = cv.Scenarios()
scens.run()
scens.plot()
```

**to_json**(filename=None, tostring=True, indent=2, verbose=False, *args, **kwargs)

Export results as JSON.

**Parameters** **filename**(str) – if None, return string; else, write to file
**Returns** A unicode string containing a JSON representation of the results, or writes the JSON
file to disk

**to_excel** *(filename=\texttt{None})*
Export results as XLSX

**Parameters**
- **filename** *(str)* – if \texttt{None}, return string; else, write to file

**Returns** An \texttt{sc.Spreadsheet} with an Excel file, or writes the file to disk

**save** *(\texttt{scenfile}=\texttt{None}, \texttt{keep_sims}=\texttt{True}, \texttt{keep_people}=\texttt{False}, **\texttt{kwargs})*
Save to disk as a gzipped pickle.

**Parameters**
- **\texttt{scenfile}** *(str)* – the name or path of the file to save to; if \texttt{None}, uses stored
- **\texttt{keep_sims}** *(bool)* – whether or not to store the actual Sim objects in the Scenarios
  object
- **\texttt{keep_people}** *(bool)* – whether or not to store the population in the Sim objects (NB,
  very large)
- **\texttt{kwargs}** *(dict)* – passed to \texttt{makefilepath()}

**Returns** the validated absolute path to the saved file

**Return type** \texttt{scenfile} *(str)*

**Example:**

```python
scens.save()  # Saves to a .scens file with the date and time of creation by default
```

**static load** *(\texttt{scenfile}, *\texttt{args}, **\texttt{kwargs})*
Load from disk from a gzipped pickle.

**Parameters**
- **\texttt{scenfile}** *(str)* – the name or path of the file to load from
- **\texttt{kwargs}** – passed to \texttt{cv.load()}

**Returns** the loaded scenarios object

**Return type** \texttt{scens} *(Scenarios)*

**Example:**

```python
scens = cv.Scenarios.load('my-scenarios.scens')
```

**disp** *(output=\texttt{False})*
Display a verbose description of the scenarios. See also \texttt{scenarios.summarize()} (medium length output)
and \texttt{scenarios.brief()} (short output).

**Parameters**
- **output** *(bool)* – if \texttt{true}, return a string instead of printing output

**Example:**

```python
scens = cv.Scenarios(cv.Sim(), label='Example scenarios')
scens.run(\texttt{verbose}=0)  # Run silently
scens.disp()  # Displays detailed output
```
**summarize** (*output=False*)
Print a moderate length summary of the scenarios. See also scenarios.disp() (detailed output) and scenarios.brief() (short output).

Parameters **output** (*bool*) – if true, return a string instead of printing output

Example:

```python
cens = cv.Scenarios(cv.Sim(), label='Example scenarios')
cens.run(verbose=0)  # Run silently
cens.summarize()  # Prints moderate length output
```

**brief** (*output=False*)
Print a compact representation of the scenarios. See also scenarios.disp() (detailed output) and scenarios.summarize() (medium length output).

Parameters **output** (*bool*) – if true, return a string instead of printing output

Example:

```python
cens = cv.Scenarios(label='Example scenarios')
cens.run()
cens.brief()  # Prints one-line output
```

**single_run** (*sim, ind=0, reseed=True, noise=0.0, noisepar=None, keep_people=False, run_args=None, sim_args=None, verbose=None, do_run=True, **kwargs*)
Convenience function to perform a single simulation run. Mostly used for parallelization, but can also be used directly.

Parameters

- **sim** (*Sim*) – the sim instance to be run
- **ind** (*int*) – the index of this sim
- **reseed** (*bool*) – whether or not to generate a fresh seed for each run
- **noise** (*float*) – the amount of noise to add to each run
- **noisepar** (*str*) – the name of the parameter to add noise to
- **keep_people** (*bool*) – whether to keep the people after the sim run
- **run_args** (*dict*) – arguments passed to sim.run()
- **sim_args** (*dict*) – extra parameters to pass to the sim, e.g. ‘n_infected’
- **verbose** (*int*) – detail to print
- **do_run** (*bool*) – whether to actually run the sim (if not, just initialize it)
- **kwargs** (*dict*) – also passed to the sim

Returns a single sim object with results

Return type **sim** (*Sim*)

Example:

```python
import covasim as cv
sim = cv.Sim()  # Create a default simulation
sim = cv.single_run(sim)  # Run it, equivalent(ish) to sim.run()
```
multi_run(sim, n_runs=4, reseed=True, noise=0.0, noisepar=None, iterpars=None, combine=False, keep_people=None, run_args=None, sim_args=None, par_args=None, do_run=True, parallel=True, n_cpus=None, verbose=None, **kwargs)

For running multiple runs in parallel. If the first argument is a list of sims, exactly these will be run and most other arguments will be ignored.

Parameters

- `sim (Sim)` – the sim instance to be run, or a list of sims.
- `n_runs (int)` – the number of parallel runs
- `reseed (bool)` – whether or not to generate a fresh seed for each run
- `noise (float)` – the amount of noise to add to each run
- `noisepar (str)` – the name of the parameter to add noise to
- `iterpars (dict)` – any other parameters to iterate over the runs; see sc.parallelize() for syntax
- `combine (bool)` – whether or not to combine all results into one sim, rather than return multiple sim objects
- `keep_people (bool)` – whether to keep the people after the sim run (default false)
- `run_args (dict)` – arguments passed to sim.run()
- `sim_args (dict)` – extra parameters to pass to the sim
- `par_args (dict)` – arguments passed to sc.parallelize()
- `do_run (bool)` – whether to actually run the sim (if not, just initialize it)
- `parallel (bool)` – whether to run in parallel using multiprocessing (else, just run in a loop)
- `n_cpus (int)` – the number of CPUs to run on (if blank, set automatically; otherwise, passed to par_args)
- `verbose (int)` – detail to print
- `kwargs (dict)` – also passed to the sim

Returns

If combine is True, a single sim object with the combined results from each sim. Otherwise, a list of sim objects (default).

Example:

```python
import covasim as cv
sim = cv.Sim()
sims = cv.multi_run(sim, n_runs=6, noise=0.2)
```

parallel(*args, **kwargs)

A shortcut to `cv.MultiSim()`, allowing the quick running of multiple simulations at once.

Parameters

- `args (list)` – The simulations to run
- `kwargs (dict)` – passed to multi_run()

Returns

A run MultiSim object.

Examples:
s1 = cv.Sim(beta=0.01, label='Low')
s2 = cv.Sim(beta=0.02, label='High')
cv.parallel(s1, s2).plot()
msim = cv.parallel([s1, s2], keep_people=True)

New in version 3.1.1.

### 1.10.2.13 covasim.settings module

Define options for Covasim, mostly plotting and Numba options. All options should be set using set(), e.g.:

```python
cv.options.set(font_size=18)
```

To reset default options, use:

```python
cv.options.set('default')
```

### 1.10.2.14 covasim.sim module

Defines the Sim class, Covasim’s core class.

```python
class Sim(pars=None, datafile=None, datacols=None, label=None, simfile=None, popfile=None,
load_pop=False, save_pop=False, version=None, **kwargs)
```

The Sim class handles the running of the simulation: the creation of the population and the dynamics of the epidemic. This class handles the mechanics of the actual simulation, while BaseSim takes care of housekeeping (saving, loading, exporting, etc.). Please see the BaseSim class for additional methods.

**Parameters**

- **pars** (dict) – parameters to modify from their default values
- **datafile** (str/df) – filename of (Excel, CSV) data file to load, or a pandas dataframe of the data
- **datacols** (list) – list of column names of the data to load
- **label** (str) – the name of the simulation (useful to distinguish in batch runs)
- **simfile** (str) – the filename for this simulation, if it’s saved (default: creation date)
- **popfile** (str) – the filename to load/save the population for this simulation
- **load_pop** (bool) – whether to load the population from the named file
- **save_pop** (bool) – whether to save the population to the named file
- **version** (str) – if supplied, use default parameters from this version of Covasim instead of the latest
- **kwargs** (dict) – passed to make_pars()

**Examples:**

```python
sim = cv.Sim()
sim = cv.Sim(pop_size=10e3, datafile='my_data.xlsx')
```

**load_data** (datafile=None, datacols=None, verbose=None, **kwargs)

Load the data to calibrate against, if provided
**initialize** *(reset=False, **kwargs)*

Perform all initializations, including validating the parameters, setting the random number seed, creating the results structure, initializing the people, validating the layer parameters (which requires the people), and initializing the interventions.

**Parameters**

- **reset** *(bool)* – whether or not to reset people even if they already exist
- **kwargs** *(dict)* – passed to init_people

**layer_keys** *

Attempt to retrieve the current layer keys, in the following order: from the people object (for an initialized sim), from the popdict (for one in the process of being initialized), from the beta_layer parameter (for an uninitialized sim), or by assuming a default (if none of the above are available).

**reset_layer_pars** *(layer_keys=None, force=False)*

Reset the parameters to match the population.

**Parameters**

- **layer_keys** *(list)* – override the default layer keys (use stored keys by default)
- **force** *(bool)* – reset the parameters even if they already exist

**validate_layer_pars** *

Handle layer parameters, since they need to be validated after the population creation, rather than before.

**validate_pars** *(validate_layers=True)*

Some parameters can take multiple types; this makes them consistent.

**Parameters**

- **validate_layers** *(bool)* – whether to validate layer parameters as well via validate_layer_pars() – usually yes, except during initialization

**init_results** *

Create the main results structure. We differentiate between flows, stocks, and cumulative results The prefix “new” is used for flow variables, i.e. counting new events (infections/deaths/recoveries) on each timestep The prefix “n” is used for stock variables, i.e. counting the total number in any given state (sus/inf/rec/etc) on any particular timestep The prefix “cum” is used for cumulative variables, i.e. counting the total number that have ever been in a given state at some point in the sim Note that, by definition, n_dead is the same as cum_deaths and n_recovered is the same as cum_recoveries, so we only define the cumulative versions

**load_population** *(popfile=None, **kwargs)*

Load the population dictionary from file – typically done automatically as part of sim.initialize(). Supports loading either saved population dictionaries (popdicts, file ending .pop by convention), or ready-to-go People objects (file ending .ppl by convention). Either object an also be supplied directly. Once a population file is loaded, it is removed from the Sim object.

**Parameters**

- **popfile** *(str or obj)* – if a string, name of the file; otherwise, the popdict or People object to load
- **kwargs** *(dict)* – passed to sc.makefilepath()

**init_people** *(save_pop=False, load_pop=False, popfile=None, reset=False, verbose=None, **kwargs)*

Create the people.

**Parameters**

- **save_pop** *(bool)* – if true, save the population dictionary to popfile
- **load_pop** *(bool)* – if true, load the population dictionary from popfile
Covasim, Release 3.1.1

- **popfile** *(str)* – filename to load/save the population
- **reset** *(bool)* – whether to regenerate the people even if they already exist
- **verbose** *(int)* – detail to print
- **kwargs** *(dict)* – passed to cv.make_people()

**init_interventions()**
Initialize and validate the interventions

**finalize_interventions()**

**init_analyzers()**
Initialize the analyzers

**finalize_analyzers()**

**init_variants()**
Initialize the variants

**init_immunity**(create=False)
Initialize immunity matrices and precompute nab waning for each variant

**rescale()**
Dynamically rescale the population – used during step()

**step()**
Step the simulation forward in time. Usually, the user would use sim.run() rather than calling sim.step() directly.

**run**(do_plot=False, until=None, restore_pars=True, reset_seed=True, verbose=None)
Run the simulation.

**Parameters**
- **do_plot**(bool) – whether to plot
- **until**(int/str) – day or date to run until
- **restore_pars**(bool) – whether to make a copy of the parameters before the run and restore it after, so runs are repeatable
- **reset_seed**(bool) – whether to reset the random number stream immediately before run
- **verbose**(float) – level of detail to print, e.g. -1 = one-line output, 0 = no output, 0.1 = print every 10th day, 1 = print every day

**Returns** A pointer to the sim object (with results modified in-place)

**finalize**(verbose=None, restore_pars=True)
Compute final results

**compute_results**(verbose=None)
Perform final calculations on the results

**compute_states()**
Compute prevalence, incidence, and other states. Prevalence is the current number of infected people divided by the number of people who are alive. Incidence is the number of new infections per day divided by the susceptible population. Also calculates the number of people alive, the number preinfectious, the number removed, and recalculates susceptibles to handle scaling.

**compute_yield()**
Compute test yield – number of positive tests divided by the total number of tests, also called test positivity
rate. Relative yield is with respect to prevalence: i.e., how the yield compares to what the yield would be from choosing a person at random from the population.

**compute_doubling** *(window=3, max_doubling_time=30)*

Calculate doubling time using exponential approximation – a more detailed approach is in utils.py. Compares infections at time t to infections at time t-window, and uses that to compute the doubling time. For example, if there are 100 cumulative infections on day 12 and 200 infections on day 19, doubling time is 7 days.

**Parameters**

- **window** *(float)* – the size of the window used (larger values are more accurate but less precise)
- **max_doubling_time** *(float)* – doubling time could be infinite, so this places a bound on it

**Returns** the doubling time results array

**Return type** doubling_time (array)

**compute_r_eff** *(method='daily', smoothing=2, window=7)*

Effective reproduction number based on number of people each person infected.

**Parameters**

- **method** *(str)* – ‘daily’ uses daily infections, ‘infectious’ counts from the date infectious, ‘outcome’ counts from the date recovered/dead
- **smoothing** *(int)* – the number of steps to smooth over for the ‘daily’ method
- **window** *(int)* – the size of the window used for ‘infectious’ and ‘outcome’ calculations (larger values are more accurate but less precise)

**Returns** the r_eff results array

**Return type** r_eff (array)

**compute_gen_time** ()

Calculate the generation time (or serial interval). There are two ways to do this calculation. The ‘true’ interval (exposure time to exposure time) or ‘clinical’ (symptom onset to symptom onset).

**Returns** the generation time results

**Return type** gen_time (dict)

**compute_summary** *(full=None, t=None, update=True, output=False, require_run=False)*

Compute the summary dict and string for the sim. Used internally; see sim.summarize() for the user version.

**Parameters**

- **full** *(bool)* – whether or not to print all results (by default, only cumulative)
- **t** *(int/str)* – day or date to compute summary for (by default, the last point)
- **update** *(bool)* – whether to update the stored sim.summary
- **output** *(bool)* – whether to return the summary
- **require_run** *(bool)* – whether to raise an exception if simulations have not been run yet

**summarize** *(full=False, t=None, sep=None, output=False)*

Print a medium-length summary of the simulation, drawing from the last time point in the simulation by
default. Called by default at the end of a sim run. See also sim.disp() (detailed output) and sim.brief() (short output).

Parameters

- **full** *(bool)* – whether or not to print all results (by default, only cumulative)
- **t** *(int/str)* – day or date to compute summary for (by default, the last point)
- **sep** *(str)* – thousands separator (default ',')
- **output** *(bool)* – whether to return the summary instead of printing it

Examples:

```python
sim = cv.Sim(label='Example sim', verbose=0) # Set to run silently
sim.run() # Run the sim
sim.summarize() # Print medium-length summary of the sim
sim.summarize(t=24, full=True) # Print a "slice" of all sim results on day 24
```

disp *(output=False)*

Display a verbose description of a sim. See also sim.summarize() (medium length output) and sim.brief() (short output).

Parameters **output** *(bool)* – if true, return a string instead of printing output

Example:

```python
sim = cv.Sim(label='Example sim', verbose=0) # Set to run silently
sim.run() # Run the sim
sim.disp() # Displays detailed output
```

brief *(output=False)*

Print a one-line description of a sim. See also sim.disp() (detailed output) and sim.summarize() (medium length output). The symbol "" is used to show infections, and "" is used to show deaths.

Parameters **output** *(bool)* – if true, return a string instead of printing output

Example:

```python
sim = cv.Sim(label='Example sim', verbose=0) # Set to run silently
sim.run() # Run the sim
sim.brief() # Prints one-line output
```

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

Compute the fit between the model and the data. See cv.Fit() for more information.

Parameters

- **args** *(list)* – passed to cv.Fit()
- **kwargs** *(dict)* – passed to cv.Fit()

Returns A Fit object

Example:

```python
sim = cv.Sim(datafile='data.csv')
sim.run()
fit = sim.compute_fit()
fit.plot()
```
**calibrate** (*calib_pars, **kwargs*)
Automatically calibrate the simulation, returning a Calibration object (a type of analyzer). See the documentation on that class for more information.

**Parameters**
- *calib_pars* *(dict)* – a dictionary of the parameters to calibrate of the format
dict(key1=[best, low, high])
- *kwargs* *(dict)* – passed to cv.Calibration()

**Returns** A Calibration object

**Example**:
```python
sim = cv.Sim(datafile='data.csv')
calib_pars = dict(beta=[0.015, 0.010, 0.020])
calib = sim.calibrate(calib_pars, n_trials=50)
calib.plot()
```

**make_age_histogram** (*args, output=True, **kwargs*)
Calculate the age histograms of infections, deaths, diagnoses, etc. See cv.age_histogram() for more information. This can be used alternatively to supplying the age histogram as an analyzer to the sim. If used this way, it can only record the final time point since the states of each person are not saved during the sim.

**Parameters**
- *output* *(bool)* – whether or not to return the age histogram; if not, store in sim.results
- *args* *(list)* – passed to cv.age_histogram()
- *kwargs* *(dict)* – passed to cv.age_histogram()

**Example**:
```python
sim = cv.Sim()
sim.run()
agehist = sim.make_age_histogram()
agehist.plot()
```

**make_transtree** (*args, output=True, **kwargs*)
Create a TransTree (transmission tree) object, for analyzing the pattern of transmissions in the simulation. See cv.TransTree() for more information.

**Parameters**
- *output* *(bool)* – whether or not to return the TransTree; if not, store in sim.results
- *args* *(list)* – passed to cv.TransTree()
- *kwargs* *(dict)* – passed to cv.TransTree()

**Example**:
```python
sim = cv.Sim()
sim.run()
tt = sim.make_transtree()
```

**plot** (*args, **kwargs*)
Plot the results of a single simulation.

**Parameters**
- *to_plot* *(dict)* – Dict of results to plot; see get_default_plots() for structure
• **do_save**(bool) – Whether or not to save the figure

• **fig_path**(str) – Path to save the figure

• **fig_args**(dict) – Dictionary of kwargs to be passed to pl.figure()

• **plot_args**(dict) – Dictionary of kwargs to be passed to pl.plot()

• **scatter_args**(dict) – Dictionary of kwargs to be passed to pl.scatter()

• **axis_args**(dict) – Dictionary of kwargs to be passed to pl.subplots_adjust()

• **legend_args**(dict) – Dictionary of kwargs to be passed to pl.legend(); if show_legend=False, do not show

• **date_args**(dict) – Control how the x-axis (dates) are shown (see below for explanation)

• **show_args**(dict) – Control which “extras” get shown: uncertainty bounds, data, interventions, ticks, and the legend

• **mpl_args**(dict) – Dictionary of kwargs to be passed to Matplotlib; options are dpi, fontsize, and fontfamily

• **as_dates**(bool) – Whether to plot the x-axis as dates or time points

• **dateformat**(str) – Date string format, e.g. ‘%B %d’

• **interval**(int) – Interval between tick marks

• **n_cols**(int) – Number of columns of subpanels to use for subplot

• **font_size**(int) – Size of the font

• **font_family**(str) – Font face

• **grid**(bool) – Whether or not to plot gridlines

• **commaticks**(bool) – Plot y-axis with commas rather than scientific notation

• **setylim**(bool) – Reset the y limit to start at 0

• **log_scale**(bool) – Whether or not to plot the y-axis with a log scale; if a list, panels to show as log

• **do_show**(bool) – Whether or not to show the figure

• **colors**(dict) – Custom color for each result, must be a dictionary with one entry per result key in to_plot

• **sep_figs**(bool) – Whether to show separate figures for different results instead of subplots

• **fig**(fig) – Handle of existing figure to plot into

• **ax**(axes) – Axes instance to plot into

• **kwargs**(dict) – Parsed among figure, plot, scatter, date, and other settings (will raise an error if not recognized)

The optional dictionary “date_args” allows several settings for controlling how the x-axis of plots are shown, if this axis is dates. These options are:

• **as_dates**: whether to format them as dates (else, format them as days since the start)

• **dateformat**: string format for the date (default %b-%d, e.g. Apr-04)

• **interval**: the number of days between tick marks
• rotation: whether to rotate labels
• start_day: the first day to plot
• end_day: the last day to plot

Returns  Figure handle
Return type  fig

Example:
```python
sim = cv.Sim()
sim.run()
sim.plot()
```

New in version 2.1.0: argument passing, date_args, and mpl_args

plot_result (key, *args, **kwargs)
Simple method to plot a single result. Useful for results that aren’t standard outputs. See sim.plot() for explanation of other arguments.

Parameters  key (str) – the key of the result to plot

Returns  Figure handle
Return type  fig

Example:
```python
sim = cv.Sim().run()
sim.plot_result('r_eff')
```

diff_sims (sim1, sim2, skip_key_diffs=False, skip=None, output=False, die=False)
Compute the difference of the summaries of two simulations, and print any values which differ.

Parameters
• sim1 (sim/dict) – either a simulation object or the sim.summary dictionary
• sim2 (sim/dict) – ditto
• skip_key_diffs (bool) – whether to skip keys that don’t match between sims
• skip (list) – a list of values to skip
• output (bool) – whether to return the output as a string (otherwise print)
• die (bool) – whether to raise an exception if the sims don’t match
• require_run (bool) – require that the simulations have been run

Example:
```python
s1 = cv.Sim(beta=0.01)
s2 = cv.Sim(beta=0.02)
s1.run()
s2.run()
cv.diff_sims(s1, s2)
```

demo (preset=None, to_plot=None, scens=None, run_args=None, plot_args=None, **kwargs)
Shortcut for cv.Sim().run().plot().

Parameters
• **preset** *(str)* – use a preset run configuration; currently the only option is “full”
• **to_plot** *(str)* – what to plot
• **scens** *(dict)* – dictionary of scenarios to run as a multisim, if preset=’full’
• **kwargs** *(dict)* – passed to Sim()
• **run_args** *(dict)* – passed to sim.run()
• **plot_args** *(dict)* – passed to sim.plot()

**Examples:**

```python
cv.demo()  # Simplest example
cv.demo('full')  # Full example
cv.demo('full', overview=True)  # Plot all results
cv.demo(beta=0.020, run_args={'verbose':0}, plot_args={'to_plot': 'overview'})  # Pass in custom values
```

**exception** AlreadyRunError  
**Bases:** RuntimeError

This error is raised if a simulation is run in such a way that no timesteps will be taken. This error is a distinct type so that it can be safely caught and ignored if required, but it is anticipated that most of the time, calling sim.run() and not taking any timesteps, would be an inadvertent error.

### 1.10.2.15 covasim.utils module

Numerical utilities for running Covasim.  
These include the viral load, transmissibility, and infection calculations at the heart of the integration loop.

**sample** *(dist=None, par1=None, par2=None, size=None, **kwargs)*  
Draw a sample from the distribution specified by the input. The available distributions are:

- ‘uniform’ : uniform distribution from low=par1 to high=par2; mean is equal to (par1+par2)/2
- ‘normal’ : normal distribution with mean=par1 and std=par2
- ‘lognormal’ : lognormal distribution with mean=par1 and std=par2 (parameters are for the lognormal distribution, not the underlying normal distribution)
- ‘normal_pos’ : right-sided normal distribution (i.e. only positive values), with mean=par1 and std=par2 of the underlying normal distribution
- ‘normal_int’ : normal distribution with mean=par1 and std=par2, returns only integer values
- ‘lognormal_int’ : lognormal distribution with mean=par1 and std=par2, returns only integer values
- ‘poisson’ : Poisson distribution with rate=par1 (par2 is not used); mean and variance are equal to par1
- ‘neg_binomial’ : negative binomial distribution with mean=par1 and k=par2; converges to Poisson with k=∞

**Parameters**

- **dist** *(str)* – the distribution to sample from
- **par1** *(float)* – the “main” distribution parameter (e.g. mean)
- **par2** *(float)* – the “secondary” distribution parameter (e.g. std)
- **size** *(int)* – the number of samples (default=1)
**kwargs (dict) – passed to individual sampling functions

**Returns** A length N array of samples

**Examples:**

```python
cv.sample()  # returns Unif(0,1)
cv.sample(dist='normal', par1=3, par2=0.5)  # returns Normal(μ=3, σ=0.5)
cv.sample(dist='lognormal_int', par1=5, par2=3)  # returns a lognormally distributed set of values with mean 5 and std 3
```

**Notes**

Lognormal distributions are parameterized with reference to the underlying normal distribution (see: https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.random.lognormal.html), but this function assumes the user wants to specify the mean and std of the lognormal distribution.

Negative binomial distributions are parameterized with reference to the mean and dispersion parameter k (see: https://en.wikipedia.org/wiki/Negative_binomial_distribution). The r parameter of the underlying distribution is then calculated from the desired mean and k. For a small mean (~1), a dispersion parameter of ∞ corresponds to the variance and standard deviation being equal to the mean (i.e., Poisson). For a large mean (e.g. >100), a dispersion parameter of 1 corresponds to the standard deviation being equal to the mean.

**get_pdf** (dist=None, par1=None, par2=None)

Return a probability density function for the specified distribution. This is used for example by test_num to retrieve the distribution of times from symptom-to-swab for testing. For example, for Washington State, these values are dist='lognormal', par1=10, par2=170.

**set_seed** (seed=None)

Reset the random seed – complicated because of Numba, which requires special syntax to reset the seed. This function also resets Python’s built-in random number generated.

**n_binomial** (prob, n)

Perform multiple binomial (Bernoulli) trials

**Parameters**

- prob (float) – probability of each trial succeeding
- n (int) – number of trials (size of array)

**Returns** Boolean array of which trials succeeded

**Example:**

```python
outcomes = cv.n_binomial(0.5, 100)  # Perform 100 coin-flips
```

**binomial_filter** (prob, arr)

Binomial “filter” – the same as n_binomial, except return the elements of arr that succeeded.

**Parameters**

- prob (float) – probability of each trial succeeding
- arr (array) – the array to be filtered

**Returns** Subset of array for which trials succeeded

**Example:**
inds = cv.binomial_filter(0.5, np.arange(20)**2) # Return which values out of the (arbitrary) array passed the coin flip

**binomial_arr** *(prob_arr)*

Binomial (Bernoulli) trials each with different probabilities.

**Parameters**

prob_arr *(array)* – array of probabilities

**Returns**

Boolean array of which trials on the input array succeeded

**Example:**

```
outcomes = cv.binomial_arr([0.1, 0.1, 0.2, 0.2, 0.8, 0.8])  # Perform 6 trials with different probabilities
```

**n_multinomial** *(probs, n)*

An array of multinomial trials.

**Parameters**

- probs *(array)* – probability of each outcome, which usually should sum to 1
- n *(int)* – number of trials

**Returns**

Array of integer outcomes

**Example:**

```
outcomes = cv.multinomial(np.ones(6)/6.0, 50)+1  # Return 50 die-rolls
```

**poisson**

A Poisson trial.

**Parameters**

rate *(float)* – the rate of the Poisson process

**Example:**

```
outcome = cv.poisson(100)  # Single Poisson trial with mean 100
```

**n_poisson**

An array of Poisson trials.

**Parameters**

- rate *(float)* – the rate of the process (mean)
- n *(int)* – number of trials

**Example:**

```
outcomes = cv.n_poisson(100, 20)  # 20 Poisson trials with mean 100
```

**n_neg_binomial** *(rate, dispersion, n, step=1)*

An array of negative binomial trials. See cv.sample() for more explanation.

**Parameters**

- rate *(float)* – the rate of the process (mean, same as Poisson)
- dispersion *(float)* – dispersion parameter; lower is more dispersion, i.e. 0 = infinite, \( \infty \) = Poisson
- n *(int)* – number of trials
• **step** (*float*) – the step size to use if non-integer outputs are desired

Example:

```python
outcomes = cv.n_neg_binomial(100, 1, 50)  # 50 negative binomial trials with mean -> 100 and dispersion roughly equal to mean (large-mean limit)
outcomes = cv.n_neg_binomial(1, 100, 20)  # 20 negative binomial trials with mean -> 1 and dispersion still roughly equal to mean (approximately Poisson)
```

**choose**

Choose a subset of items (e.g., people) without replacement.

**Parameters**

- **max_n** (*int*) – the total number of items
- **n** (*int*) – the number of items to choose

**Example:**

```python
choices = cv.choose(5, 2)  # choose 2 out of 5 people with equal probability (without repeats)
```

**choose_r**

Choose a subset of items (e.g., people), with replacement.

**Parameters**

- **max_n** (*int*) – the total number of items
- **n** (*int*) – the number of items to choose

**Example:**

```python
choices = cv.choose_r(5, 10)  # choose 10 out of 5 people with equal probability (with repeats)
```

**choose_w** (*probs*, **n**, **unique=True**)  
Choose n items (e.g. people), each with a probability from the distribution `probs`.

**Parameters**

- **probs** (*array*) – list of probabilities, should sum to 1
- **n** (*int*) – number of samples to choose
- **unique** (*bool*) – whether or not to ensure unique indices

**Example:**

```python
choices = cv.choose_w([0.2, 0.5, 0.1, 0.1, 0.1], 2)  # choose 2 out of 5 people with nonequal probability.
```

**true** (*arr*)

Returns the indices of the values of the array that are true: just an alias for `arr.nonzero()[0]`.

**Parameters**  
**arr** (*array*) – any array

**Example:**

```python
inds = cv.true(np.array([1, 0, 0, 1, 1, 0, 1]))  # Returns array([0, 3, 4, 6])
```

**false** (*arr*)

Returns the indices of the values of the array that are false.
Parameters **arr** *(array)* – any array

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| ```python
inds = cv.false(np.array([1,0,0,1,0,1]))
``` |

**defined**(arr)

Returns the indices of the values of the array that are not-nan.

Parameters **arr** *(array)* – any array

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| ```python
inds = cv.defined(np.array([1,np.nan,0,np.nan,1,0,1]))
``` |

**undefined**(arr)

Returns the indices of the values of the array that are not-nan.

Parameters **arr** *(array)* – any array

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| ```python
inds = cv.defined(np.array([1,np.nan,0,np.nan,1,0,1]))
``` |

**itrue**(arr, inds)

Returns the indices that are true in the array – name is short for indices[true]

Parameters
- **arr** *(array)* – a Boolean array, used as a filter
- **inds** *(array)* – any other array (usually, an array of indices) of the same size

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| ```python
inds = cv.itrue(np.array([True,False,True,True]), inds=np.array([5,22,47,93]))
``` |

**ifalse**(arr, inds)

Returns the indices that are true in the array – name is short for indices[false]

Parameters
- **arr** *(array)* – a Boolean array, used as a filter
- **inds** *(array)* – any other array (usually, an array of indices) of the same size

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| ```python
inds = cv.ifalse(np.array([True,False,True,True]), inds=np.array([5,22,47,93]))
``` |

**idefined**(arr, inds)

Returns the indices that are defined in the array – name is short for indices[defined]

Parameters
- **arr** *(array)* – any array, used as a filter
- **inds** *(array)* – any other array (usually, an array of indices) of the same size

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| ```python
inds = cv.idefined(np.array([3,np.nan,np.nan,4]), inds=np.array([5,22,47,93]))
``` |
**iundefined** *(arr, inds)*

Returns the indices that are undefined in the array – name is short for indices[undefined]

**Parameters**
- **arr** *(array)* – any array, used as a filter
- **inds** *(array)* – any other array (usually, an array of indices) of the same size

**Example:**

```python
inds = cv.iundefined(np.array([3, np.nan, np.nan, 4]), inds=np.array([5, 22, 47, 93]))
```

**itruei** *(arr, inds)*

Returns the indices that are true in the array – name is short for indices[true[indices]]

**Parameters**
- **arr** *(array)* – a Boolean array, used as a filter
- **inds** *(array)* – an array of indices for the original array

**Example:**

```python
inds = cv.itruei(np.array([True, False, True, True, False, False, True, False]), inds=np.array([0, 1, 3, 5]))
```

**ifalsei** *(arr, inds)*

Returns the indices that are false in the array – name is short for indices[false[indices]]

**Parameters**
- **arr** *(array)* – a Boolean array, used as a filter
- **inds** *(array)* – an array of indices for the original array

**Example:**

```python
inds = cv.ifalsei(np.array([True, False, True, True, False, False, True, False]), inds=np.array([0, 1, 3, 5]))
```

**idefinedi** *(arr, inds)*

Returns the indices that are defined in the array – name is short for indices[defined[indices]]

**Parameters**
- **arr** *(array)* – any array, used as a filter
- **inds** *(array)* – an array of indices for the original array

**Example:**

```python
inds = cv.idefinedi(np.array([4, np.nan, 0, np.nan, np.nan, 4, 7, 4, np.nan]), inds=np.array([0, 1, 3, 5]))
```

**iundefinedi** *(arr, inds)*

Returns the indices that are undefined in the array – name is short for indices[defined[indices]]

**Parameters**
- **arr** *(array)* – any array, used as a filter
- **inds** *(array)* – an array of indices for the original array

**Example:**

```python
```
inds = cv.iundefinedi(np.array([4, np.nan, 0, np.nan, np.nan, 4, 7, 4, np.nan]), inds=np.array([0, 1, 3, 5]))

1.10.2.16 covasim.version module

Version and license information.
C

covasim, 149
 covasim.analysis, 150
 covasim.base, 160
 covasim.data, 149
 covasim.data.country_age_data, 149
 covasim.data.household_size_data, 149
 covasim.data.loaders, 149
 covasim.data.state_age_data, 150
 covasim.defaults, 169
 covasim.immunity, 170
 covasim.interventions, 170
 covasim.misc, 183
 covasim.parameters, 188
 covasim.people, 189
 covasim.plotting, 192
 covasim.population, 193
 covasim.requirements, 195
 covasim.run, 196
 covasim.settings, 206
 covasim.sim, 206
 covasim.utils, 214
 covasim.version, 220
Index

A

add_contacts() (BasePeople method), 166
add_layer() (Contacts method), 167
age_histogram (class in covasim.analysis), 151
AlreadyRunError, 214
Analyzer (class in covasim.analysis), 150
animate() (TransTree method), 159
append() (Layer method), 168
apply() (age_histogram method), 152
apply() (Analyzer method), 150
apply() (BaseVaccination method), 179
apply() (change_beta method), 173
apply() (clip_edges method), 173
apply() (contact_tracing method), 176
apply() (daily_age_stats method), 153
apply() (daily_stats method), 154
apply() (dynamic_pars method), 172
apply() (historical_wave method), 182
apply() (Intervention method), 171
apply() (nab_histogram method), 154
apply() (sequence method), 172
apply() (simple_vaccine method), 178
apply() (snapshot method), 151
apply() (test_num method), 174
apply() (test_prob method), 175
apply() (variant method), 170

B

BasePeople (class in covasim.base), 165
BaseSim (class in covasim.base), 160
BaseVaccination (class in covasim.interventions), 178
binomial_arr() (in module covasim.utils), 216
binomial_filter() (in module covasim.utils), 215
brief() (MultiSim method), 201
brief() (Scenarios method), 204
brief() (Sim method), 210

C

calibrate() (Calibration method), 157
calibrate() (Sim method), 210
Calibration (class in covasim.analysis), 156
change_beta (class in covasim.interventions), 172
check_critical() (People method), 190
check_death() (People method), 190
check_diagnosed() (People method), 190
check inds() (People method), 190
check_infectious() (People method), 190
check_quar() (People method), 190
check_recovery() (People method), 190
check_save_version() (in module covasim.misc), 186
check_sciris() (in module covasim.requirements), 195
check_severe() (People method), 190
check_symptomatic() (People method), 190
check_synthpops() (in module covasim.requirements), 196
check_version() (in module covasim.misc), 186
choose (in module covasim.utils), 217
choose_r (in module covasim.utils), 217
choose_w() (in module covasim.utils), 217
clip_edges (class in covasim.interventions), 173
combine() (MultiSim method), 198
compare() (MultiSim method), 198
compare() (Scenarios method), 202
compute() (Fit method), 155
compute_diffs() (Fit method), 155
compute_doubling() (Sim method), 209
compute_fit() (Sim method), 210
compute_gen_time() (Sim method), 209
compute_gof() (in module covasim.misc), 187
compute_gofs() (Fit method), 155
compute_losses() (Fit method), 155
compute_mismatch() (Fit method), 155
compute_r_eff() (Sim method), 209
compute_results() (Sim method), 208
compute_summary() (Sim method), 208
compute_states() (Sim method), 209
compute_windows() (age_histogram method), 152
compute_yield() (Sim method), 208
contact_tracing (class in covasim.interventions), 175
Contacts (class in covasim.base), 167
copy() (BaseSim method), 162
count() (BasePeople method), 165
count_by_variant() (BasePeople method), 165
count_not() (BasePeople method), 165
count_transmissions() (TransTree method), 158
covasim (module), 149
covasim.analysis (module), 150
covasim.base (module), 160
covasim.data (module), 149
covasim.data.country_age_data (module), 149
covasim.data.household_size_data (module), 149
covasim.data.loaders (module), 149
covasim.data.state_age_data (module), 150
covasim.defaults (module), 169
covasim.immunity (module), 170
covasim.interventions (module), 170
covasim.misc (module), 183
covasim.parameters (module), 188
covasim.people (module), 189
covasim.plotting (module), 192
covasim.population (module), 193
covasim.requirements (module), 195
covasim.run (module), 196
covasim.settings (module), 206
covasim.sim (module), 206
covasim.utils (module), 214
covasim.version (module), 220
daily_age_stats (class in covasim.analysis), 152
daily_stats (class in covasim.analysis), 153
date() (BaseSim method), 161
date() (in module covasim.misc), 183
date_formatter() (in module covasim.plotting), 192
date_keys() (BasePeople method), 166
date_range() (in module covasim.misc), 184
datevec (BaseSim attribute), 161
day() (BaseSim method), 161
day() (in module covasim.misc), 183
day() (TransTree method), 158
daydiff() (in module covasim.misc), 183
default_float (in module covasim.defaults), 169
default_int (in module covasim.defaults), 169
defined() (BasePeople method), 165
defined() (in module covasim.utils), 218
demo() (in module covasim.sim), 213
diff_sims() (in module covasim.sim), 213
disp() (Intervention method), 171
disp() (MultiSim method), 200
disp() (Scenarios method), 203
disp() (Sim method), 210
dur_keys() (BasePeople method), 166
dynamic_pars (class in covasim.interventions), 172

e estimate_prob() (historical_vaccinate_prob static method), 182
export_pars() (BaseSim method), 162
export_results() (BaseSim method), 162

F
false() (BasePeople method), 165
false() (in module covasim.utils), 217
finalize() (age_histogram method), 152
finalize() (Analyzer method), 150
finalize() (BaseVaccination method), 178
finalize() (clip_edges method), 173
finalize() (Intervention method), 171
finalize() (Sim method), 208
finalize() (snapshot method), 151
finalize() (test_num method), 174
finalize() (test_prob method), 175
finalize_analyzers() (Sim method), 208
finalize_interventions() (Sim method), 208
find_contacts() (Layer method), 168
Fit (class in covasim.analysis), 154
FlexDict (class in covasim.base), 167
from_df() (Layer method), 168
from_people() (BasePeople method), 166
from_sim() (age_histogram method), 152

G
get() (age_histogram method), 152
get() (BasePeople method), 165
get() (snapshot method), 151
get_age_distribution() (in module covasim.data.loaders), 150
get_analyzer() (BaseSim method), 165
get_analyzers() (BaseSim method), 165
get_country_aliases() (in module covasim.data.loaders), 149
get_default_colors() (in module covasim.defaults), 169
get_default_plots() (in module covasim.defaults), 169
get_doubling_time() (in module covasim.misc), 187
get_household_size() (in module covasim.data.loaders), 150
get_intervention() (BaseSim method), 164
get_interventions() (BaseSim method), 164
get_pdf() (in module covasim.utils), 215
get_png_metadata() (in module covasim.misc), 187
get_prognoses() (in module covasim.misc), 189
get_vaccine_choices() (in module covasim.parameters), 189
get_variant_choices() (in module covasim.parameters), 189
get_version_pars() (in module covasim.misc), 187
git_info() (in module covasim.misc), 186

H
historical_vaccinate_prob (class in covasim.interventions), 181
historical_wave (class in covasim.interventions), 182

I
idefined() (in module covasim.utils), 218
idefinedi() (in module covasim.utils), 219
identify_contacts() (contact_tracing method), 176
ifalse() (in module covasim.utils), 218
ifalsei() (in module covasim.utils), 219
indices() (BasePeople method), 166
infect() (People method), 191
init_analyzers() (Sim method), 208
init_contacts() (BasePeople method), 166
init_flows() (People method), 189
init_immunity() (Sim method), 208
init_interventions() (Sim method), 208
init_people() (Sim method), 207
init_results() (Sim method), 207
init_sims() (MultiSim method), 196
init_variants() (Sim method), 208
initialize() (age_histogram method), 152
initialize() (Analyzer method), 150
initialize() (BaseVaccination method), 178
initialize() (change_beta method), 173
initialize() (clip_edges method), 173
initialize() (contact_tracing method), 176
initialize() (daily_age_stats method), 153
initialize() (daily_stats method), 154
initialize() (historical_vaccinate_prob method), 182
initialize() (Intervention method), 171
initialize() (nab_histogram method), 154
initialize() (People method), 190
initialize() (sequence method), 172
initialize() (Sim method), 206
initialize() (simple_vaccine method), 178
initialize() (snapshot method), 151
initialize() (test_num method), 174
initialize() (test_prob method), 175
initialize() (vaccinate_num method), 181
initialize() (vaccinate_prob method), 180
initialize() (variant method), 170
intersect() (daily_stats method), 154
Intervention (class in covasim.interventions), 170
InterventionDict() (in module covasim.interventions), 170
items() (FlexDict method), 167
itru() (in module covasim.utils), 218
itru() (in module covasim.utils), 219
iundefined() (in module covasim.utils), 218
iundefinedi() (in module covasim.utils), 219

K
keys() (BasePeople method), 165
keys() (FlexDict method), 167

L
Layer (class in covasim.base), 167
layer_keys() (BasePeople method), 166
layer_keys() (Sim method), 207
load() (BaseSim static method), 163
load() (in module covasim.misc), 184
load() (MultiSim static method), 199
load() (Scenarios static method), 203
load_data() (in module covasim.misc), 184
load_data() (Sim method), 206
load_population() (Sim method), 207
lock() (BasePeople method), 165

M
make_age_histogram() (Sim method), 211
make_detailed() (TransTree method), 158
make_edgelist() (BasePeople method), 166
make_hybrid_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_metapars() (in module covasim.run), 196
make_microstructured_contacts() (in module covasim.population), 195
make_me