HPVsim

Release 0.4.0

Institute for Disease Modeling

Nov 16, 2022

CONTENTS

HPVsim is a stochastic agent-based simulator, written in Python, for exploring and analyzing human papillomavirus (HPV) transmission.

CHAPTER

ONE

FULL CONTENTS

1.1 HPVsim overview

This repository contains the code for IDM's human papillomavirus simulator, HPVsim.

HPVsim is currently under development.

The structure is as follows:

- HPVsim, in the folder `hpvsim`, is a standalone Python library for performing HPV analyses.
- Data is in the `data` folder.
- Docs are in the `docs` folder.
- Tests are in the `tests` folder.

1.1.1 Installation

The easiest way to install is simply via pip: `pip install hpvsim`. Alternatively, you can clone this repository, then run `pip install -e .` (don't forget the dot!) in this folder to install `hpvsim` and its dependencies. This will make `hpvsim` available on the Python path. The first time HPVsim is imported, it will automatically download the required data files (~30 MB).

1.1.2 Usage

See the tests folder for usage examples.

1.1.3 Documentation

Documentation is available at https://docs.idmod.org/projects/hpvsim/en/latest/.

1.1.4 Contributing

Style guide

Please follow the starsim style guide at: https://github.com/amath-idm/styleguide

1.1.5 Disclaimer

The code in this repository was developed by IDM, the Burnet Institute, and other collaborators to support our joint research on HPV. We've made it publicly available under the MIT License to provide others with a better understanding of our research and an opportunity to build upon it for their own work. Note that HPVsim depends on a number of user-installed Python packages that can be installed automatically via `pip install`. 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 MIT License. See the contributing and code of conduct READMEs for more information.

1.2 Tutorials

These tutorials walk through how to use HPVsim. If you want to explore them interactively, you can run them on Binder via http://tutorials.hpvsim.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 HPVsim is quite simple.

HPVsim is a Python package that can be pip-installed by typing `pip install hpvsim` into a terminal. You can then check that the installation was successful by importing HPVsim with `import hpvsim as hpv`.

The basic design philosophy of HPVsim 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.

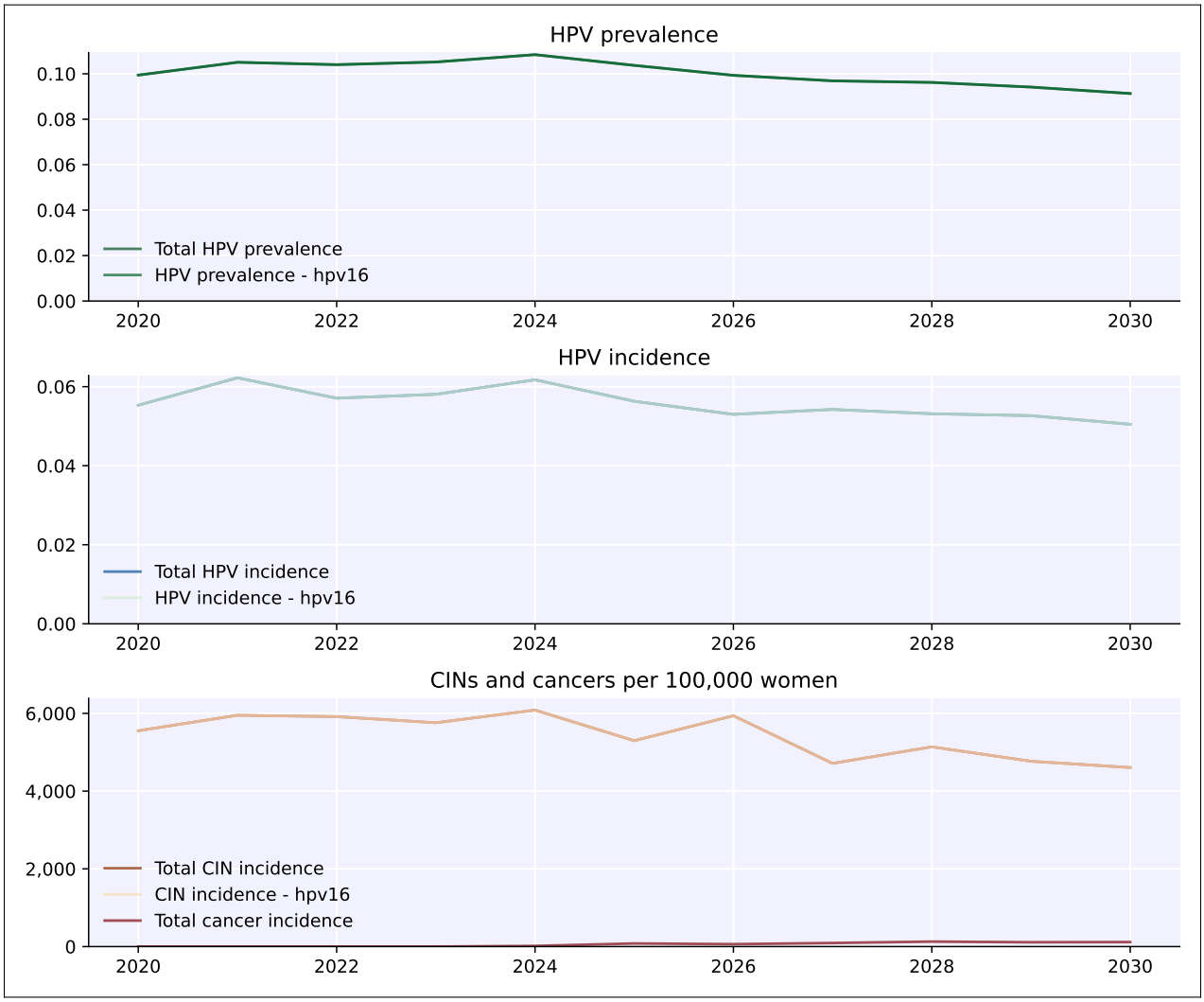
Click here to open an interactive version of this notebook.

1.2.1.1 Hello world

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

[1]:
```
import hpvsim as hpv

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

```
HPVsim 0.4.0 (2022-11-16) - © 2022 by IDM
No genotypes provided, will assume only simulating HPV16 by default
Initializing sim with 20000 agents
  Running 2015.0 ( 0/80) (0.04 s)  ---------------------- 1%
  Running 2017.0 (10/80) (0.29 s)  ••---------------------- 14%
  Running 2019.0 (20/80) (0.54 s)  •••••----------------- 26%
  Running 2021.0 (30/80) (0.81 s)  •••••••-------------- 39%
  Running 2023.0 (40/80) (1.10 s)  •••••••••----------- 51%
  Running 2025.0 (50/80) (1.40 s)  ••••••••••---------- 64%
  Running 2027.0 (60/80) (1.72 s)  ••••••••••••••----- 76%
  Running 2029.0 (70/80) (2.05 s)  •••••••••••••••••--- 89%
Simulation summary:
        1,034 total infections
          240 total cin1s
          267 total cin2s
          130 total cin3s
          637 total cins
           16 total cancers
            0 total cancer detections
            0 total cancer deaths
            0 total detected cancer deaths
          411 total reinfections
          131 total reactivations
            0 total hiv infections
         5.05 total hpv incidence (/100)
        1,736 total cin1 incidence (/100,000)
        1,931 total cin2 incidence (/100,000)
          940 total cin3 incidence (/100,000)
        4,607 total cin incidence (/100,000)
          116 total cancer incidence (/100,000)
         9.13 total hpv prevalence (/100)
```



1.2.1.2 Defining parameters and genotypes, and running simulations

Parameters are defined as a dictionary. Some common parameters to modify are the number of agents in the simulation, the genotypes to simulate, and the start and end dates of the simulation. We can define those as:

[2]:
```
pars = dict(
    n_agents = 10e3,
    genotypes = [16, 18, 'hrhpv'], # Simulate genotypes 16 and 18, plus all other high-
↪risk HPV genotypes pooled together
    start = 1980,
    end = 2030,
)
```

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

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

```
Initializing sim with 10000 agents
  Running 1980.0 ( 0/255) (0.03 s)  ---------------------- 0%
  Running 1982.0 (10/255) (0.18 s)  ---------------------- 4%
  Running 1984.0 (20/255) (0.34 s)  •--------------------- 8%
  Running 1986.0 (30/255) (0.51 s)  ••-------------------- 12%
  Running 1988.0 (40/255) (0.69 s)  •••------------------- 16%
  Running 1990.0 (50/255) (0.88 s)  ••••------------------ 20%
  Running 1992.0 (60/255) (1.08 s)  ••••------------------ 24%
  Running 1994.0 (70/255) (1.28 s)  •••••----------------- 28%
  Running 1996.0 (80/255) (1.49 s)  ••••••---------------- 32%
  Running 1998.0 (90/255) (1.70 s)  •••••••--------------- 36%
  Running 2000.0 (100/255) (1.93 s)  •••••••--------------- 40%
  Running 2002.0 (110/255) (2.16 s)  ••••••••-------------- 44%
  Running 2004.0 (120/255) (2.40 s)  •••••••••------------ 47%
  Running 2006.0 (130/255) (2.64 s)  •••••••••------------ 51%
  Running 2008.0 (140/255) (2.89 s)  ••••••••••---------- 55%
  Running 2010.0 (150/255) (3.15 s)  ••••••••••---------- 59%
  Running 2012.0 (160/255) (3.42 s)  •••••••••••--------- 63%
  Running 2014.0 (170/255) (3.72 s)  •••••••••••--------- 67%
  Running 2016.0 (180/255) (4.00 s)  ••••••••••••------- 71%
  Running 2018.0 (190/255) (4.30 s)  ••••••••••••------- 75%
  Running 2020.0 (200/255) (4.61 s)  •••••••••••••------ 79%
  Running 2022.0 (210/255) (4.93 s)  •••••••••••••••---- 83%
  Running 2024.0 (220/255) (5.25 s)  ••••••••••••••••--- 87%
  Running 2026.0 (230/255) (5.60 s)  •••••••••••••••••-- 91%
  Running 2028.0 (240/255) (5.94 s)  •••••••••••••••••-- 95%
  Running 2030.0 (250/255) (6.31 s)  ••••••••••••••••••- 98%
Simulation summary:
        3,402 total infections
          538 total cin1s
          485 total cin2s
          249 total cin3s
        1,272 total cins
           29 total cancers
            0 total cancer detections
           23 total cancer deaths
            0 total detected cancer deaths
        1,186 total reinfections
          533 total reactivations
            0 total hiv infections
         3.38 total hpv incidence (/100)
        2,522 total cin1 incidence (/100,000)
        2,273 total cin2 incidence (/100,000)
        1,167 total cin3 incidence (/100,000)
        5,963 total cin incidence (/100,000)
          136 total cancer incidence (/100,000)
        14.21 total hpv prevalence (/100)
```

[3]:
```
Sim(<no label>; 1980 to 2030; pop: 10000 default; epi: 3402, 29)
```

This will generate a results dictionary `sim.results`. Results by genotype are named things like `sim.results['infections']` and stored as arrays where each row corresponds to a genotype, while totals across all genotypes have names like `sim.results['total_infections']` or `sim.results['total_cancers']`.

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 = hpv.Sim(n_agents=10e3, start=1980, end=2030)
sim.run()
```

```
No genotypes provided, will assume only simulating HPV16 by default
Initializing sim with 10000 agents
  Running 1980.0 ( 0/255) (0.03 s)  ---------------------- 0%
  Running 1982.0 (10/255) (0.21 s)  ---------------------- 4%
  Running 1984.0 (20/255) (0.40 s)  •--------------------- 8%
  Running 1986.0 (30/255) (0.59 s)  ••-------------------- 12%
  Running 1988.0 (40/255) (0.81 s)  •••------------------- 16%
  Running 1990.0 (50/255) (1.03 s)  ••••------------------ 20%
  Running 1992.0 (60/255) (1.26 s)  ••••------------------ 24%
  Running 1994.0 (70/255) (1.50 s)  •••••----------------- 28%
  Running 1996.0 (80/255) (1.74 s)  ••••••---------------- 32%
  Running 1998.0 (90/255) (1.99 s)  •••••••--------------- 36%
  Running 2000.0 (100/255) (2.26 s)  •••••••--------------- 40%
  Running 2002.0 (110/255) (2.52 s)  ••••••••-------------- 44%
  Running 2004.0 (120/255) (2.80 s)  •••••••••------------ 47%
  Running 2006.0 (130/255) (3.10 s)  •••••••••----------- 51%
  Running 2008.0 (140/255) (3.39 s)  ••••••••••---------- 55%
  Running 2010.0 (150/255) (3.70 s)  ••••••••••---------- 59%
  Running 2012.0 (160/255) (4.02 s)  •••••••••••--------- 63%
  Running 2014.0 (170/255) (4.36 s)  ••••••••••••-------- 67%
  Running 2016.0 (180/255) (4.70 s)  ••••••••••••------- 71%
  Running 2018.0 (190/255) (5.06 s)  •••••••••••••------ 75%
  Running 2020.0 (200/255) (5.42 s)  •••••••••••••----- 79%
  Running 2022.0 (210/255) (5.80 s)  ••••••••••••••---- 83%
  Running 2024.0 (220/255) (6.20 s)  ••••••••••••••---- 87%
  Running 2026.0 (230/255) (6.60 s)  •••••••••••••••--- 91%
  Running 2028.0 (240/255) (7.02 s)  •••••••••••••••--- 95%
  Running 2030.0 (250/255) (7.45 s)  ••••••••••••••••-- 98%
Simulation summary:
        1,519 total infections
          362 total cin1s
          333 total cin2s
          182 total cin3s
          877 total cins
           26 total cancers
            0 total cancer detections
           11 total cancer deaths
            0 total detected cancer deaths
          586 total reinfections
          201 total reactivations
            0 total hiv infections
         4.92 total hpv incidence (/100)
        1,659 total cin1 incidence (/100,000)
        1,526 total cin2 incidence (/100,000)
          834 total cin3 incidence (/100,000)
        4,020 total cin incidence (/100,000)
          119 total cancer incidence (/100,000)
         8.26 total hpv prevalence (/100)
```

(continues on next page)

(continued from previous page)

[4]:
```
Sim(<no label>; 1980 to 2030; pop: 10000 default; epi: 1519, 26)
```

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:

[5]:
```
sim = hpv.Sim(pars, end=2050) # Use parameters defined above, except set the end data to␣
↪2050
sim.run()
```

```
Initializing sim with 10000 agents
  Running 1980.0 ( 0/355) (0.05 s)  ---------------------- 0%
  Running 1982.0 (10/355) (0.21 s)  ---------------------- 3%
  Running 1984.0 (20/355) (0.37 s)  •--------------------- 6%
  Running 1986.0 (30/355) (0.54 s)  •--------------------- 9%
  Running 1988.0 (40/355) (0.72 s)  ••-------------------- 12%
  Running 1990.0 (50/355) (0.92 s)  ••-------------------- 14%
  Running 1992.0 (60/355) (1.11 s)  •••------------------- 17%
  Running 1994.0 (70/355) (1.31 s)  ••••------------------ 20%
  Running 1996.0 (80/355) (1.52 s)  ••••------------------ 23%
  Running 1998.0 (90/355) (1.74 s)  •••••----------------- 26%
  Running 2000.0 (100/355) (1.97 s)  •••••----------------- 28%
  Running 2002.0 (110/355) (2.20 s)  ••••••---------------- 31%
  Running 2004.0 (120/355) (2.44 s)  ••••••---------------- 34%
  Running 2006.0 (130/355) (2.69 s)  •••••••--------------- 37%
  Running 2008.0 (140/355) (2.94 s)  •••••••--------------- 40%
  Running 2010.0 (150/355) (3.20 s)  ••••••••-------------- 43%
  Running 2012.0 (160/355) (3.46 s)  ••••••••-------------- 45%
  Running 2014.0 (170/355) (3.75 s)  ••••••••-------------- 48%
  Running 2016.0 (180/355) (4.03 s)  •••••••••------------- 51%
  Running 2018.0 (190/355) (4.32 s)  •••••••••------------- 54%
  Running 2020.0 (200/355) (4.62 s)  ••••••••••------------ 57%
  Running 2022.0 (210/355) (4.94 s)  ••••••••••------------ 59%
  Running 2024.0 (220/355) (5.26 s)  •••••••••••----------- 62%
  Running 2026.0 (230/355) (5.60 s)  •••••••••••----------- 65%
  Running 2028.0 (240/355) (5.95 s)  •••••••••••----------- 68%
  Running 2030.0 (250/355) (6.32 s)  ••••••••••••---------- 71%
  Running 2032.0 (260/355) (6.69 s)  ••••••••••••---------- 74%
  Running 2034.0 (270/355) (7.07 s)  •••••••••••••--------- 76%
  Running 2036.0 (280/355) (7.46 s)  •••••••••••••--------- 79%
  Running 2038.0 (290/355) (7.86 s)  ••••••••••••••------- 82%
  Running 2040.0 (300/355) (8.27 s)  ••••••••••••••------- 85%
  Running 2042.0 (310/355) (8.69 s)  •••••••••••••••------ 88%
  Running 2044.0 (320/355) (9.13 s)  •••••••••••••••••---- 90%
  Running 2046.0 (330/355) (9.58 s)  •••••••••••••••••---- 93%
  Running 2048.0 (340/355) (10.05 s)  ••••••••••••••••••-- 96%
  Running 2050.0 (350/355) (10.52 s)  ••••••••••••••••••-- 99%
Simulation summary:
        4,452 total infections
          599 total cin1s
          631 total cin2s
          335 total cin3s
```

(continues on next page)

(continued from previous page)

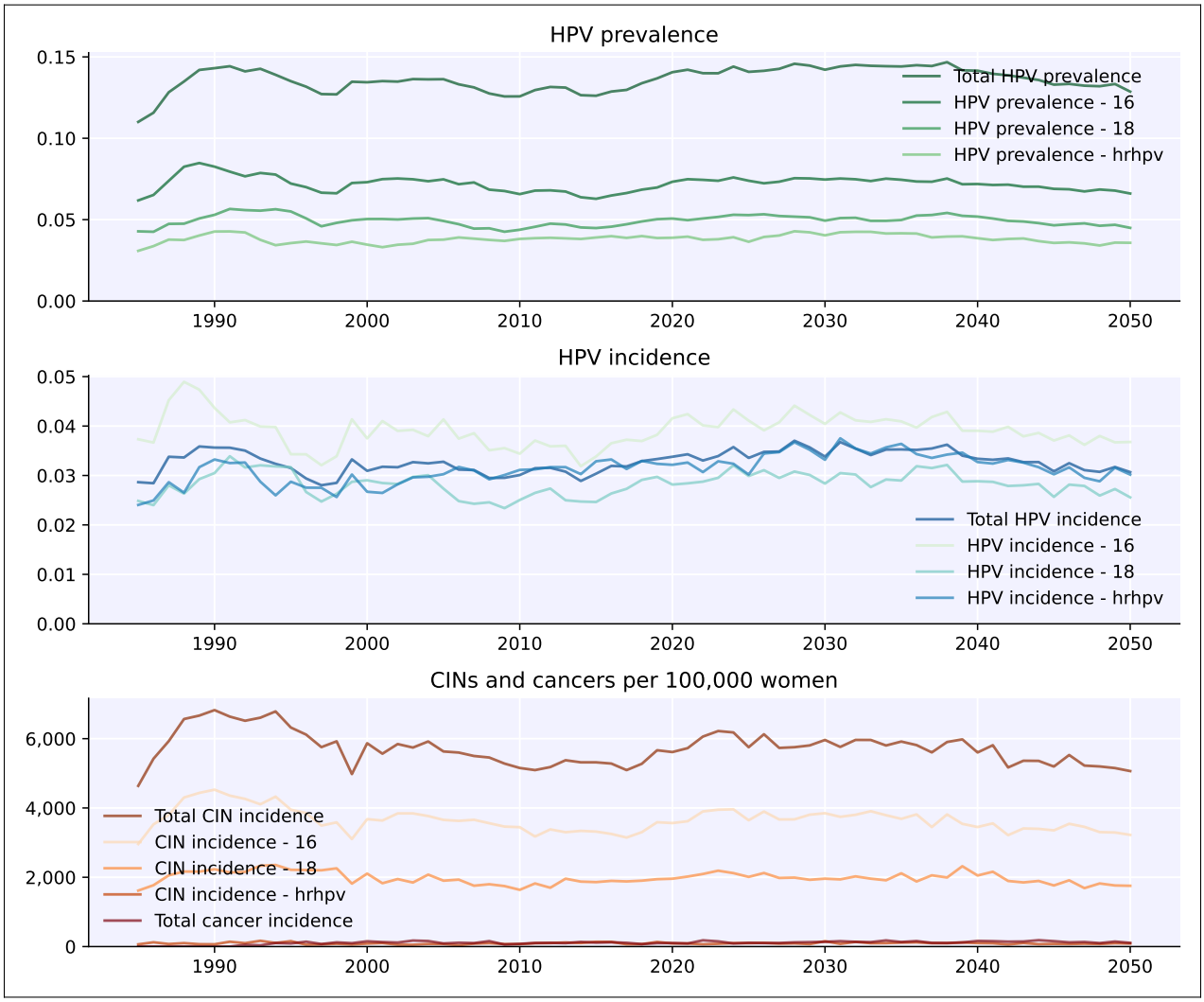
```
        1,565 total cins
           33 total cancers
            0 total cancer detections
           27 total cancer deaths
            0 total detected cancer deaths
        1,578 total reinfections
          773 total reactivations
            0 total hiv infections
         3.07 total hpv incidence (/100)
        1,938 total cin1 incidence (/100,000)
        2,042 total cin2 incidence (/100,000)
        1,084 total cin3 incidence (/100,000)
        5,064 total cin incidence (/100,000)
          107 total cancer incidence (/100,000)
        12.87 total hpv prevalence (/100)
```

[5]:
```
Sim(<no label>; 1980 to 2050; pop: 10000 default; epi: 4452, 33)
```

1.2.1.3 Plotting results

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

[6]:
```
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.

[7]:
```
import hpvsim as hpv

# Custom vaccination intervention
def custom_vx(sim):
    if sim.yearvec[sim.t] == 2000:
        target_group = (sim.people.age>9) * (sim.people.age<14)
        sim.people.peak_imm[0, target_group] = 1

pars = dict(
    location = 'tanzania', # Use population characteristics for Japan
    n_agents = 10e3, # Have 50,000 people total in the population
    start = 1980, # Start the simulation in 1980
    n_years = 50, # Run the simulation for 50 years
```

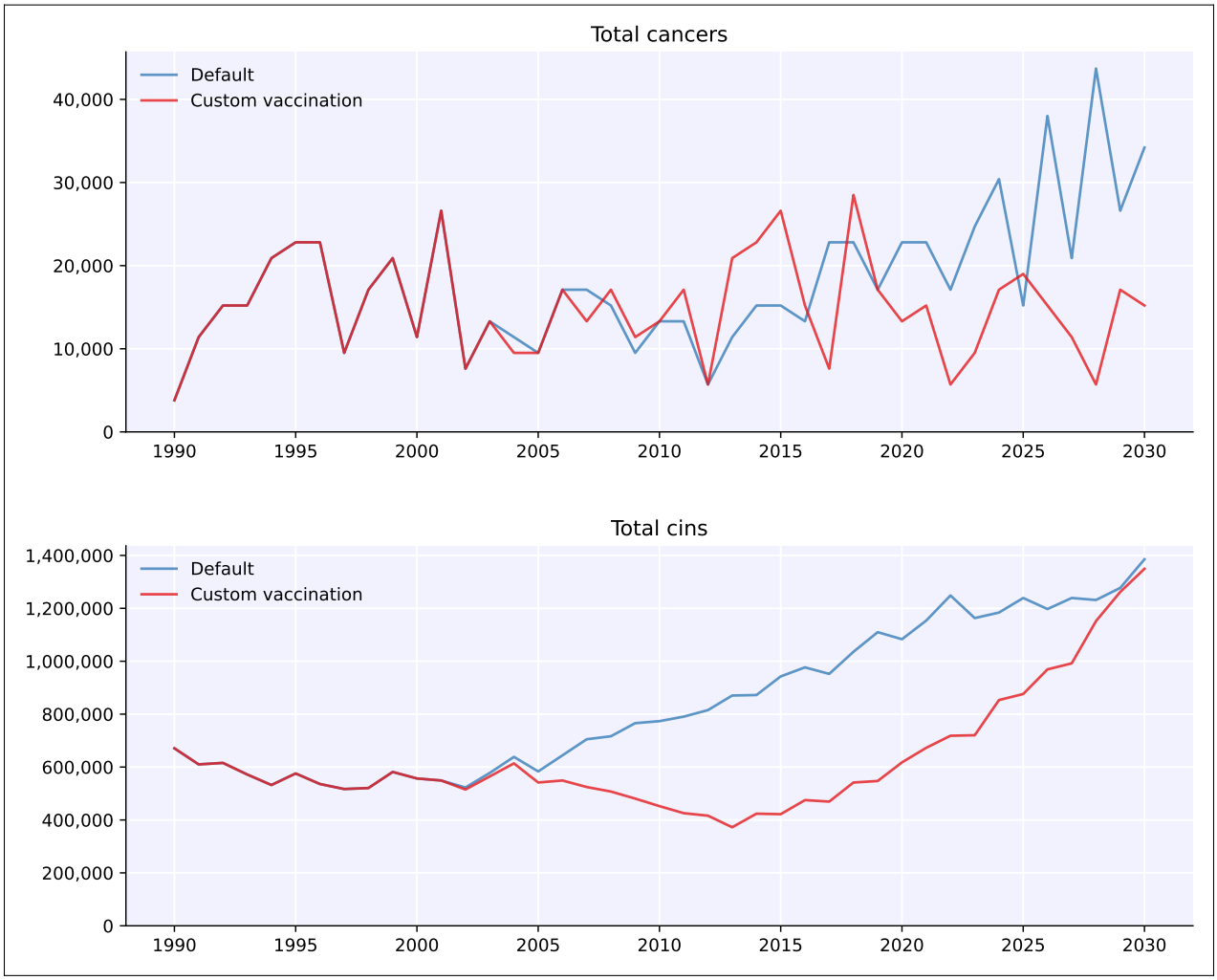
(continues on next page)

(continued from previous page)

```
    burnin = 10, # Discard the first 20 years as burnin period
    verbose = 0, # Do not print any output
)

# Running with multisims -- see Tutorial 3
s1 = hpv.Sim(pars, label='Default')
s2 = hpv.Sim(pars, interventions=custom_vx, label='Custom vaccination')
msim = hpv.MultiSim([s1, s2])
msim.run()
fig = msim.plot(['total_cancers', 'total_cins'])
```

```
Loading location-specific demographic data for "tanzania"
Loading location-specific demographic data for "tanzania"
No genotypes provided, will assume only simulating HPV16 by default
No genotypes provided, will assume only simulating HPV16 by default
```

```
/home/docs/checkouts/readthedocs.org/user_builds/institute-for-disease-modeling-hpvsim/
↪envs/latest/lib/python3.9/site-packages/hpvsim/data/loaders.py:209: FutureWarning: The␣
↪default value of numeric_only in DataFrameGroupBy.sum is deprecated. In a future␣
↪version, numeric_only will default to False. Either specify numeric_only or select␣
↪only columns which should be valid for the function.
  dd = full_df.groupby("Time").sum()["PopTotal"]
/home/docs/checkouts/readthedocs.org/user_builds/institute-for-disease-modeling-hpvsim/
↪envs/latest/lib/python3.9/site-packages/hpvsim/data/loaders.py:209: FutureWarning: The␣
↪default value of numeric_only in DataFrameGroupBy.sum is deprecated. In a future␣
↪version, numeric_only will default to False. Either specify numeric_only or select␣
↪only columns which should be valid for the function.
  dd = full_df.groupby("Time").sum()["PopTotal"]
```



[ ]:

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

HPVsim 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 `hpv.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:

[1]:
```
import hpvsim as hpv

hpv.options(jupyter=True, verbose=0) # Standard options for Jupyter notebook

sim = hpv.Sim()
sim.run()
```

```
HPVsim 0.4.0 (2022-11-16) - © 2022 by IDM
No genotypes provided, will assume only simulating HPV16 by default
```

[1]:
```
Sim(<no label>; 2015.0 to 2030.0; pop: 20000 default; epi: 1034, 16)
```

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()`:

[2]:
```
sim.brief()
```

```
Sim(<no label>; 2015.0 to 2030.0; pop: 20000 default; epi: 1034, 16)
```

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

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

```
Simulation summary:
        1,034 total infections
          240 total cin1s
          267 total cin2s
          130 total cin3s
          637 total cins
           16 total cancers
            0 total cancer detections
            0 total cancer deaths
            0 total detected cancer deaths
          411 total reinfections
          131 total reactivations
            0 total hiv infections
         5.05 total hpv incidence (/100)
        1,736 total cin1 incidence (/100,000)
        1,931 total cin2 incidence (/100,000)
          940 total cin3 incidence (/100,000)
        4,607 total cin incidence (/100,000)
          116 total cancer incidence (/100,000)
         9.13 total hpv prevalence (/100)
```

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

[4]:
```
sim.disp()
```

```
<hpvsim.sim.Sim at 0x7f7a543e98e0>
------------------------------------------------------------
Methods:
  _brief()            get_intervention()  reset_layer_pars()
  _disp()             get_interventio...  result_keys()
  _get_ia()           get_t()             run()
  brief()             init_analyzers()    save()
  compute_fit()       init_genotypes()    set_metadata()
  compute_results()   init_immunity()     set_seed()
  compute_states()    init_interventi...  shrink()
  compute_summary()   init_people()       step()
  copy()              init_results()      summarize()
  disp()              init_states()       to_df()
  export_pars()       initialize()        to_excel()
  export_results()    layer_keys()        to_json()
  finalize()          load()              update_pars()
  finalize_analyz...  load_data()         validate_init_c...
  get_analyzer()      load_hiv_data()     validate_layer_...
  get_analyzers()     plot()              validate_pars()
------------------------------------------------------------
Properties:
  n
------------------------------------------------------------
 _default_ver: None
   _orig_pars: {'n_agents': 20000, 'total_pop': None, 'pop_scale':
               1.0, 'use_multi [...]
 art_datafile: None
     complete: True
      created: None
         data: None
     datafile: None
     hiv_data: #0. 'infection_rates': None
               #1. 'art_adherence':   None
 hiv_datafile: None
  initialized: True
        label: None
         npts: 80
         pars: {'n_agents': 20000, 'total_pop': None, 'pop_scale':
               1.0, 'use_multi [...]
       people: People(n=30682; layers: m, c, o)
      popdict: None
      popfile: None
     res_npts: 16
     res_tvec: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
               12, 13, 14,  [...]
  res_yearvec: array([2015., 2016., 2017., 2018., 2019., 2020., 2021.,
               2022., 2023 [...]
      resfreq: 5
      results: #0. 'total_infections':
               <hpvsim.base.Result at 0x7f7a26942d30>
```

(continues on next page)

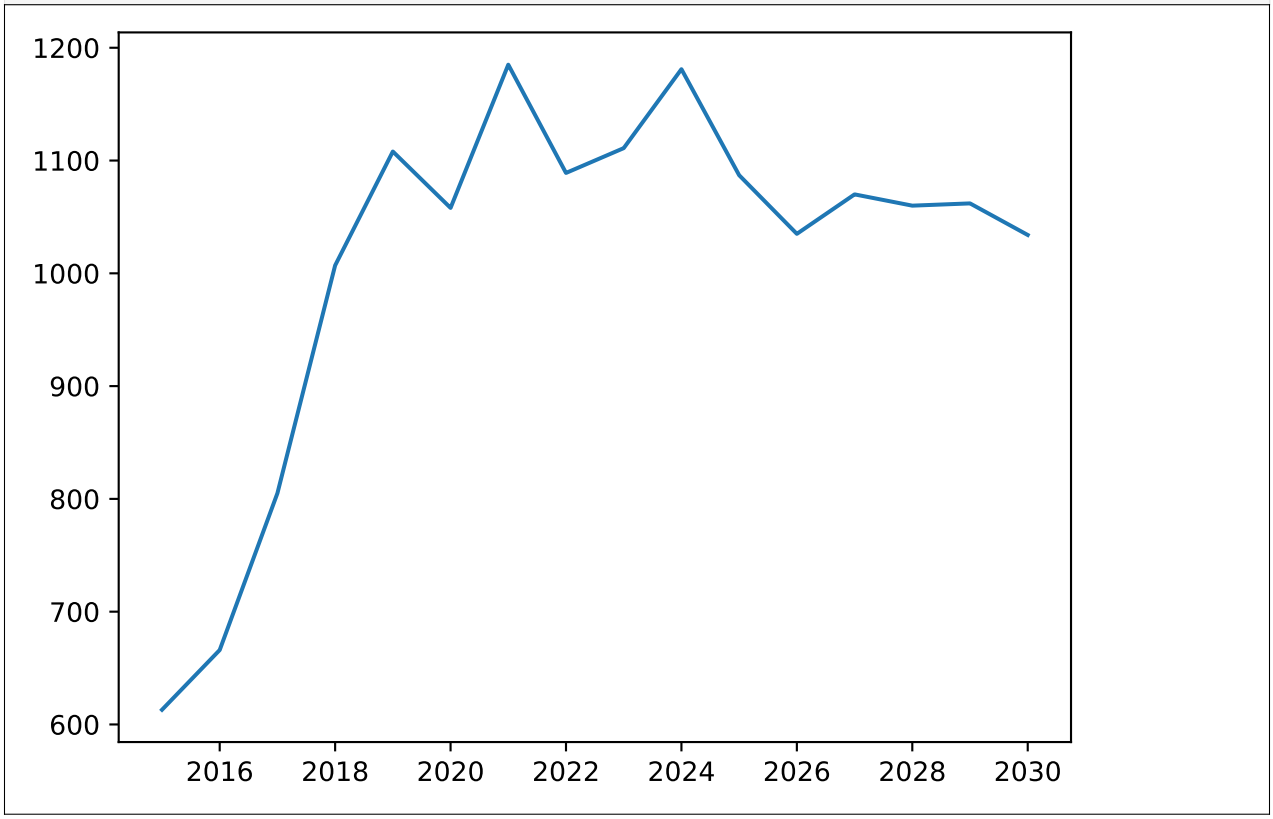
(continued from previous page)

```
               ---- [...]
results_ready: True
      summary: #0. 'total_infections':             1034.0
               #1. 'total_cin1s':       [...]
            t: 79
         tvec: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
               12, 13, 14,  [...]
        years: array([2015., 2016., 2017., 2018., 2019., 2020., 2021.,
               2022., 2023 [...]
      yearvec: array([2015. , 2015.2, 2015.4, 2015.6, 2015.8, 2016. ,
               2016.2, 2016 [...]
------------------------------------------------------------
```

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:

[5]:
```
import pylab as plt # Shortcut for import matplotlib.pyplot as plt
plt.plot(sim.results['year'], sim.results['total_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.total_infections.values`.)

An alternative, you can also select one or more quantities to plot with the first (`to_plot`) argument, e.g.

[6]:
```
sim.plot(to_plot=['total_infections', 'total_hpv_incidence']);
```

While we can save this figure using Matplotlib's built-in `savefig()`, if we use HPVsim's `hpv.savefig()` we get a couple of advantages:

[7]:
```
hpv.savefig('my-fig.png')
```

[7]:
```
'my-fig.png'
```

```
<Figure size 614.4x460.8 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.

[8]:
```
hpv.get_png_metadata('my-fig.png')
```

```
HPVsim version: 0.4.0
HPVsim branch: Branch N/A
```

(continues on next page)

(continued from previous page)

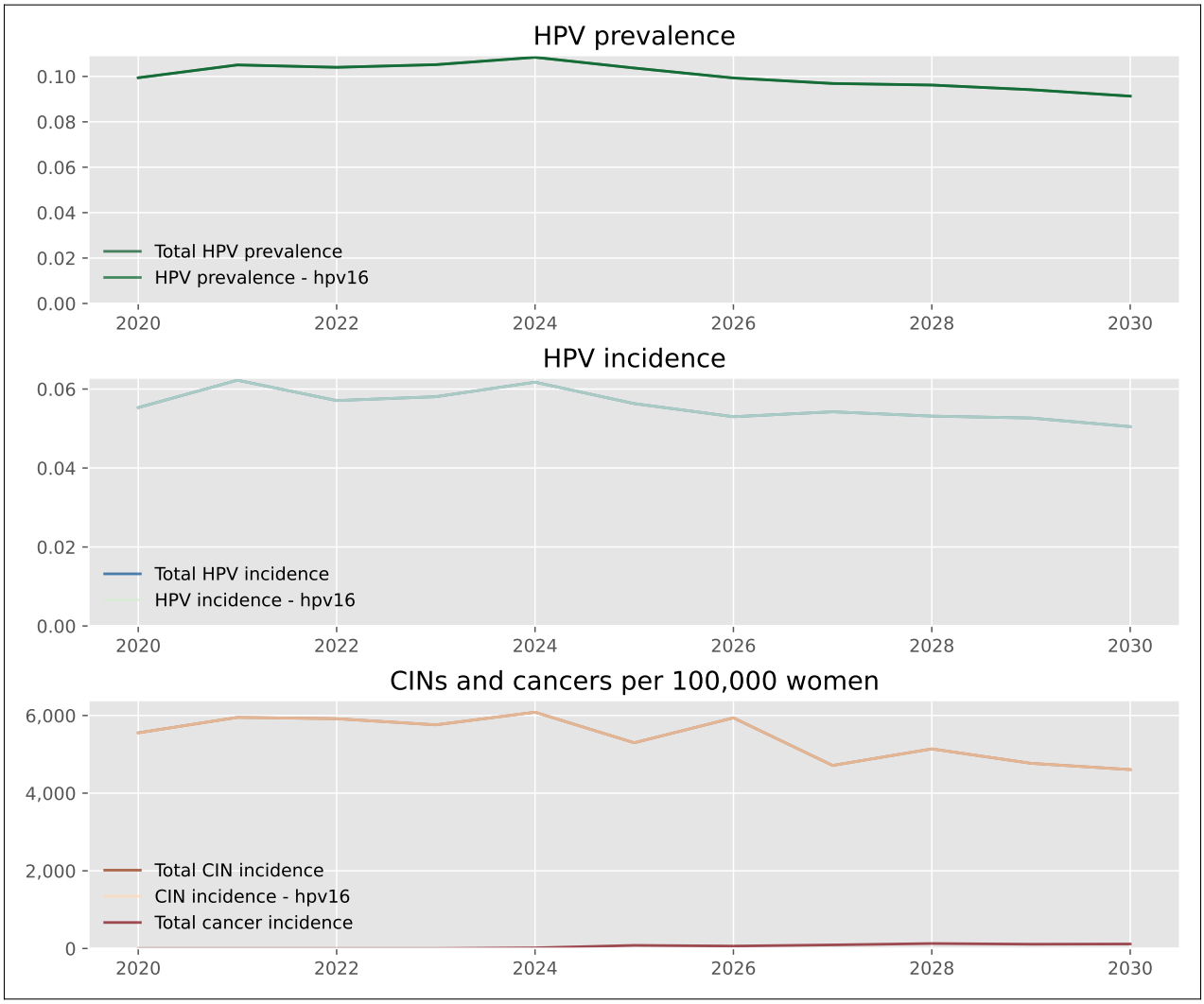
```
HPVsim hash: Hash N/A
HPVsim date: Date N/A
HPVsim caller branch: Branch N/A
HPVsim caller hash: Hash N/A
HPVsim caller date: Date N/A
HPVsim caller filename: /home/docs/checkouts/readthedocs.org/user_builds/institute-for-
↪disease-modeling-hpvsim/envs/latest/lib/python3.9/site-packages/hpvsim/misc.py
HPVsim current time: 2022-Nov-16 06:41:38
HPVsim calling file: /tmp/ipykernel_1079/1797766398.py
```

1.2.2.4 Customizing plots

We saw above how to set default plot configuration options for Jupyter. HPVsim provides a lot of flexibility in customizing the appearance of plots as well. There are three different levels at which you can set plotting options: global, just for HPVsim, or just for the current plot. To give an example with changing the figure DPI: - Change the setting globally (for both HPVsim and Matplotlib): `sc.options(dpi=150)` or `pl.rc('figure', dpi=150)` (where `sc` is `import sciris as sc`) - Change for HPVsim plots, but not for Matplotlib plots: `hpv.options(dpi=150)` - Change for the current HPVsim plot, but not other HPVsim plots: `sim.plot(dpi=150)`

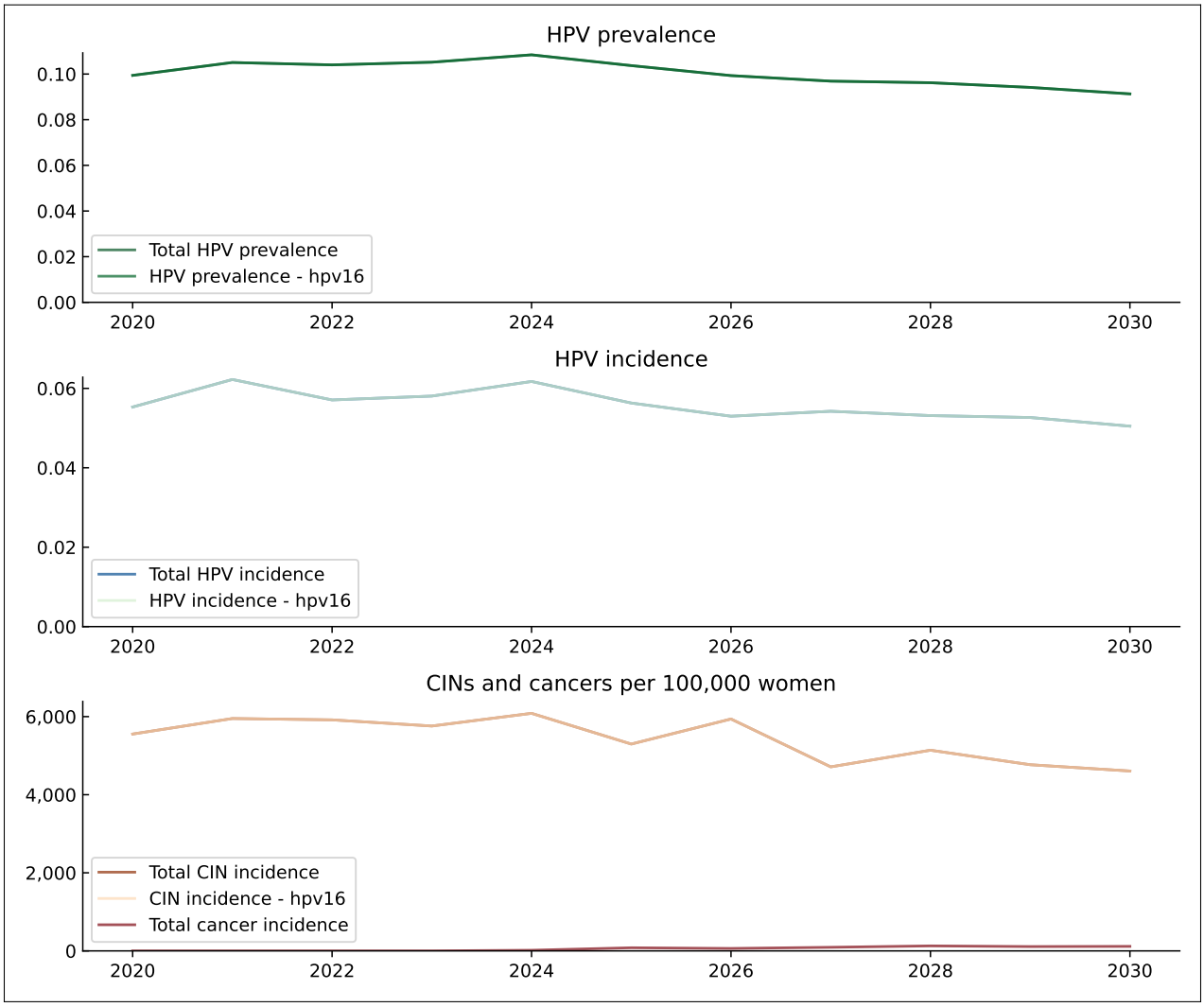
The easiest way to change the style of HPVsim plots is with the `style` argument. For example, to plot using a built-in Matplotlib style would simply be:

[9]:
```
sim.plot(style='ggplot');
```



In addition to the default style (`'hpvsim'`), there is also a "simple" style. You can combine built-in styles with additional overrides, including any valid Matplotlib commands:

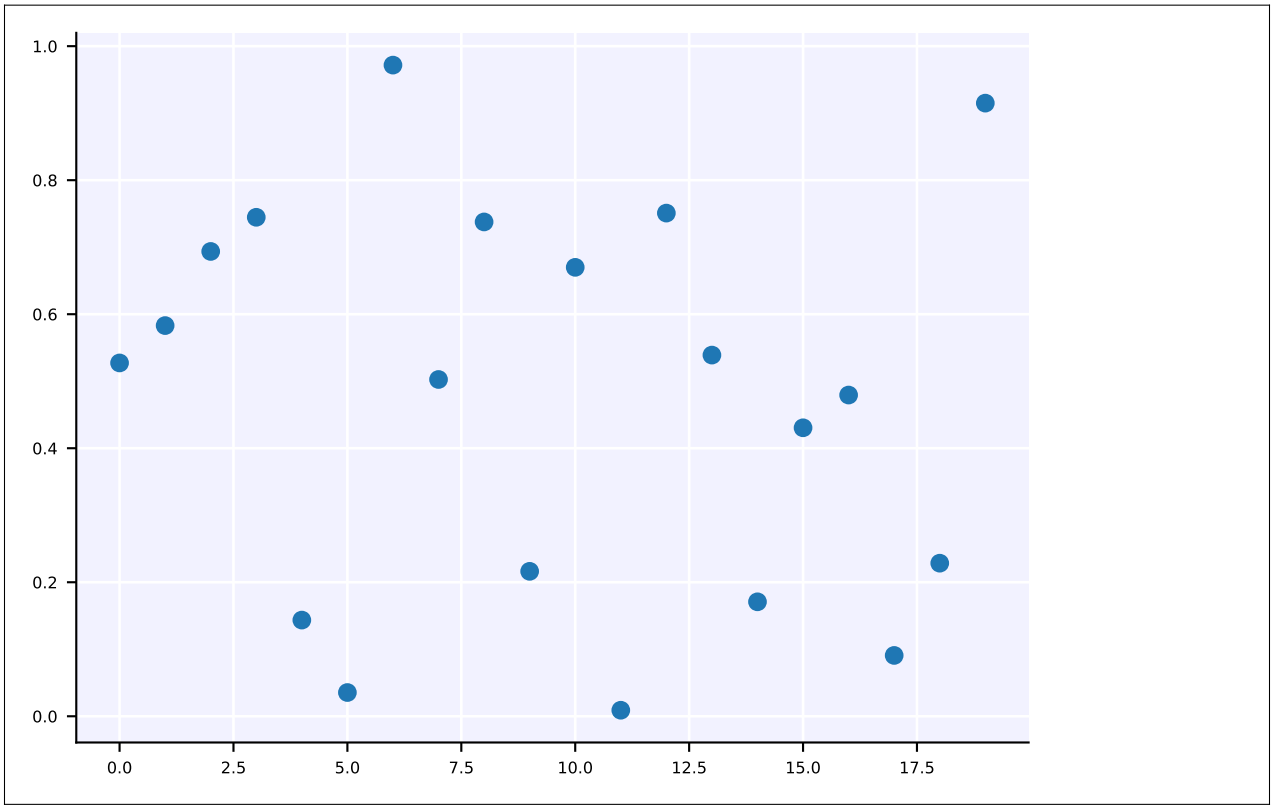
[10]:
```
sim.plot(style='simple', legend_args={'frameon':True}, style_args={'ytick.direction':'in
↪'});
```



Although most style handling is done automatically, you can also use it yourself in a with block, e.g.:

[11]:
```
import numpy as np
with hpv.options.with_style(fontsize=6):
    sim.plot() # This will have 6 point font
    plt.figure(); plt.plot(np.random.rand(20), 'o') # So will this
```


HPV prevalence
0.10
0.08
0.06
0.04
0.02
0.00
Total HPV prevalence
HPV prevalence - hpv16
2020
2022
2024
2026
2028
2030
HPV incidence
0.06
0.05
0.04
0.03
0.02
0.01
0.00
Total HPV incidence
HPV incidence - hpv16
2020
2022
2024
2026
2028
2030
CINs and cancers per 100,000 women
6,000
5,000
4,000
3,000
2,000
1,000
0
Total CIN incidence
CIN incidence - hpv16
Total cancer incidence
2020
2022
2024
2026
2028
2030



1.2.2.5 Saving options

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

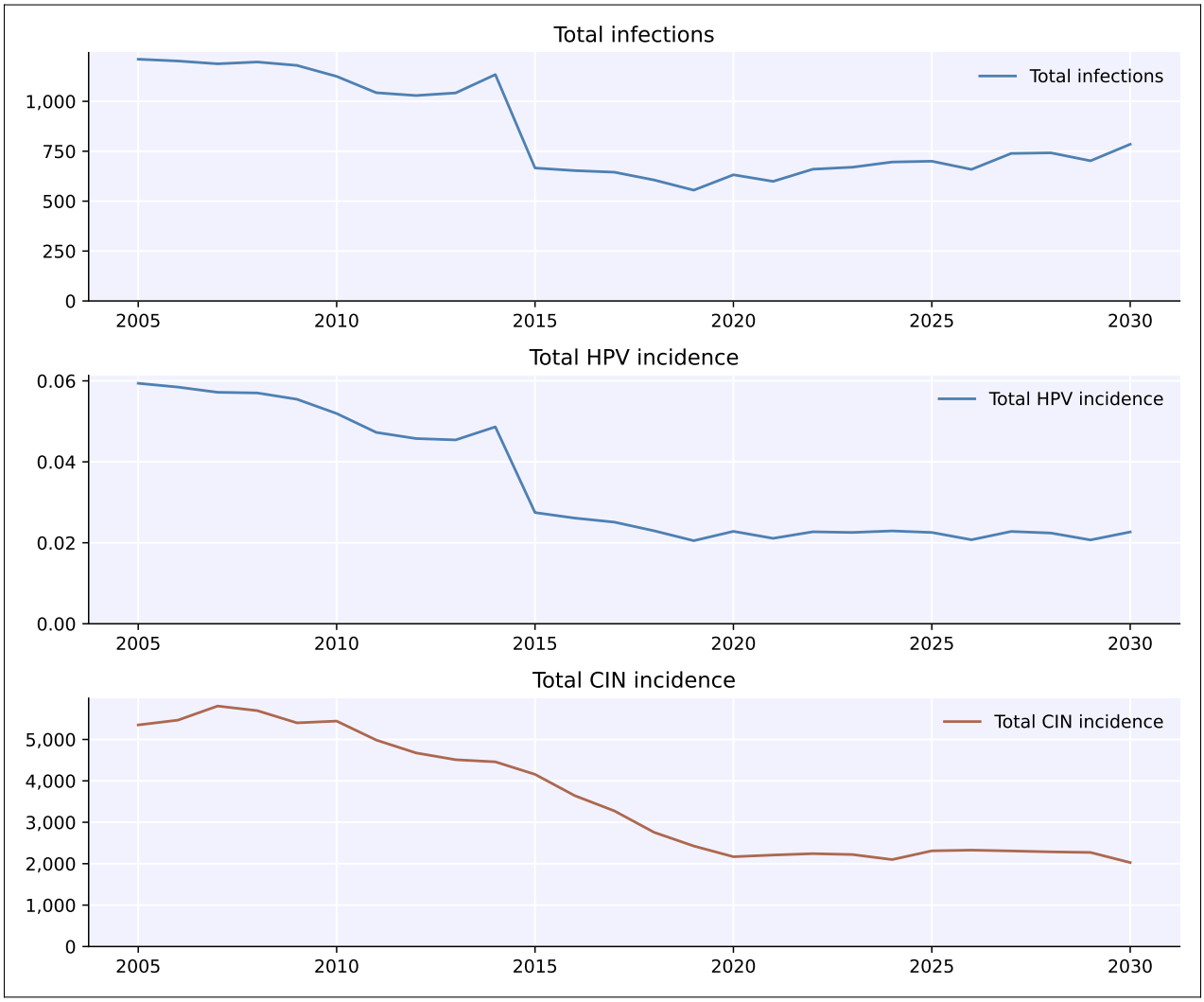
[12]:
```
sim.save('my-sim.sim')
```

[12]:
```
'/home/docs/checkouts/readthedocs.org/user_builds/institute-for-disease-modeling-hpvsim/
↪checkouts/latest/docs/tutorials/my-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:

[13]:
```
sim_orig = hpv.Sim(start=2000, end=2030, label='Load & save example')
sim_orig.run(until='2015')
sim_orig.save('my-half-finished-sim.sim') # Note: HPVsim always saves the people if the␣
↪sim isn't finished running yet

sim = hpv.load('my-half-finished-sim.sim')
sim['beta'] *= 0.3
sim.run()
sim.plot(['total_infections', 'total_hpv_incidence', 'total_cin_incidence']);
```

```
No genotypes provided, will assume only simulating HPV16 by default
```



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:

[14]:
```
import pandas as pd

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

```
WARNING: skipping infections from export since not 1D array
WARNING: skipping cin1s from export since not 1D array
WARNING: skipping cin2s from export since not 1D array
WARNING: skipping cin3s from export since not 1D array
WARNING: skipping cins from export since not 1D array
WARNING: skipping cancers from export since not 1D array
WARNING: skipping reinfections from export since not 1D array
WARNING: skipping reactivations from export since not 1D array
WARNING: skipping n_susceptible from export since not 1D array
```

(continues on next page)

(continued from previous page)

```
WARNING: skipping n_infectious from export since not 1D array
WARNING: skipping n_inactive from export since not 1D array
WARNING: skipping n_no_dysp from export since not 1D array
WARNING: skipping n_cin1 from export since not 1D array
WARNING: skipping n_cin2 from export since not 1D array
WARNING: skipping n_cin3 from export since not 1D array
WARNING: skipping n_cancerous from export since not 1D array
WARNING: skipping n_infected from export since not 1D array
WARNING: skipping n_cin from export since not 1D array
WARNING: skipping n_precin from export since not 1D array
WARNING: skipping n_latent from export since not 1D array
WARNING: skipping hpv_incidence from export since not 1D array
WARNING: skipping cin1_incidence from export since not 1D array
WARNING: skipping cin2_incidence from export since not 1D array
WARNING: skipping cin3_incidence from export since not 1D array
WARNING: skipping cin_incidence from export since not 1D array
WARNING: skipping cancer_incidence from export since not 1D array
WARNING: skipping total_infections_by_sex from export since not 1D array
WARNING: skipping other_deaths_by_sex from export since not 1D array
WARNING: skipping cancers_by_age from export since not 1D array
WARNING: skipping no_dysp_types from export since not 1D array
WARNING: skipping cin1_types from export since not 1D array
WARNING: skipping cin2_types from export since not 1D array
WARNING: skipping cin3_types from export since not 1D array
WARNING: skipping cancer_types from export since not 1D array
WARNING: skipping new_vaccinated from export since not 1D array
WARNING: skipping cum_vaccinated from export since not 1D array
WARNING: skipping n_alive_by_sex from export since not 1D array
WARNING: skipping hpv_prevalence from export since not 1D array
Object saved to /home/docs/checkouts/readthedocs.org/user_builds/institute-for-disease-
↪modeling-hpvsim/checkouts/latest/docs/tutorials/my-sim.xlsx.
    year   t  total_infections  total_cin1s  total_cin2s  total_cin3s  \
0   2000   0               542           80            0            0
1   2001   1               613          136           45            0
2   2002   2               777          184          113            0
3   2003   3               992          207          160           43
4   2004   4              1135          246          203           64
5   2005   5              1211          301          225          110
6   2006   6              1202          286          266          118
7   2007   7              1188          286          304          143
8   2008   8              1197          284          291          165
9   2009   9              1180          290          275          157
10  2010  10              1125          290          286          172
11  2011  11              1043          249          287          166
12  2012  12              1029          243          264          168
13  2013  13              1042          285          237          146
14  2014  14              1134          244          267          165
15  2015  15               666          257          261          131
16  2016  16               653          180          249          153
17  2017  17               645          159          223          153
18  2018  18               606          146          157          156
19  2019  19               555          158          138          116
```

(continues on next page)

(continued from previous page)

```
20  2020  20               632          143          146           87
21  2021  21               599          154          158           80
22  2022  22               660          169          153           84
23  2023  23               670          163          159           89
24  2024  24               696          166          156           74
25  2025  25               700          178          158          109
26  2026  26               659          192          183           83
27  2027  27               739          182          191           90
28  2028  28               742          186          173          109
29  2029  29               702          181          181          113
30  2030  30               785          169          176           88

    total_cins  total_cancers  total_detected_cancers  total_cancer_deaths  \
0           80              0                       0                    0
1          181              0                       0                    0
2          297              0                       0                    0
3          410              0                       0                    0
4          513              0                       0                    0
5          636              0                       0                    0
6          670              0                       0                    0
7          733              0                       0                    0
8          740              0                       0                    0
9          722              1                       0                    0
10         748              7                       0                    0
11         702              6                       0                    0
12         675             10                       0                    0
13         668             11                       0                    0
14         676             18                       0                    0
15         649             18                       0                    0
16         582             22                       0                    0
17         535             19                       0                    0
18         459             17                       0                    0
19         412             27                       0                    1
20         376             14                       0                    7
21         392             13                       0                    2
22         406             20                       0                    5
23         411             18                       0                   11
24         396             22                       0                   13
25         445             15                       0                    9
26         458             21                       0                    9
27         463             12                       0                   14
28         468              3                       0                   12
29         475              9                       0                   19
30         433             16                       0                   18

    ...  cum_cancer_treatments  cum_cancer_treated  detected_cancer_incidence  \
0   ...                       0                   0                          0
1   ...                       0                   0                          0
2   ...                       0                   0                          0
3   ...                       0                   0                          0
4   ...                       0                   0                          0
5   ...                       0                   0                          0
```

(continues on next page)

(continued from previous page)

```
6   ...                     0                  0                       0
7   ...                     0                  0                       0
8   ...                     0                  0                       0
9   ...                     0                  0                       0
10  ...                     0                  0                       0
11  ...                     0                  0                       0
12  ...                     0                  0                       0
13  ...                     0                  0                       0
14  ...                     0                  0                       0
15  ...                     0                  0                       0
16  ...                     0                  0                       0
17  ...                     0                  0                       0
18  ...                     0                  0                       0
19  ...                     0                  0                       0
20  ...                     0                  0                       0
21  ...                     0                  0                       0
22  ...                     0                  0                       0
23  ...                     0                  0                       0
24  ...                     0                  0                       0
25  ...                     0                  0                       0
26  ...                     0                  0                       0
27  ...                     0                  0                       0
28  ...                     0                  0                       0
29  ...                     0                  0                       0
30  ...                     0                  0                       0

    cancer_mortality  n_alive       cdr       cbr  hiv_incidence  \
0           0.000000    20621  0.008390  0.038504              0
1           0.000000    21277  0.007426  0.038257              0
2           0.000000    21968  0.006509  0.037964              0
3           0.000000    22642  0.007906  0.037673              0
4           0.000000    23339  0.007455  0.037320              0
5           0.000000    24070  0.006564  0.036934              0
6           0.000000    24787  0.007544  0.036471              0
7           0.000000    25498  0.008079  0.035964              0
8           0.000000    26247  0.006820  0.035356              0
9           0.000000    26977  0.007636  0.034696              0
10          0.000000    27729  0.006816  0.033936              0
11          0.000000    28433  0.008335  0.033095              0
12          0.000000    29157  0.007442  0.032274              0
13          0.000000    29890  0.006892  0.031415              0
14          0.000000    30602  0.007287  0.030554              0
15          0.000000    31537  0.004186  0.029648              0
16          0.000000    32257  0.006727  0.029048              0
17          0.000000    32970  0.006855  0.028480              0
18          0.000000    33644  0.007906  0.027940              0
19          5.838393    34308  0.008132  0.027515              0
20         40.002286    34995  0.007687  0.027518              0
21         11.165699    35727  0.006970  0.027514              0
22         27.340332    36474  0.006909  0.027526              0
23         58.848705    37209  0.007498  0.027547              0
24         68.212824    37951  0.007641  0.027536              0
```

(continues on next page)

(continued from previous page)

```
25          46.255846    38735  0.007074  0.027546              0
26          45.257970    39525  0.007312  0.027527              0
27          69.081220    40325  0.007340  0.027526              0
28          58.060770    41131  0.007658  0.027546              0
29          90.068737    41965  0.007196  0.027523              0
30          83.561580    42779  0.008088  0.027537              0

    hiv_prevalence  total_hpv_prevalence
0                0              0.031085
1                0              0.044696
2                0              0.059587
3                0              0.076274
4                0              0.089978
5                0              0.098297
6                0              0.102271
7                0              0.102165
8                0              0.102945
9                0              0.100567
10               0              0.097804
11               0              0.092393
12               0              0.088418
13               0              0.086149
14               0              0.086105
15               0              0.071630
16               0              0.062932
17               0              0.057901
18               0              0.052907
19               0              0.049726
20               0              0.049579
21               0              0.047723
22               0              0.047979
23               0              0.047327
24               0              0.047930
25               0              0.047631
26               0              0.046022
27               0              0.047216
28               0              0.047069
29               0              0.045300
30               0              0.045817

[31 rows x 71 columns]
```

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

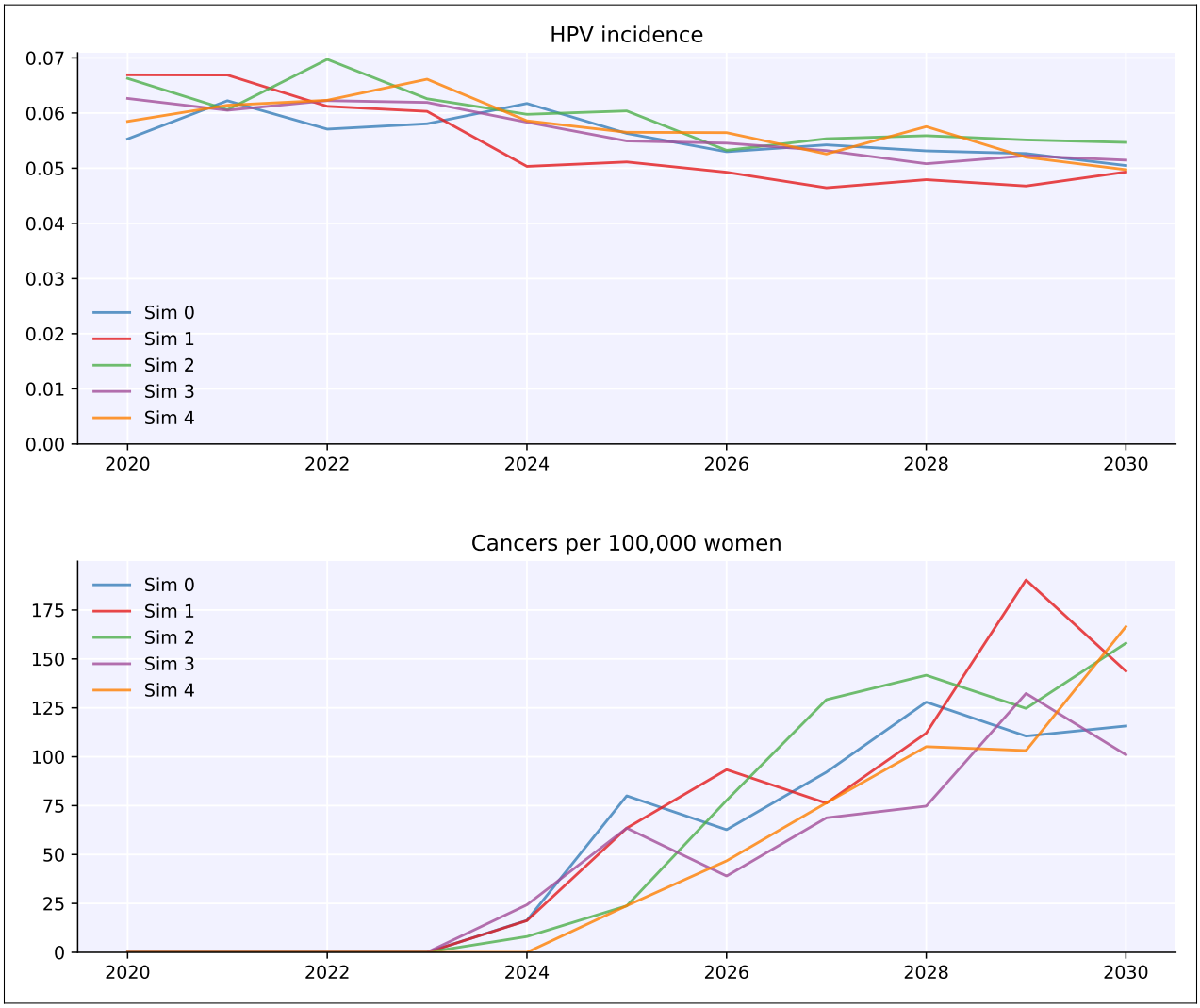
The most common way to run multiple simulations is with the *MultiSim* object. As the name suggests, this is a relatively simple container for a number of sims. However, it contains powerful methods for plotting, statistics, and running all the sims in parallel.

Running one sim with uncertainty

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

[1]:
```
import hpvsim as hpv
hpv.options(jupyter=True, verbose=0)

sim = hpv.Sim()
msim = hpv.MultiSim(sim)
msim.run(n_runs=5)
msim.plot();
```

```
HPVsim 0.4.0 (2022-11-16) - © 2022 by IDM
No genotypes provided, will assume only simulating HPV16 by default
No genotypes provided, will assume only simulating HPV16 by default
No genotypes provided, will assume only simulating HPV16 by defaultNo genotypes provided,
↪ will assume only simulating HPV16 by default

No genotypes provided, will assume only simulating HPV16 by default
```



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.

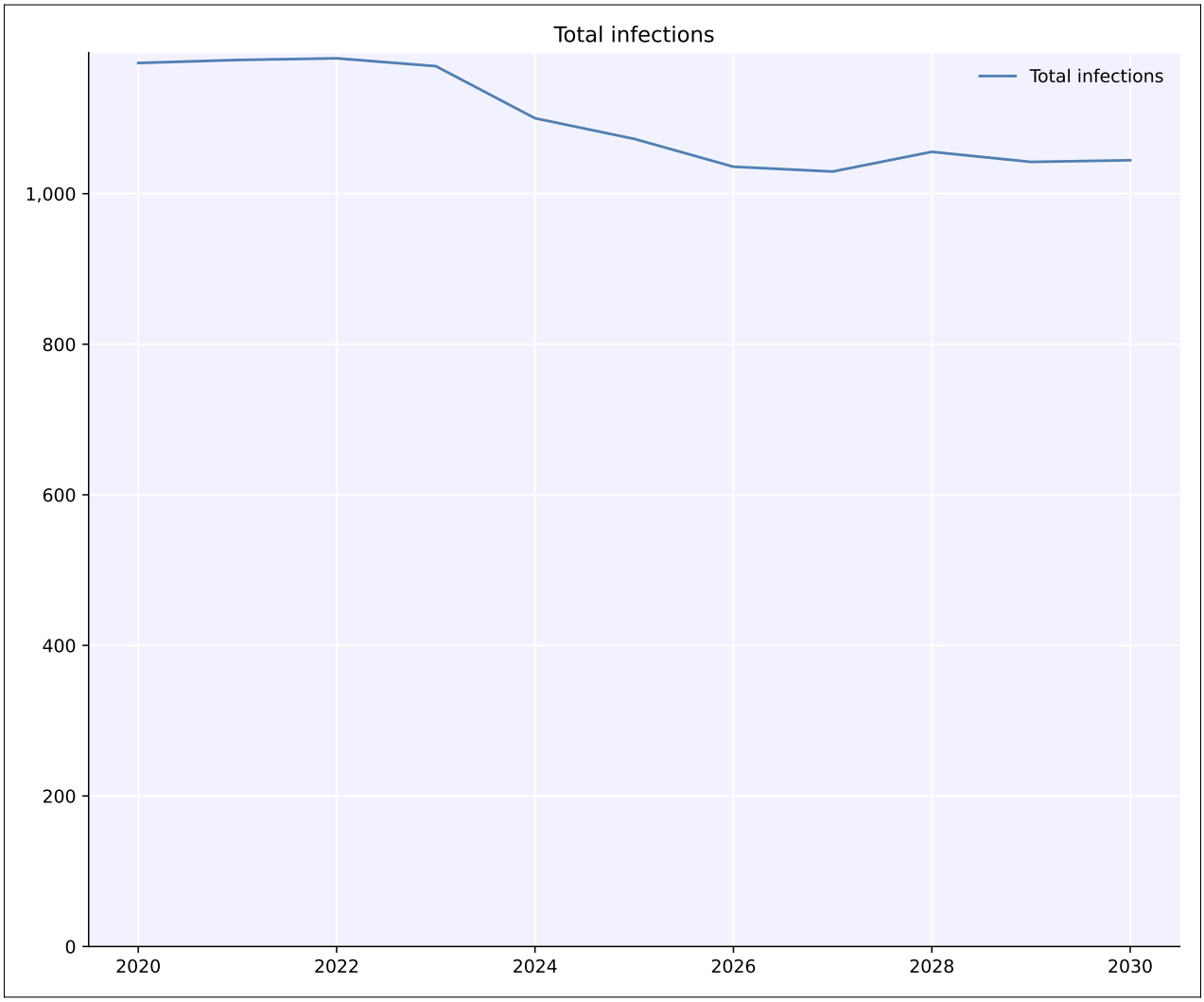
By default, the multisim simply plots each simulation. These simulations are stored in the `sims` attribute, which is just a simple list of sims:

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

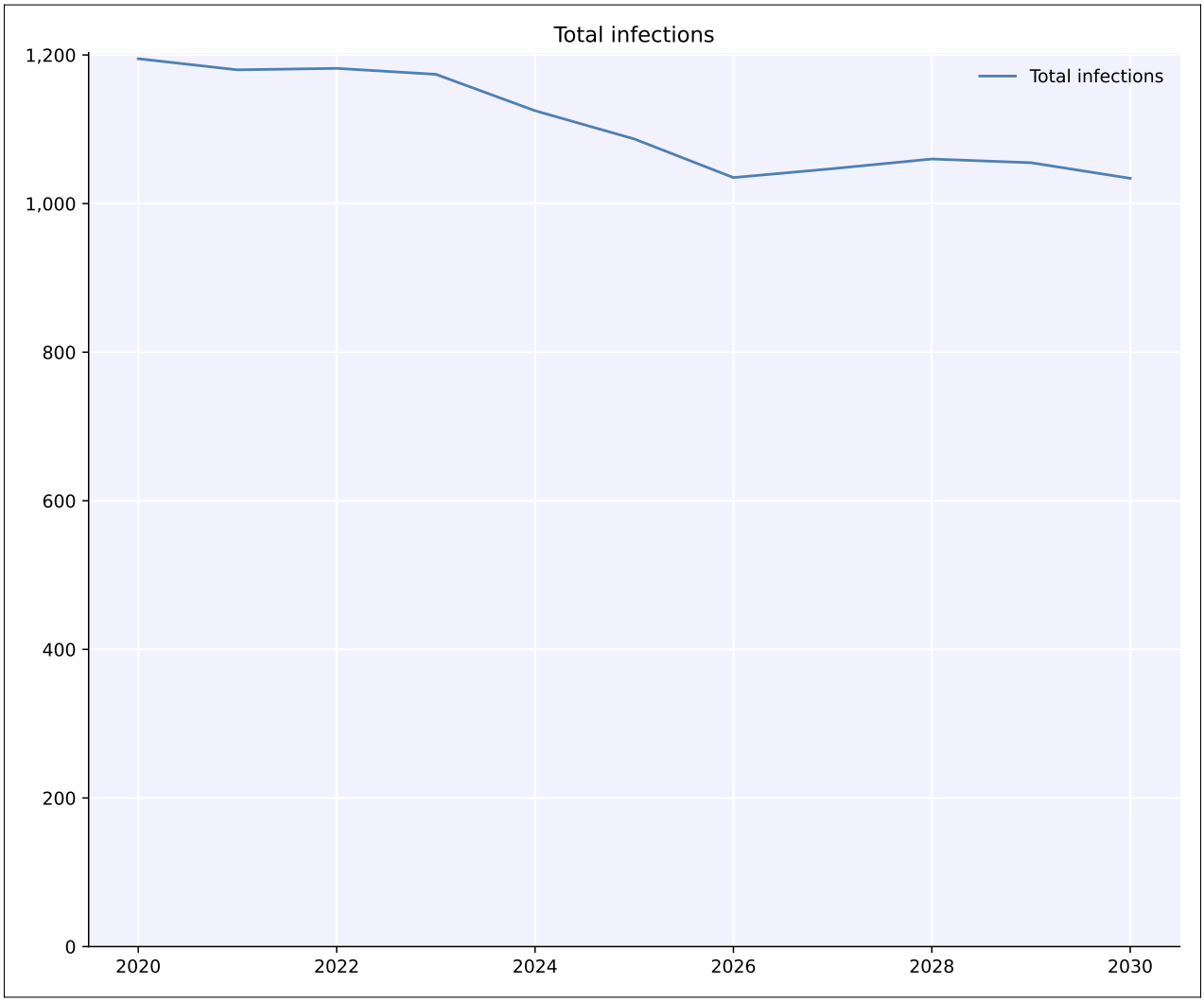
```
Sim("Sim 0"; 2015.0 to 2030.0; pop: 20000 default; epi: 1034, 16)
Sim("Sim 1"; 2015.0 to 2030.0; pop: 20000 default; epi: 1016, 20)
Sim("Sim 2"; 2015.0 to 2030.0; pop: 20000 default; epi: 1101, 22)
Sim("Sim 3"; 2015.0 to 2030.0; pop: 20000 default; epi: 1053, 14)
Sim("Sim 4"; 2015.0 to 2030.0; pop: 20000 default; epi: 1018, 23)
```

However, you often don't care about the individual sims (especially when you run the same parameters with different random seeds); 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:

[3]:
```
msim.mean()
msim.plot('total_infections');
```

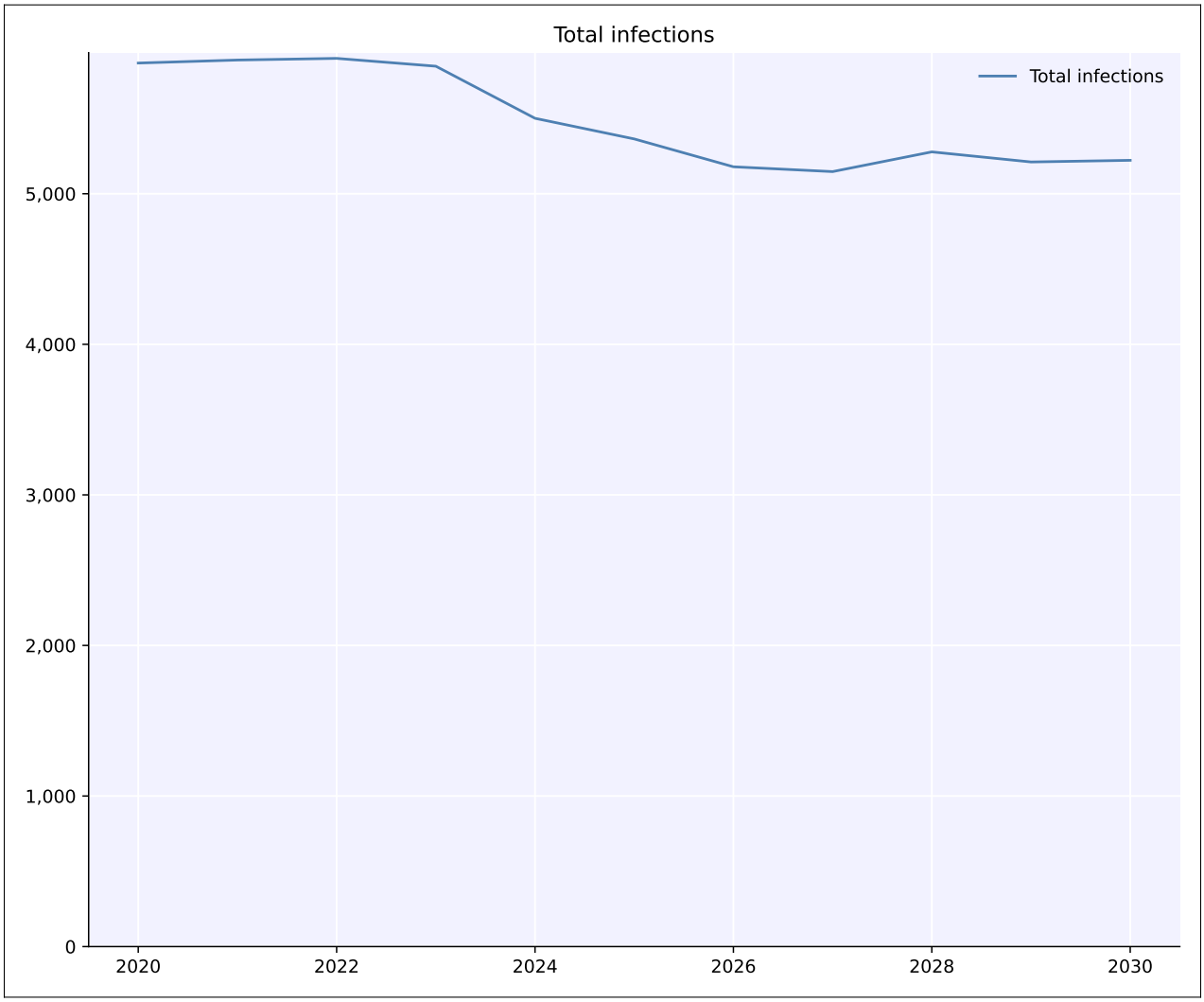


[4]:
```
msim.median()
msim.plot('total_infections');
```



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('total_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 the relative transmissibility of people with high-grade (CIN3) lesions:

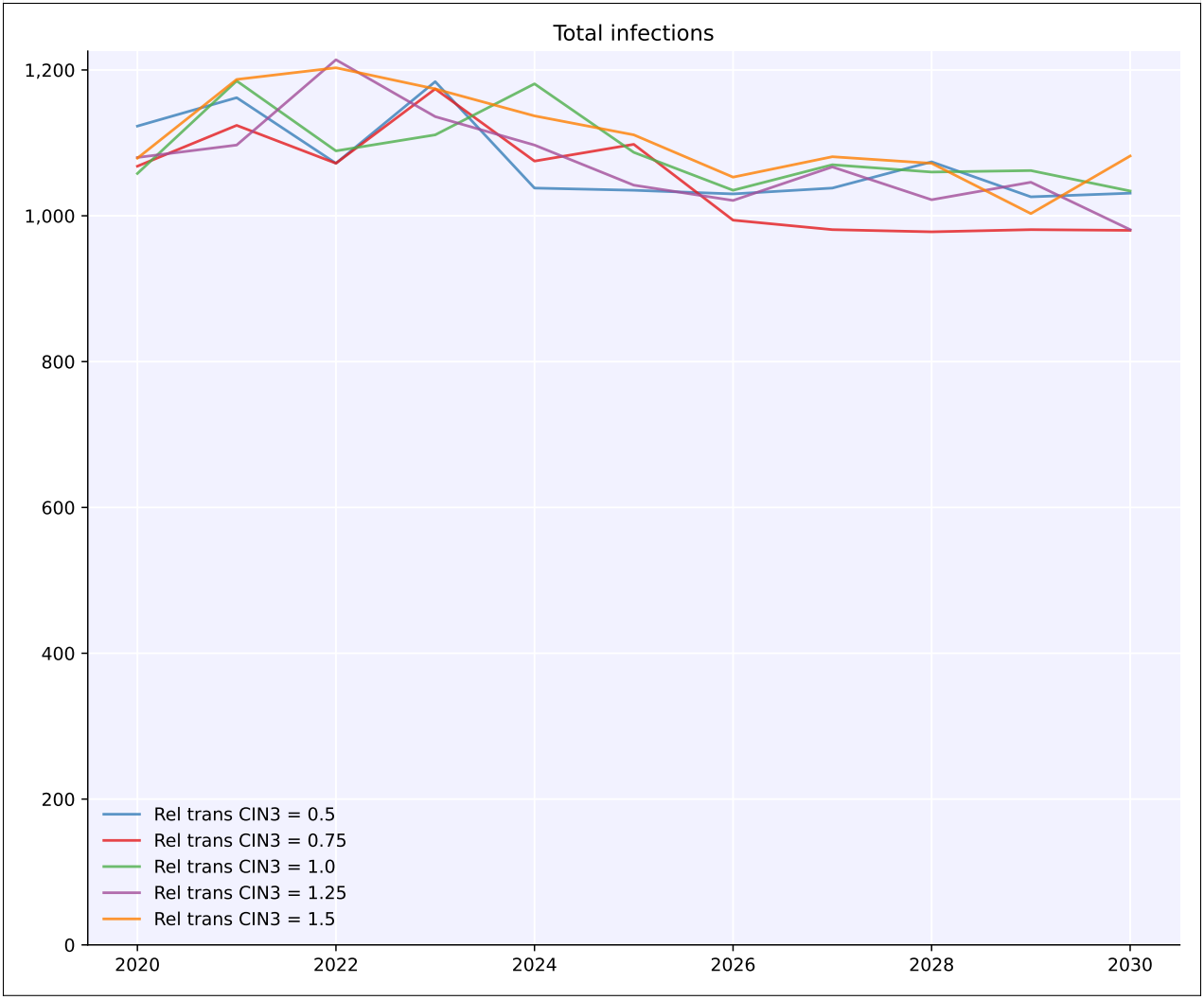
[6]:
```
import numpy as np

rel_trans_cin3_vals = np.linspace(0.5, 1.5, 5) # Sweep from 0.5 to 1.5 with 5 values
sims = []
for rel_trans_cin3 in rel_trans_cin3_vals:
    sim = hpv.Sim(rel_trans_cin3=rel_trans_cin3, label=f'Rel trans CIN3 = {rel_trans_
↪cin3}')
```

(continues on next page)

(continued from previous page)

```
    sims.append(sim)
msim = hpv.MultiSim(sims)
msim.run()
msim.plot('total_infections');
```

```
No genotypes provided, will assume only simulating HPV16 by defaultNo genotypes provided,
↪ will assume only simulating HPV16 by default

No genotypes provided, will assume only simulating HPV16 by defaultNo genotypes provided,
↪ will assume only simulating HPV16 by default

No genotypes provided, will assume only simulating HPV16 by default
```



As you would expect, the more transmissible people with CIN3s are, the more infections we get.

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

[7]:
```
def custom_vx(sim):
    if sim.yearvec[sim.t] == 2000:
```

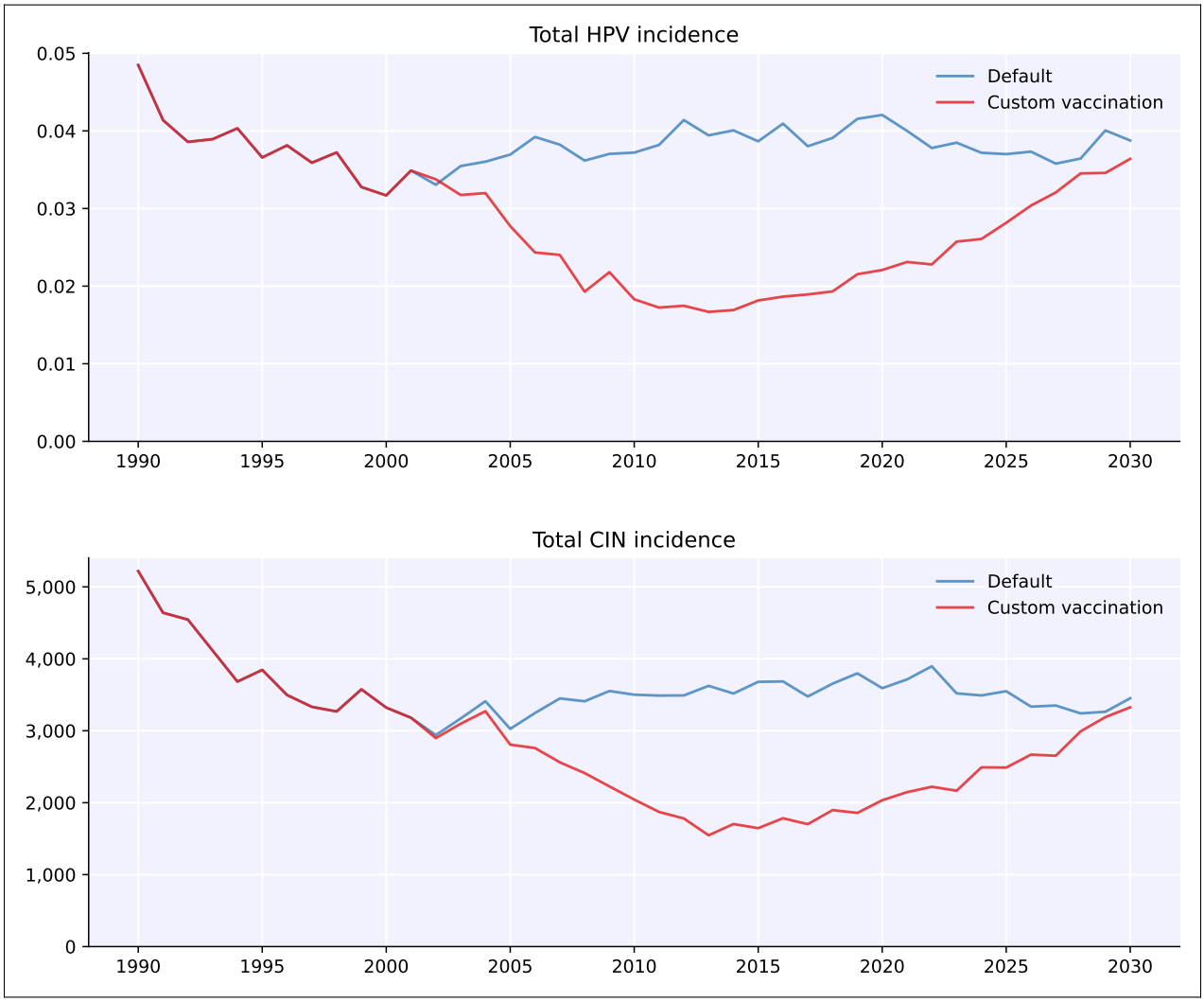
(continues on next page)

(continued from previous page)

```
        target_group = (sim.people.age>9) * (sim.people.age<14)
        sim.people.peak_imm[0, target_group] = 1

pars = dict(
    location = 'tanzania', # Use population characteristics for Japan
    n_agents = 10e3, # Have 50,000 people total in the population
    start = 1980, # Start the simulation in 1980
    n_years = 50, # Run the simulation for 50 years
    burnin = 10, # Discard the first 20 years as burnin period
    verbose = 0, # Do not print any output
)

# Running with multisims -- see Tutorial 3
s1 = hpv.Sim(pars, label='Default')
s2 = hpv.Sim(pars, interventions=custom_vx, label='Custom vaccination')
hpv.parallel(s1, s2).plot(['total_hpv_incidence', 'total_cin_incidence']);
```

```
Loading location-specific demographic data for "tanzania"
Loading location-specific demographic data for "tanzania"
No genotypes provided, will assume only simulating HPV16 by default
No genotypes provided, will assume only simulating HPV16 by default
```

```
/home/docs/checkouts/readthedocs.org/user_builds/institute-for-disease-modeling-hpvsim/
↪envs/latest/lib/python3.9/site-packages/hpvsim/data/loaders.py:209: FutureWarning: The␣
↪default value of numeric_only in DataFrameGroupBy.sum is deprecated. In a future␣
↪version, numeric_only will default to False. Either specify numeric_only or select␣
↪only columns which should be valid for the function.
  dd = full_df.groupby("Time").sum()["PopTotal"]
/home/docs/checkouts/readthedocs.org/user_builds/institute-for-disease-modeling-hpvsim/
↪envs/latest/lib/python3.9/site-packages/hpvsim/data/loaders.py:209: FutureWarning: The␣
↪default value of numeric_only in DataFrameGroupBy.sum is deprecated. In a future␣
↪version, numeric_only will default to False. Either specify numeric_only or select␣
↪only columns which should be valid for the function.
  dd = full_df.groupby("Time").sum()["PopTotal"]
```



Warning: 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 `rel_trans_cin3` value, but then merge multisims for different `rel_trans_cin3` values together into one multisim:

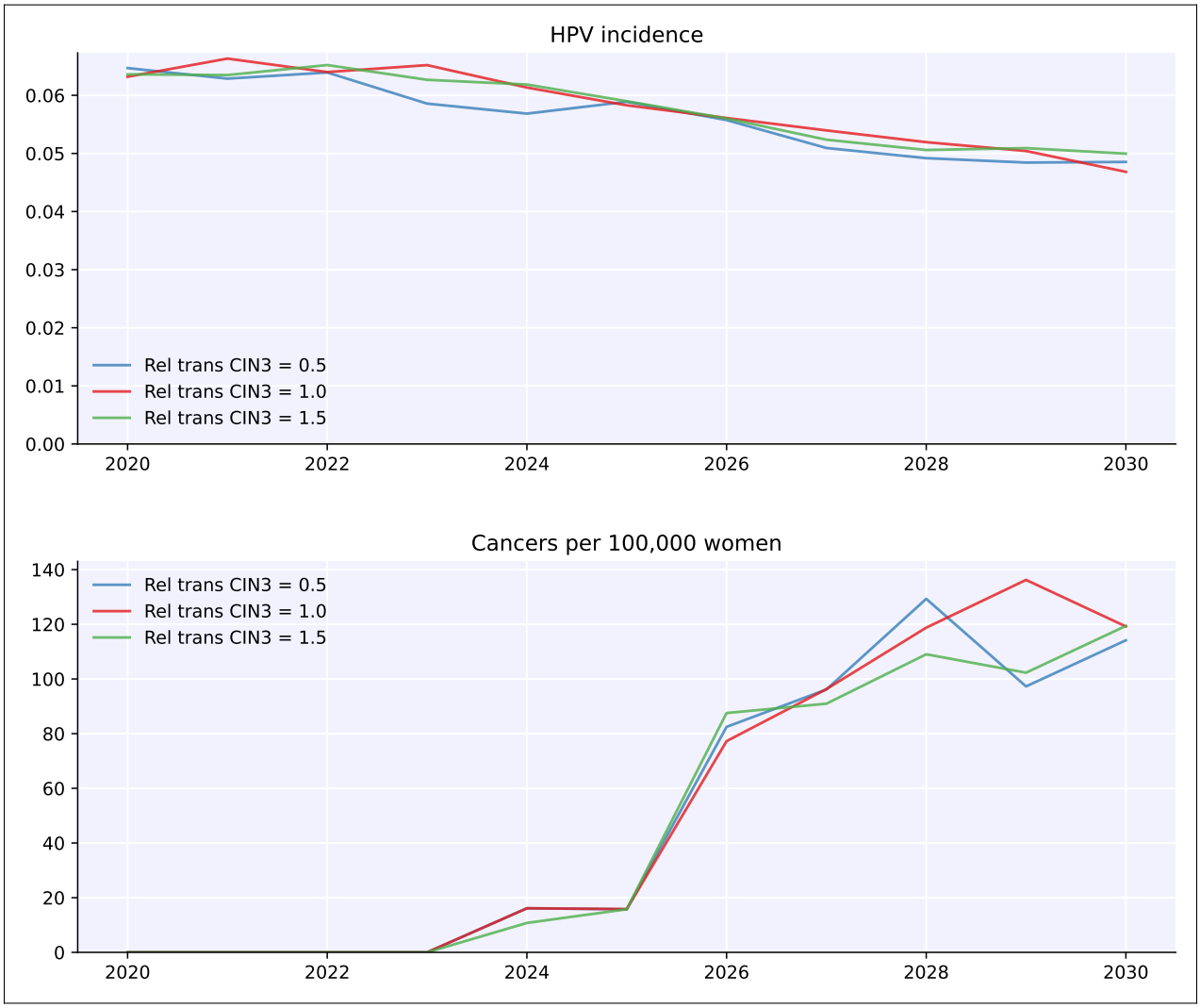
[8]:
```
n_sims = 3
rel_trans_cin3_vals = [0.5, 1.0, 1.5]

msims = []
for rel_trans_cin3 in rel_trans_cin3_vals:
```

(continues on next page)

(continued from previous page)

```
    sims = []
    for s in range(n_sims):
        sim = hpv.Sim(n_agents=10e3, rel_trans_cin3=rel_trans_cin3, rand_seed=s, label=f
↪'Rel trans CIN3 = {rel_trans_cin3}')
        sims.append(sim)
    msim = hpv.MultiSim(sims)
    msim.run()
    msim.mean()
    msims.append(msim)

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

```
No genotypes provided, will assume only simulating HPV16 by defaultNo genotypes provided,
↪ will assume only simulating HPV16 by default

No genotypes provided, will assume only simulating HPV16 by default
No genotypes provided, will assume only simulating HPV16 by defaultNo genotypes provided,
↪ will assume only simulating HPV16 by default

No genotypes provided, will assume only simulating HPV16 by default
No genotypes provided, will assume only simulating HPV16 by defaultNo genotypes provided,
↪ will assume only simulating HPV16 by default

No genotypes provided, will assume only simulating HPV16 by default
```



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.

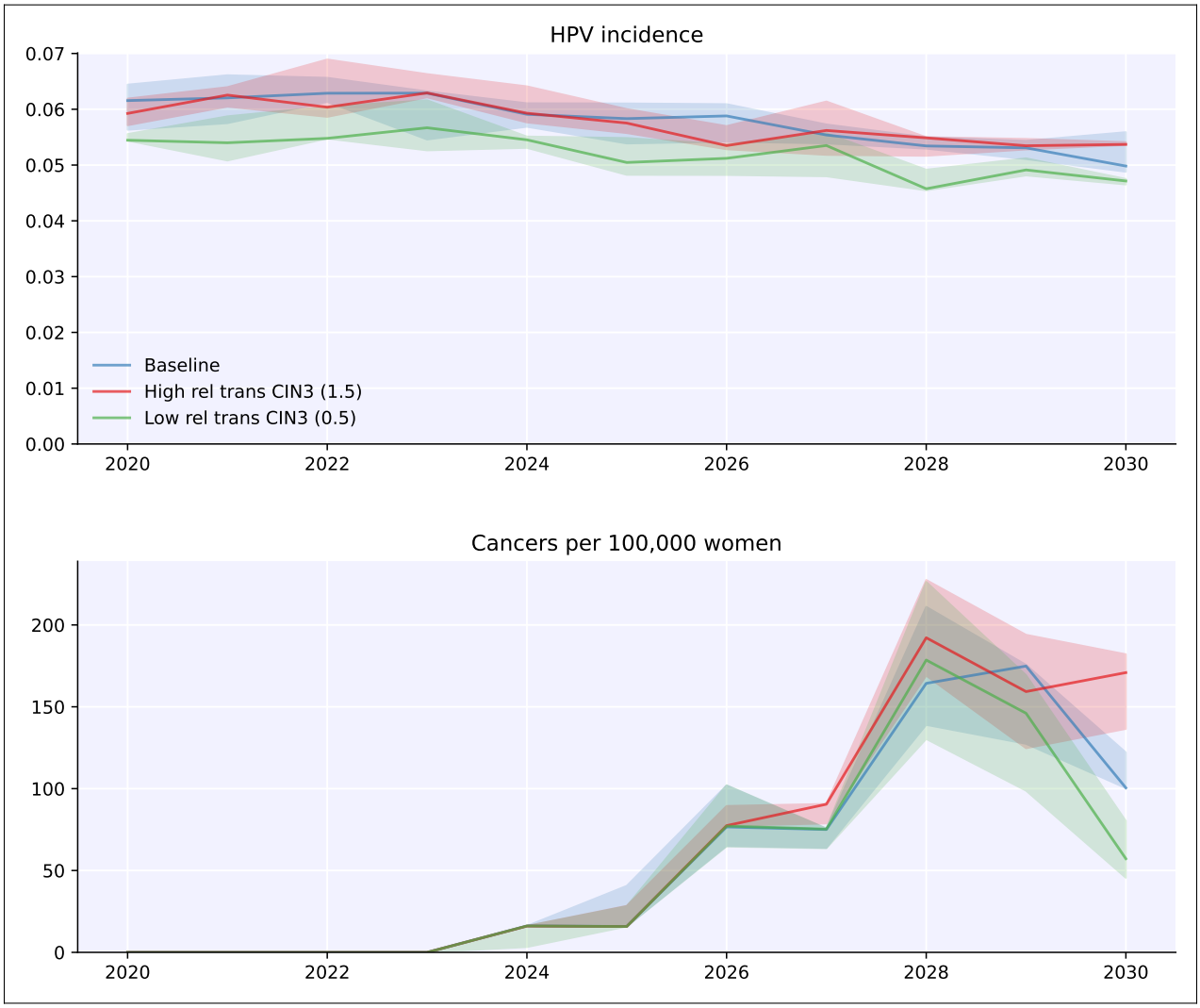
[9]:
```
# Set base parameters -- these will be shared across all scenarios
basepars = {'n_agents':10e3}

# Configure the settings for each scenario
scenarios = {'baseline': {
              'name':'Baseline',
              'pars': {}
```

(continues on next page)

(continued from previous page)

```
            },
          'high_rel_trans_cin3': {
            'name':'High rel trans CIN3 (1.5)',
            'pars': {
                'rel_trans_cin3': 1.5,
                }
            },
          'low_rel_trans_cin3': {
            'name':'Low rel trans CIN3 (0.5)',
            'pars': {
                'rel_trans_cin3': 0.5,
                }
            },
           }

# Run and plot the scenarios
scens = hpv.Scenarios(basepars=basepars, scenarios=scenarios)
scens.run()
scens.plot();
```

```
No genotypes provided, will assume only simulating HPV16 by default
```



1.2.4 T4 - People, populations, and networks

This tutorial gives a brief introduction to people, populations, and contact layers.

Click here to open an interactive version of this notebook.

1.2.4.1 Demographic data

HPVsim includes pre-downloaded demographic data for almost all countries, including:

- Population sizes from 1950-2100 from the UN's World Population Projections;
- Birth rates from 1950-2100 from the World Bank;
- Age- and sex-specific mortality rates from 1950-2100 from the UN's World Population Projections.

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 `hpv.data.show_locations()`.

1.2.4.2 People and contact network layers

Agents in HPVsim 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).

HPV transmits via sexual contact, and in HPVsim this is represented by sexual networks that allow agents to interact with one another. For the moment, HPVsim only models heterosexual partnerships. The sexual contact networks in HPVsim can have multiple *contact layers*, with each layer having different properties that characterize sexual contact, including the duration of the contact, age mixing preferences of partners, etc. HPVsim comes with two options for the sexual network:

- The *random* option has a single contact layer. The number of partners that each agent has is Poisson distributed with a mean of 1.
- The *default* option has 3 contact layers, representing marital, casual, and one-off partnership types.

1.2.5 T5 - Using interventions

Interventions are one of the most critical parts of HPVsim. 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 Products and interventions

HPVsim contains *products*, which can be thought of as the actual test, diagnostic, treatment, or vaccine product being used, as well as *interventions*, which are responsible for delivering the products to the population. Information about the default products included with HPVsim (e.g. attributes like test positivity and efficacy) are available in the `hpvsim/data/products_*.csv` files. Specifically:

- Screening products (VIA, HPV testing, and HPV16/18 testing): `hpvsim/data/products_dx.csv`
- Triage products (VIA triage, assignment to treatment): `hpvsim/data/products_dx.csv`
- Treatment products (ablation, excision, and therapeutic vaccines): `hpvsim/data/products_tx.csv`
- Prophylactic vaccine products (bivalent, quadrivalent, nonavalent): `hpvsim/data/products_vx.csv` It's also possible to make a custom product, e.g.

[1]:
```
import hpvsim as hpv
import pandas as pd
my_treatment_data = pd.DataFrame({'name':'new_tx', 'state':['precin','cin1','cin2','cin3
↪','cancerous'],'genotype':'all','efficacy':[.2,.3,.3,.4,.4]})
my_treatment = hpv.tx(df=my_treatment_data)
```

```
HPVsim 0.4.0 (2022-11-16) - © 2022 by IDM
```

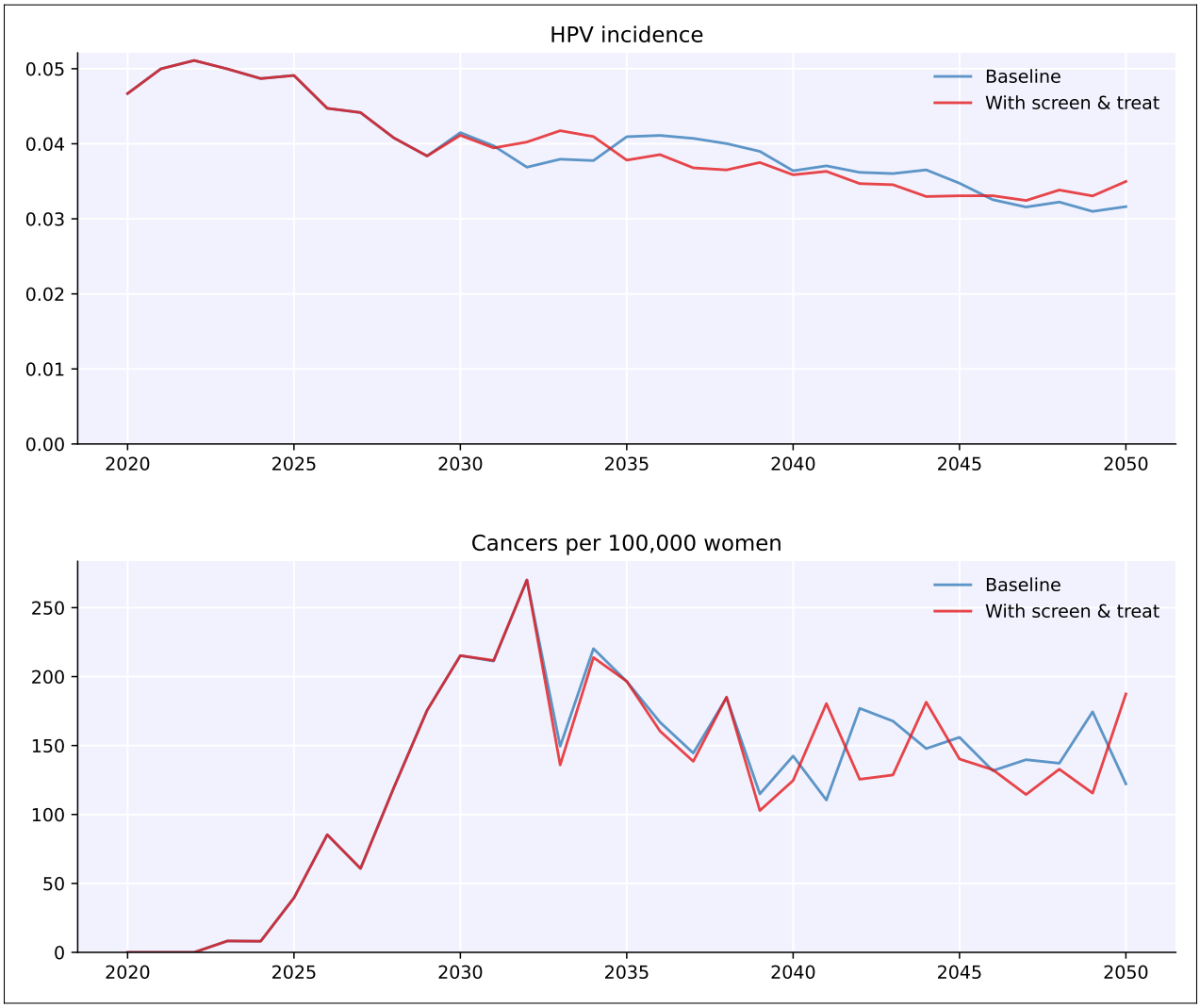
Custom products can be made for diagnostics and vaccination in the same way. The efficacy of some products varies by genotype, in which case efficacy values for each genotype can be entered as separate dataframe rows.

Most of the time, it isn't necessary to create your own products if you just want to use one of the standard options. When setting up screening, triage, or treatment interventions, it's possible to pass in a string that will create a standard default product.

1.2.5.2 Screening and treatment interventions

Screening and treatment is implemented in HPVsim as a flexible set of interventions that can be mixed and matched. By specifying how each of the components link together, it's possible to create quite complex algorithms. This is illustrated in the following example:

[2]:
```
# Define a series of interventions to screen, triage, assign treatment, and administer␣
↪treatment
prob = 0.02
screen      = hpv.routine_screening(start_year=2030, prob=prob, product='via', label=
↪'screen') # Routine screening
to_triage   = lambda sim: sim.get_intervention('screen').outcomes['positive'] # Define␣
↪who's eligible for triage
triage      = hpv.routine_triage(eligibility=to_triage, prob=prob, product='via_triage',␣
↪label='triage') # Triage people
to_treat    = lambda sim: sim.get_intervention('triage').outcomes['positive'] # Define␣
↪who's eligible to be assigned treatment
assign_tx   = hpv.routine_triage(eligibility=to_treat, prob=prob, product='tx_assigner',␣
↪label='assign_tx') # Assign treatment
to_ablate   = lambda sim: sim.get_intervention('assign_tx').outcomes['ablation'] #␣
↪Define who's eligible for ablation treatment
ablation    = hpv.treat_num(eligibility=to_ablate, prob=prob, product='ablation') #␣
↪Administer ablation
to_excise   = lambda sim: sim.get_intervention('assign_tx').outcomes['excision'] #␣
↪Define who's eligible for excision
excision    = hpv.treat_delay(eligibility=to_excise, prob=prob, product='excision') #␣
↪Administer excision

# Define the parameters
pars = dict(
    n_agents      = 20e3,       # Population size
    n_years       = 35,         # Number of years to simulate
    verbose       = 0,          # Don't print details of the run
    rand_seed     = 2,          # Set a non-default seed
    genotypes     = [16, 18],   # Include the two genotypes of greatest general interest
)

# Create the sim with and without interventions
orig_sim = hpv.Sim(pars, label='Baseline')
sim = hpv.Sim(pars, interventions = [screen, triage, assign_tx, ablation, excision],␣
↪label='With screen & treat')

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



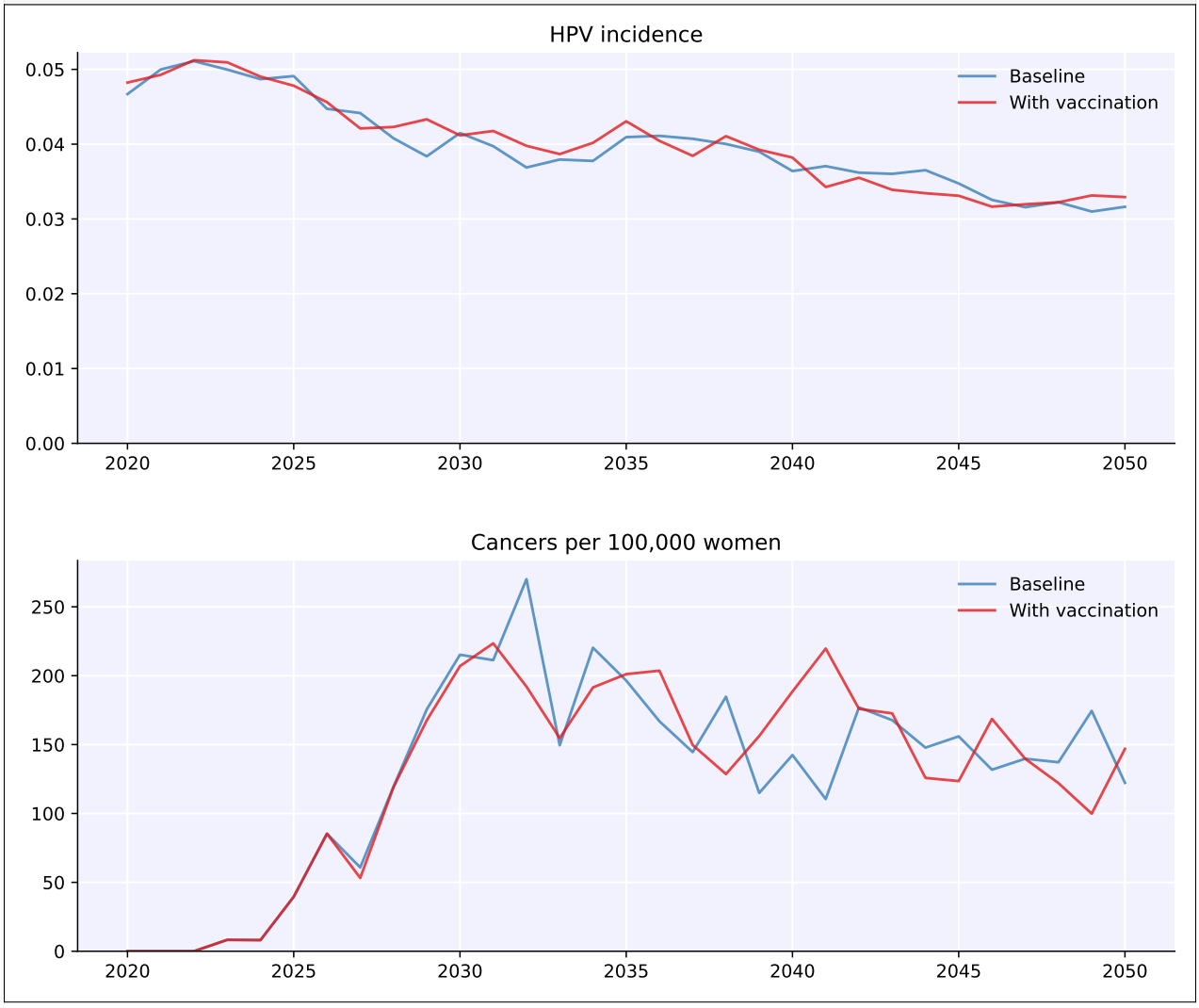
A few things to note here:

- By default, interventions are shown with vertical dashed lines. You can turn this off by passing `do_plot=False` to the intervention.
- 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.
- Several of the interventions above are defined as routine interventions, e.g. `hpv.routine_screening()` and `hpv.routine_triage()`. In general, most interventions exist as both routine and campaign versions. The difference between the two comes down to how the dates are interpreted:
- `hpv.routine_screening(start=2020, end=2030, prob=0.2)` implies that the intervention will be in place each year between 2020-2030;
- `hpv.campaign_screening(years=[2020,2030], prob=0.2)` implies that the intervention will be delivered twice: once in 2020 and once in 2030.

1.2.5.3 Prophylactic vaccination

Prophylactic vaccination within HPVsim simply targets a vaccine product towards a subset of the population.

[3]:
```
vx = hpv.routine_vx(prob=prob, start_year=2020, age_range=[9,10], product='bivalent')

# Create the sim with and without interventions
orig_sim = hpv.Sim(pars, label='Baseline')
sim = hpv.Sim(pars, interventions = vx, label='With vaccination')

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



Note that probabilities passed to interventions are annual probabilities, not total.

1.2.5.4 Therapeutic vaccination

Therapeutic vaccination can be included in a few different formats/use cases:

- As a part of a screen & treat algorithm
- As a mass vaccination without screening.

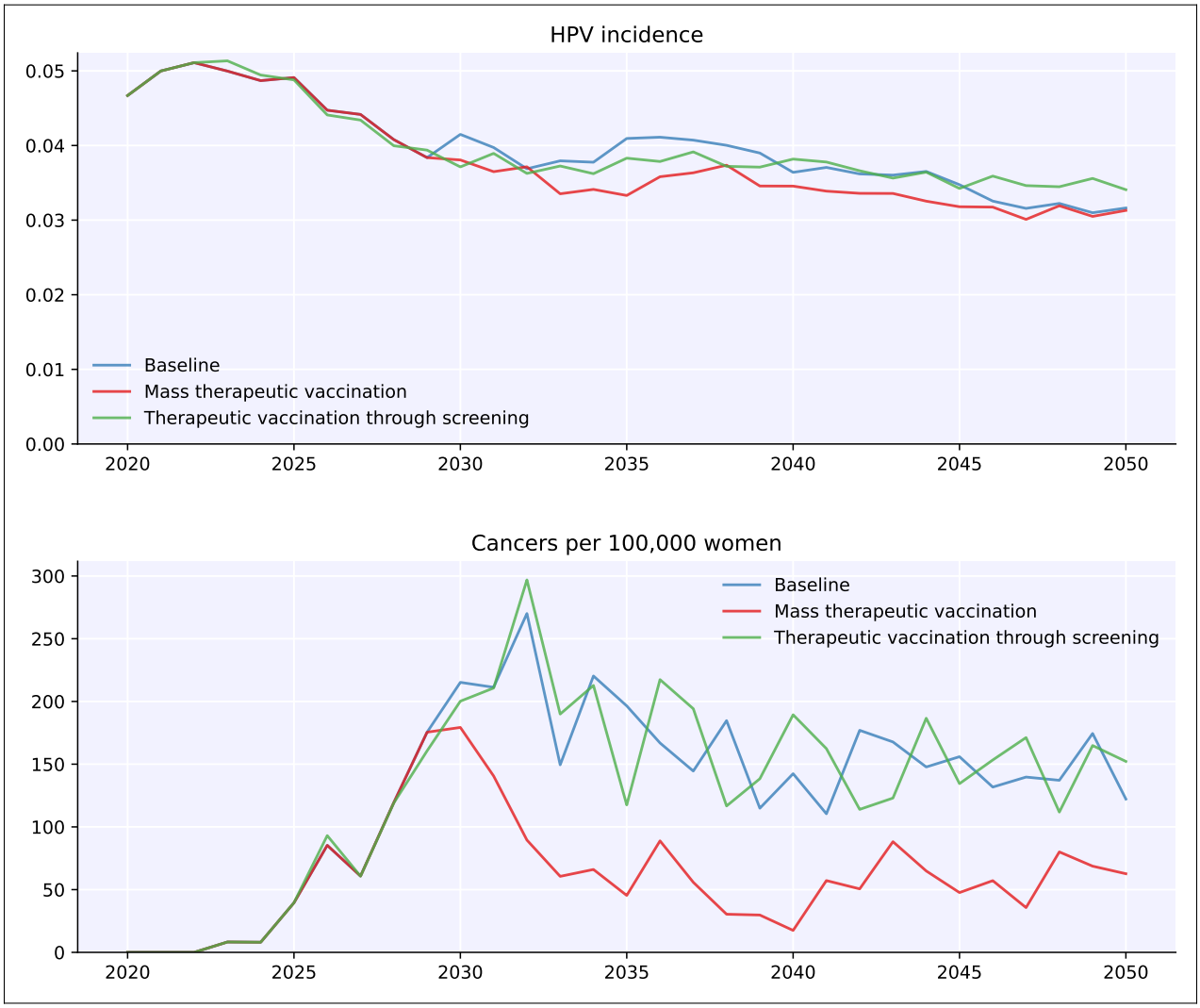
The following examples illustrate this:

[4]:
```
import numpy as np
import hpvsim as hpv

# Define mass therapeutic vaccination:
campaign_txvx_dose1 = hpv.campaign_txvx(prob = 0.9, years = 2030, age_range = [30,50],␣
↪product = 'txvx1', label = 'campaign txvx')
second_dose_eligible = lambda sim: (sim.people.txvx_doses == 1) | (sim.t > (sim.people.
↪date_tx_vaccinated + 0.5 / sim['dt']))
campaign_txvx_dose2 = hpv.campaign_txvx(prob = 0.7, years=[2030,2031], age_range=[30,␣
↪70], product = 'txvx2', eligibility = second_dose_eligible, label = 'campaign txvx 2nd␣
↪dose')
routine_txvx_dose1 = hpv.routine_txvx(prob = 0.9, start_year = 2031, age_range = [30,31],
↪ product = 'txvx2',label = 'routine txvx')
second_dose_eligible = lambda sim: (sim.people.txvx_doses == 1) | (sim.t > (sim.people.
↪date_tx_vaccinated + 0.5 / sim['dt']))
routine_txvx_dose2 = hpv.routine_txvx(prob = 0.8, start_year = 2031, age_range = [30,31],
↪ product = 'txvx1', eligibility=second_dose_eligible, label = 'routine txvx 2nd dose')
mass_vx_intvs = [campaign_txvx_dose1, campaign_txvx_dose2, routine_txvx_dose1, routine_
↪txvx_dose2]
for intv in mass_vx_intvs: intv.do_plot=False

# Define therapeutic vaccination within screen and treat
campaign_txvx_dose1 = hpv.campaign_txvx(prob = 0.9, years = 2030, age_range = [30,50],␣
↪product = 'txvx1', label = 'campaign txvx')
second_dose_eligible = lambda sim: (sim.people.txvx_doses == 1) | (sim.t > (sim.people.
↪date_tx_vaccinated + 0.5 / sim['dt']))
campaign_txvx_dose2 = hpv.campaign_txvx(prob = 0.7, years=[2030,2031], age_range=[30,␣
↪70], product = 'txvx2', eligibility = second_dose_eligible, label = 'campaign txvx 2nd␣
↪dose')
routine_txvx_dose1 = hpv.routine_txvx(prob = 0.9, start_year = 2031, age_range = [30,31],
↪ product = 'txvx2',label = 'routine txvx')
second_dose_eligible = lambda sim: (sim.people.txvx_doses == 1) | (sim.t > (sim.people.
↪date_tx_vaccinated + 0.5 / sim['dt']))
routine_txvx_dose2 = hpv.routine_txvx(prob = 0.8, start_year = 2031, age_range = [30,31],
↪ product = 'txvx1', eligibility=second_dose_eligible, label = 'routine txvx 2nd dose')
mass_vx_intvs = [campaign_txvx_dose1, campaign_txvx_dose2, routine_txvx_dose1, routine_
↪txvx_dose2]
for intv in mass_vx_intvs: intv.do_plot=False


# Screen, triage, assign treatment, treat
screen_eligible = lambda sim: np.isnan(sim.people.date_screened) | (sim.t > (sim.people.
↪date_screened + 5 / sim['dt']))
routine_screen = hpv.routine_screening(start_year=2023, product='via', prob=0.1,␣
↪eligibility=screen_eligible, age_range=[30, 50], label='routine screening')
```

(continues on next page)

(continued from previous page)

```
screened_pos = lambda sim: sim.get_intervention('routine screening').outcomes['positive
↪'] # Get those who screen positive
pos_screen_assesser = hpv.routine_triage(start_year=2023, product = 'txvx_assigner',␣
↪prob = 1.0, annual_prob=False, eligibility = screened_pos, label = 'txvx assigner') #␣
↪Offer TxVx or refer them for further testing
txvx_eligible = lambda sim: sim.get_intervention('txvx assigner').outcomes['txvx'] # Get␣
↪people who've been classified as txvx eligible based on the positive screen assessment
deliver_txvx = hpv.linked_txvx(prob = 0.8, product = 'txvx1', eligibility = txvx_
↪eligible, label = 'txvx') # Deliver txvx to them

screen_vx_intv = [routine_screen, pos_screen_assesser, deliver_txvx]
for intv in screen_vx_intv: intv.do_plot=False

sim0 = hpv.Sim(pars=pars, label='Baseline')
sim1 = hpv.Sim(pars=pars, interventions=mass_vx_intvs, label='Mass therapeutic␣
↪vaccination')
sim2 = hpv.Sim(pars=pars, interventions=screen_vx_intv, label='Therapeutic vaccination␣
↪through screening')

# Run and plot
msim = hpv.parallel(sim0, sim1, sim2)
msim.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 Results by age

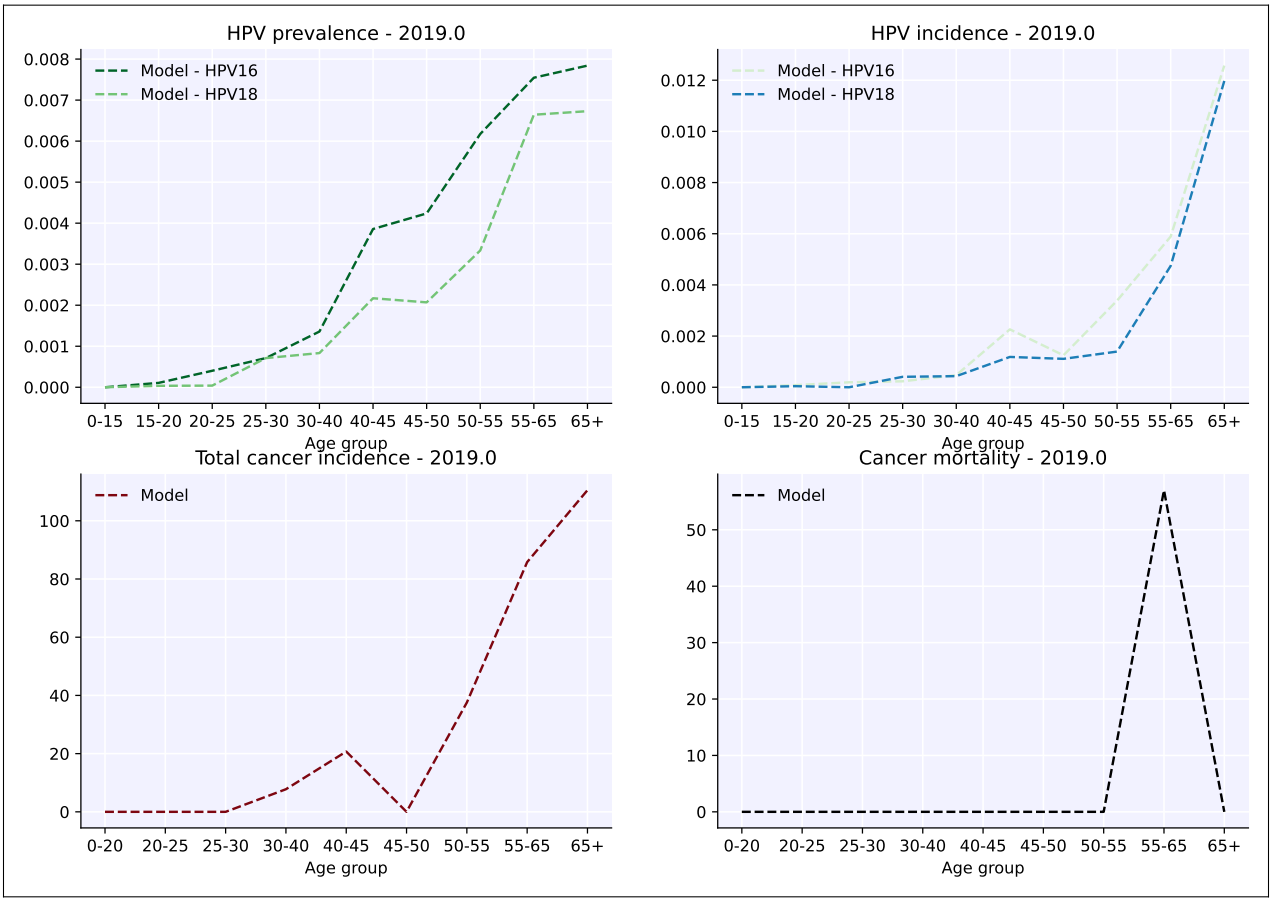
By far the most common reason to use an analyzer is to report results by age. The results in `sim.results` are aggregated over all ages, whereas data on cervical cancers are generally reported by age. Age-specific outputs can be customized using an analyzer to match the age bins of the data. The following example shows how to set this up:

[1]:
```
import numpy as np
import sciris as sc
import hpvsim as hpv

# Create some parameters, setting beta (per-contact transmission probability) higher
# to create more cancers for illutration
pars = dict(beta=0.5, n_agents=50e3, start=1970, n_years=50, dt=1., location='tanzania')

# Also set initial HPV prevalence to be high, again to generate more cancers
pars['init_hpv_prev'] = {
    'age_brackets'  : np.array([  12,   17,   24,   34,  44,   64,    80, 150]),
    'm'             : np.array([ 0.0, 0.75, 0.9, 0.45, 0.1, 0.05, 0.005, 0]),
    'f'             : np.array([ 0.0, 0.75, 0.9, 0.45, 0.1, 0.05, 0.005, 0]),
}

# Create the age analyzers.
az1 = hpv.age_results(
    result_keys=sc.objdict(
        hpv_prevalence=sc.objdict( # The keys of this dictionary are any results you␣
↪want by age, and can be any key of sim.results
            timepoints=['2019'], # List the years that you want to generate results for
            edges=np.array([0., 15., 20., 25., 30., 40., 45., 50., 55., 65., 100.]),
        ),
        hpv_incidence=sc.objdict(
            timepoints=['2019'],
            edges=np.array([0., 15., 20., 25., 30., 40., 45., 50., 55., 65., 100.]),
        ),
        total_cancer_incidence=sc.objdict(
            timepoints=['2019'],
            edges=np.array([0.,20.,25.,30.,40.,45.,50.,55.,65.,100.]),
        ),
        cancer_mortality=sc.objdict(
            timepoints=['2019'],
            edges=np.array([0., 20., 25., 30., 40., 45., 50., 55., 65., 100.]),
        )
    )
)

sim = hpv.Sim(pars, genotypes=[16, 18], analyzers=[az1])
sim.run()
a = sim.get_analyzer()
a.plot();
```

```
HPVsim 0.4.0 (2022-11-16) - © 2022 by IDM
Loading location-specific demographic data for "tanzania"
Initializing sim with 50000 agents
Loading location-specific data for "tanzania"
```

```
/home/docs/checkouts/readthedocs.org/user_builds/institute-for-disease-modeling-hpvsim/
↪envs/latest/lib/python3.9/site-packages/hpvsim/data/loaders.py:209: FutureWarning: The␣
↪default value of numeric_only in DataFrameGroupBy.sum is deprecated. In a future␣
↪version, numeric_only will default to False. Either specify numeric_only or select␣
↪only columns which should be valid for the function.
  dd = full_df.groupby("Time").sum()["PopTotal"]
```

```
  Running 1970.0 ( 0/51) (0.96 s)  ---------------------- 2%
  Running 1980.0 (10/51) (1.55 s)  ••••------------------ 22%
  Running 1990.0 (20/51) (2.28 s)  ••••••••------------ 41%
  Running 2000.0 (30/51) (3.19 s)  ••••••••••••-------- 61%
  Running 2010.0 (40/51) (4.39 s)  ••••••••••••••••---- 80%
  Running 2020.0 (50/51) (6.22 s)  •••••••••••••••••••• 100%

Simulation summary:
       66,936 total infections
       14,726 total cin1s
       17,671 total cin2s
        9,371 total cin3s
       41,768 total cins
        1,071 total cancers
            0 total cancer detections
        1,071 total cancer deaths
            0 total detected cancer deaths
       14,458 total reinfections
       17,939 total reactivations
            0 total hiv infections
         0.06 total hpv incidence (/100)
           48 total cin1 incidence (/100,000)
           58 total cin2 incidence (/100,000)
           31 total cin3 incidence (/100,000)
          136 total cin incidence (/100,000)
            3 total cancer incidence (/100,000)
         0.31 total hpv prevalence (/100)
```



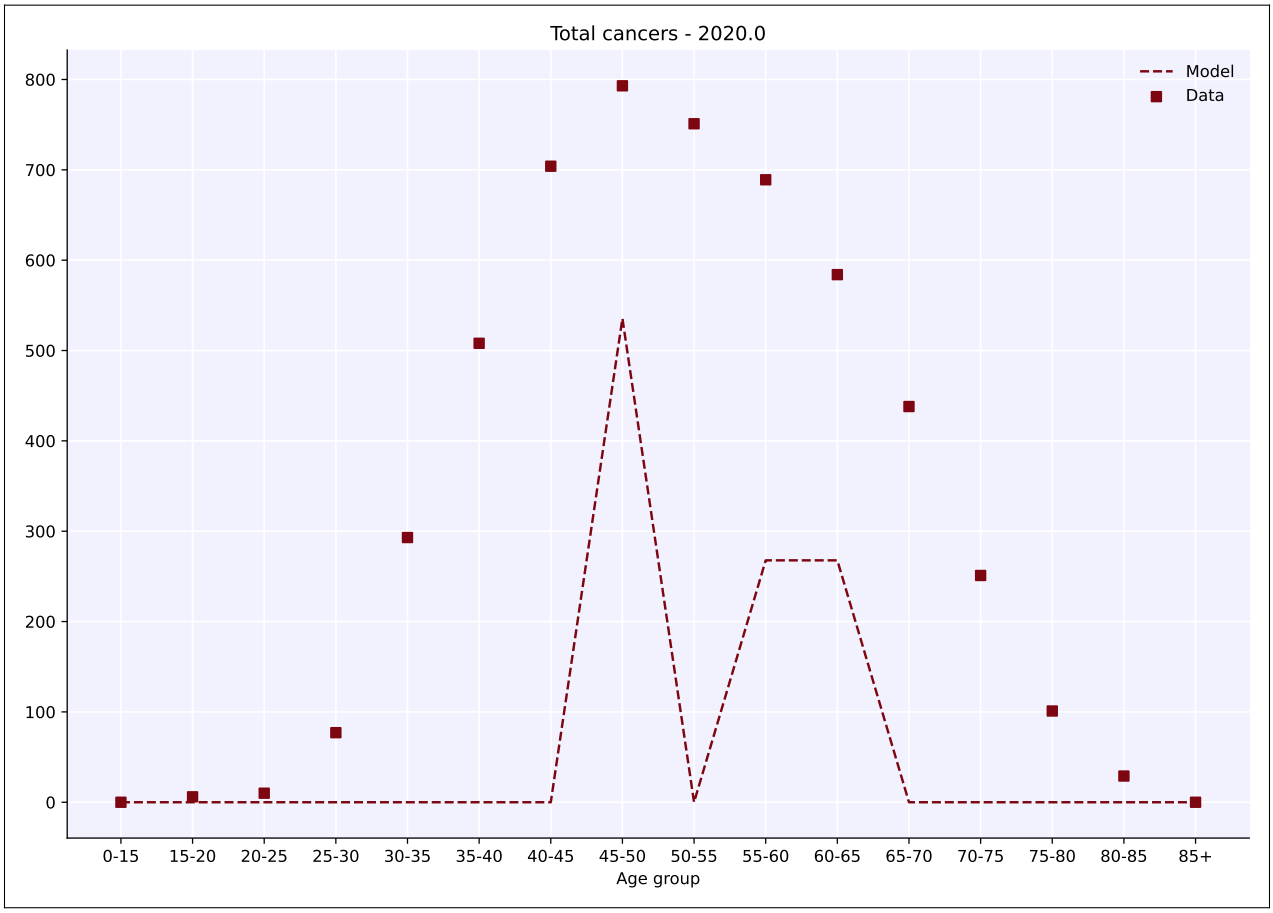
It's also possible to plot these results alongside data.

[2]:
```
az2 = hpv.age_results(
    result_keys=sc.objdict(
        total_cancers=sc.objdict(
            datafile='example_cancer_cases.csv',
        ),
    )
)
sim = hpv.Sim(pars, genotypes=[16, 18], analyzers=[az2])
sim.run()
a = sim.get_analyzer()
a.plot();
```

```
Loading location-specific demographic data for "tanzania"
Initializing sim with 50000 agents
Loading location-specific data for "tanzania"
  Running 1970.0 ( 0/51) (0.09 s)  ---------------------- 2%
```

```
/home/docs/checkouts/readthedocs.org/user_builds/institute-for-disease-modeling-hpvsim/
↪envs/latest/lib/python3.9/site-packages/hpvsim/data/loaders.py:209: FutureWarning: The␣
↪default value of numeric_only in DataFrameGroupBy.sum is deprecated. In a future␣
↪version, numeric_only will default to False. Either specify numeric_only or select␣
↪only columns which should be valid for the function.
  dd = full_df.groupby("Time").sum()["PopTotal"]
```

```
  Running 1980.0 (10/51) (0.65 s)  ••••---------------- 22%
  Running 1990.0 (20/51) (1.37 s)  ••••••••------------ 41%
  Running 2000.0 (30/51) (2.27 s)  ••••••••••••-------- 61%
  Running 2010.0 (40/51) (3.49 s)  ••••••••••••••••---- 80%
  Running 2020.0 (50/51) (5.29 s)  •••••••••••••••••••• 100%

Simulation summary:
       66,936 total infections
       14,726 total cin1s
       17,671 total cin2s
        9,371 total cin3s
       41,768 total cins
        1,071 total cancers
            0 total cancer detections
        1,071 total cancer deaths
            0 total detected cancer deaths
       14,458 total reinfections
       17,939 total reactivations
            0 total hiv infections
         0.06 total hpv incidence (/100)
           48 total cin1 incidence (/100,000)
           58 total cin2 incidence (/100,000)
           31 total cin3 incidence (/100,000)
          136 total cin incidence (/100,000)
            3 total cancer incidence (/100,000)
         0.31 total hpv prevalence (/100)
```



These results are not particularly well matched to the data, but we will deal with this in the calibration tutorial later.

1.2.6.2 Snapshots

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. The following example leverages a snapshot in order to create a figure demonstrating age mixing patterns among sexual contacts:

[3]:
```
snap = hpv.snapshot(timepoints=['2020'])
sim = hpv.Sim(pars, analyzers=snap)
sim.run()

a = sim.get_analyzer()
people = a.snapshots[0]

# Plot age mixing
import pylab as pl
import matplotlib as mpl
fig, ax = pl.subplots(nrows=1, ncols=1, figsize=(5, 4))

fc = people.contacts['m']['age_f'] # Get the age of female contacts in marital␣
↪partnership
```

(continues on next page)

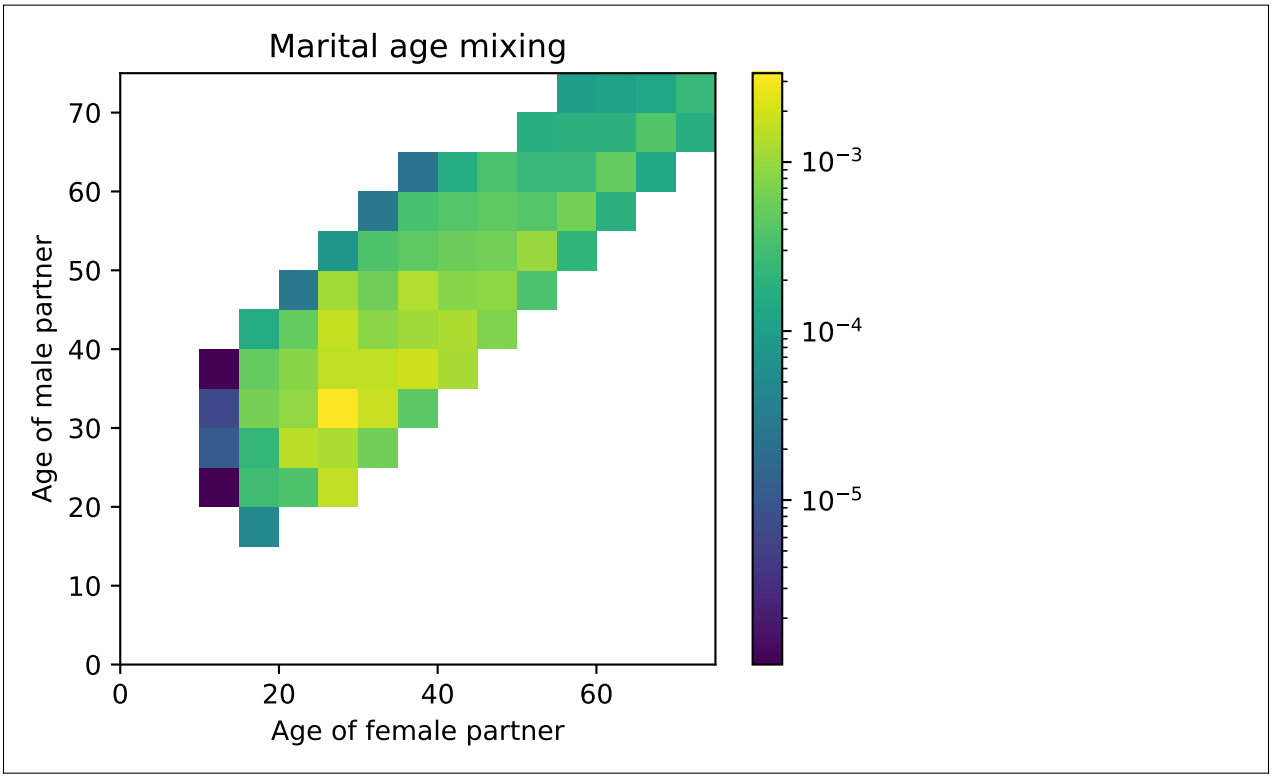
(continued from previous page)

```
mc = people.contacts['m']['age_m'] # Get the age of male contacts in marital partnership
h = ax.hist2d(fc, mc, bins=np.linspace(0, 75, 16), density=True, norm=mpl.colors.
↪LogNorm())
ax.set_xlabel('Age of female partner')
ax.set_ylabel('Age of male partner')
fig.colorbar(h[3], ax=ax)
ax.set_title('Marital age mixing')
pl.show();
```

```
Loading location-specific demographic data for "tanzania"
No genotypes provided, will assume only simulating HPV16 by default
Initializing sim with 50000 agents
Loading location-specific data for "tanzania"
  Running 1970.0 ( 0/51) (0.07 s)  ---------------------- 2%
```

```
/home/docs/checkouts/readthedocs.org/user_builds/institute-for-disease-modeling-hpvsim/
↪envs/latest/lib/python3.9/site-packages/hpvsim/data/loaders.py:209: FutureWarning: The␣
↪default value of numeric_only in DataFrameGroupBy.sum is deprecated. In a future␣
↪version, numeric_only will default to False. Either specify numeric_only or select␣
↪only columns which should be valid for the function.
  dd = full_df.groupby("Time").sum()["PopTotal"]
```

```
  Running 1980.0 (10/51) (0.88 s)  ••••---------------- 22%
  Running 1990.0 (20/51) (1.94 s)  ••••••••------------ 41%
  Running 2000.0 (30/51) (3.20 s)  ••••••••••••-------- 61%
  Running 2010.0 (40/51) (4.69 s)  ••••••••••••••••---- 80%
  Running 2020.0 (50/51) (6.45 s)  •••••••••••••••••••• 100%

Simulation summary:
       44,714 total infections
       13,387 total cin1s
       12,049 total cin2s
        5,890 total cin3s
       31,326 total cins
        1,339 total cancers
            0 total cancer detections
          803 total cancer deaths
            0 total detected cancer deaths
       11,245 total reinfections
       10,710 total reactivations
            0 total hiv infections
         0.07 total hpv incidence (/100)
           44 total cin1 incidence (/100,000)
           39 total cin2 incidence (/100,000)
           19 total cin3 incidence (/100,000)
          102 total cin incidence (/100,000)
            4 total cancer incidence (/100,000)
         0.22 total hpv prevalence (/100)
```



1.2.6.3 Age pyramids

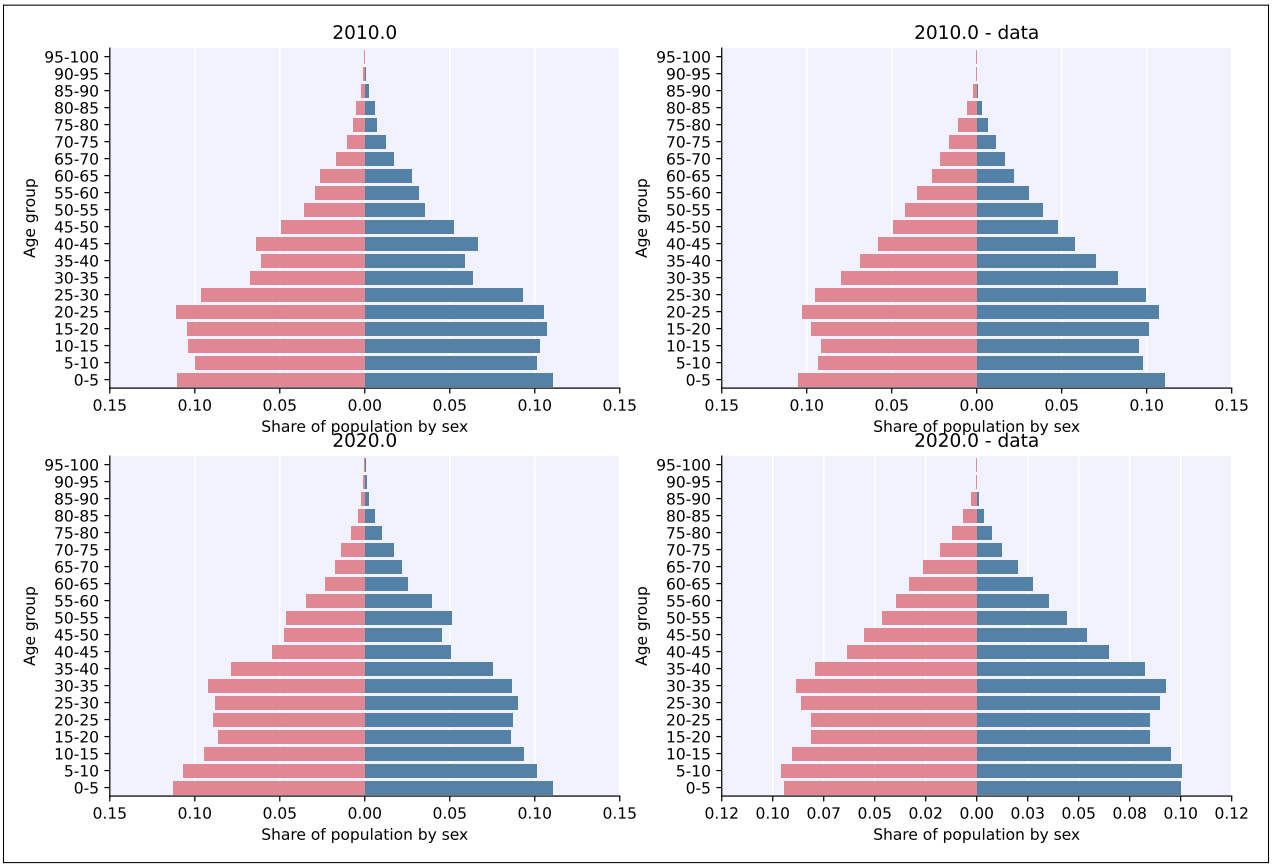
Age pyramids, like snapshots, take a picture of the people at a given point in time, and then bin them into age groups by sex. These can also be plotted alongside data:

[4]:
```
# Create some parameters
pars = dict(n_agents=50e3, start=2000, n_years=30, dt=0.5)

# Make the age pyramid analyzer
age_pyr = hpv.age_pyramid(
    timepoints=['2010', '2020'],
    datafile='south_africa_age_pyramid.csv',
    edges=np.linspace(0, 100, 21))

# Make the sim, run, get the analyzer, and plot
sim = hpv.Sim(pars, location='south africa', analyzers=age_pyr)
sim.run()
a = sim.get_analyzer()
fig = a.plot(percentages=True);
```

```
Loading location-specific demographic data for "south africa"
No genotypes provided, will assume only simulating HPV16 by default
Initializing sim with 50000 agents
Loading location-specific data for "south africa"
Dates provided in the age pyramid datafile ({'2020.0', '2010.0', '2000.0', '1990.0'})␣
↪are not the same as the age pyramid dates that were requested (['2010.0' '2020.0']).
Plots will only show requested dates, not all dates in the datafile.
  Running 2000.0 ( 0/62) (0.08 s)  ---------------------- 2%
```

```
/home/docs/checkouts/readthedocs.org/user_builds/institute-for-disease-modeling-hpvsim/
↪envs/latest/lib/python3.9/site-packages/hpvsim/data/loaders.py:209: FutureWarning: The␣
↪default value of numeric_only in DataFrameGroupBy.sum is deprecated. In a future␣
↪version, numeric_only will default to False. Either specify numeric_only or select␣
↪only columns which should be valid for the function.
  dd = full_df.groupby("Time").sum()["PopTotal"]
```

```
  Running 2005.0 (10/62) (0.70 s)  •••---------------- 18%
  Running 2010.0 (20/62) (1.39 s)  ••••••--------------- 34%
  Running 2015.0 (30/62) (2.13 s)  ••••••••••---------- 50%
  Running 2020.0 (40/62) (2.91 s)  ••••••••••••------- 66%
  Running 2025.0 (50/62) (3.73 s)  •••••••••••••••---- 82%
  Running 2030.0 (60/62) (4.59 s)  ••••••••••••••••••- 98%
Simulation summary:
    1,283,552 total infections
      323,451 total cin1s
      295,487 total cin2s
      198,545 total cin3s
      817,484 total cins
       21,439 total cancers
            0 total cancer detections
       10,254 total cancer deaths
            0 total detected cancer deaths
      366,330 total reinfections
      195,749 total reactivations
            0 total hiv infections
         2.40 total hpv incidence (/100)
          992 total cin1 incidence (/100,000)
          906 total cin2 incidence (/100,000)
          609 total cin3 incidence (/100,000)
        2,507 total cin incidence (/100,000)
           66 total cancer incidence (/100,000)
         5.19 total hpv prevalence (/100)
```



1.3 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

1.3.1 Version 0.4.0 (2022-11-16)

- Adds merge method for scenarios and fixes printing bugs
- *GitHub info*: PRs 422

1.3.2 Version 0.3.9 (2022-11-15)

- Simplifies genotype initialization, adds checks for HIV runs.
- Since the last release, changes were also made to virological clearance rates for people receiving treatment - previously all treated people would clear infection, but now some may control latently instead.
- *GitHub info*: PRs 421 and 420

1.3.3 Version 0.3.8 (2022-11-02)

- Store treatment properties as part of sim.people
- *GitHub info*: PR 413

1.3.4 Version 0.3.7 (2022-11-01)

- Fix to ensure consistent results for the number of txvx doses
- *GitHub info*: PR 411

1.3.5 Version 0.3.6 (2022-11-01)

- Fix bug related to screening eligibility. NB, this has a sizeable impact on results - screening strategies will be much more effective after this fix.
- *GitHub info*: PR 396

1.3.6 Version 0.3.5 (2022-10-31)

- Store stocks related to interventions
- *GitHub info*: PR 395

1.3.7 Version 0.3.4 (2022-10-31)

- Bugfixes for therapeutic vaccination
- *GitHub info*: PR 394

1.3.8 Version 0.3.3 (2022-10-30)

- Changes to therapeautic vaccine efficacy assumptions
- *GitHub info*: PR 393

1.3.9 Version 0.3.2 (2022-10-26)

- Additional tutorials and minor release tidying
- *GitHub info*: PR 380

1.3.10 Version 0.3.1 (2022-10-26)

- Fixes bug with screening
- Increases coverage of baseline test
- *GitHub info*: PR 373

1.3.11 Version 0.3.0 (2022-10-26)

- Implements multiscale modeling
- Minor release tidying
- *GitHub info*: PR 365

1.3.12 Version 0.2.11 (2022-10-25)

- Changes the way dates of HPV clearance are assigned to use durations sampled
- *GitHub info*: PR 374

1.3.13 Version 0.2.10 (2022-10-24)

- Fixes bug with treatment
- *GitHub info*: PR 354

1.3.14 Version 0.2.9 (2022-10-18)

- Prevents infectious people from being passed to People.infect()
- Fixes bugs with initialization within scenario runs
- Remove ununsed prevalence results
- *GitHub info*: PR 338

1.3.15 Version 0.2.8 (2022-10-17)

- Fixes bug with intervention year interpolation
- Changes reactivation probabilities to annual, not per time step
- Refactor prognoses calls
- *GitHub info*: PR 338

1.3.16 Version 0.2.7 (2022-10-14)

- Adds robust relative paths via `hpv.datadir`
- *GitHub info*: PR 333

1.3.17 Version 0.2.6 (2022-10-12)

- Removes Numba since slower for small sims and only 10% faster for large sims.
- Moves functions from `utils.py` into `people.py`, `sim.py`, and `population.py`.
- *GitHub info*: PR 326

1.3.18 Version 0.2.5 (2022-10-07)

- Adds people filtering (NB: not used, and later removed).
- Fixes bug with `print(sim)` not working.
- Adds baseline tests.
- *GitHub info*: PR 310

1.3.19 Version 0.2.4 (2022-10-07)

- Changes to dysplasia progression parameterization
- Adds a new implementation of HPV natural history for HIV positive women
- Note: HIV was added since the previous version
- *GitHub info*: PR 304

1.3.20 Version 0.2.3 (2022-09-01)

- Adds a `use_migration` parameter that activates immigration/emigration to ensure population sizes line up with data.
- Adds simple data versioning.
- *GitHub info*: PR 279

1.3.21 Version 0.2.2 (2022-08-22)

- Separates out the `Calibration` class into a separate file and to no longer inherit from `Analyzer`. Functionality is unchanged.
- *GitHub info*: PR 255

1.3.22 Version 0.2.1 (2022-08-19)

- Improves calibration to enable support for MySQL.
- Fixes plotting bug.
- *GitHub info*: PR 253

1.3.23 Version 0.2.0 (2022-08-19)

- Fixed tests and data loading logic.
- *GitHub info*: PR 251

1.3.24 Version 0.1.0 (2022-08-01)

- Updated calibration.
- *GitHub info*: PR 215

1.3.25 Version 0.0.3 (2022-07-18)

- Updated data loading scripts.
- *GitHub info*: PR 156

1.3.26 Version 0.0.2 (2022-06-15)

- Made into a Python module.
- *GitHub info*: PR 64

1.3.27 Version 0.0.1 (2022-04-04)

- Initial version.

1.4 API reference

1.4.1 Subpackages

1.4.1.1 hpvsim.data package

Submodules

hpvsim.data.get_data module

Download data needed for HPVsim.

Typically, this is done automatically: on load, HPVsim checks if the data are already downloaded, and if not, downloads them using the quick_download() function. The "slow download" functions supply the files that are usually zipped and stored in a separate repository, hpvsim_data.

To ensure the data is updated, update the data_version parameter below.

`get_data`()
: Download data

`quick_download`(*verbose=True*, *init=False*)
: Download pre-processed data files

check_downloaded(*verbose=1*, *check_version=True*)

Check if data is downloaded. Note: to update data, update the date here and in data/files/metadata.json.

Parameters

- **verbose** (*int*) – detail to print (0 = none, 1 = reason for failure, 2 = everything)
- **check_version** (*bool*) – whether to treat a version mismatch as a failure

remove_data(*verbose=True*, ***kwargs*)

Remove downloaded data; arguments passed to sc.rmpath()

hpvsim.data.loaders module

Load data

get_country_aliases(*wb=False*)

Define aliases for countries with odd names in the data

map_entries(*json*, *location*, *df=None*, *wb=False*)

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

get_age_distribution(*location=None*, *year=None*, *total_pop_file=None*)

Load age distribution for a given country or countries.

Parameters

- **location** (*str*) – name of the country to load the age distribution for
- **year** (*int*) – year to load the age distribution for
- **total_pop_file** (*str*) – optional filepath to save total population size for every year

Returns

Numpy array of age distributions, or dict if multiple locations

Return type

age_data (array)

get_total_pop(*location=None*)

Load total population for a given country or countries.

Parameters

location (*str or list*) – name of the country to load the total population for

Returns

Dataframe of year and pop_size columns

Return type

pop_data (dataframe)

get_death_rates(*location=None*, *by_sex=True*, *overall=False*)

Load death rates for a given country or countries.

Parameters

- **location** (*str or list*) – name of the country or countries to load the age distribution for
- **by_sex** (*bool*) – whether to rates by sex
- **overall** (*bool*) – whether to load total rate

Returns
death rates by age and sex

Return type
death_rates (dict)

get_birth_rates(*location=None*)

Load crude birth rates for a given country

Parameters
location (*str or list*) – name of the country to load the birth rates for

Returns
years and crude birth rates

Return type
birth_rates (arr)

get_life_expectancy(*location=None*, *by_sex=True*, *overall=False*)

Load life expectancy by age for a given country or countries.

Parameters

- **location** (*str or list*) – name of the country or countries to load the age distribution for
- **by_sex** (*bool*) – whether to rates by sex
- **overall** (*bool*) – whether to load total rate

Returns
life expectancy by age and sex

Return type
life_expectancy (dict)

1.4.2 Submodules

1.4.2.1 hpvsim.analysis module

Additional analysis functions that are not part of the core workflow, but which are useful for particular investigations.

class Analyzer(*label=None*)

Bases: `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)

static reduce(*analyzers*, *use_mean=False*)

Create a reduced analyzer from a list of analyzers, using

Parameters

- **analyzers** – list of analyzers
- **use_mean** (*bool*) – whether to use medians (the default) or means to create the reduced analyzer

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(*timepoints*, **args*, *die=True*, ***kwargs*)

Bases: *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

- **timepoints** (*list*) – list of ints/strings/date objects, the days on which to take the snapshot
- **die** (*bool*) – whether or not to raise an exception if a date is not found (default true)
- **kwargs** (*dict*) – passed to *Analyzer*

Example:

```
sim = hpv.Sim(analyzers=hpv.snapshot('2015.4', '2020'))
sim.run()
snapshot = sim['analyzers'][0]
people = snapshot.snapshots[0]            # Option 1
```

(continues on next page)

(continued from previous page)

```
people = snapshot.snapshots['2020']       # Option 2
people = snapshot.get('2020')             # 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_pyramid(*timepoints*, *edges=None*, *age_labels=None*, *datafile=None*, *die=False*, ***kwargs*)

Bases: *Analyzer*

Constructs an age/sex pyramid at specified points within the sim. Can be used with data

Parameters

- **timepoints** (*list*) – list of ints/strings/date objects, the days on which to take the snapshot
- **die** (*bool*) – whether or not to raise an exception if a date is not found (default true)
- **kwargs** (*dict*) – passed to Analyzer()

Example:

```
sim = hpv.Sim(analyzers=hpv.age_pyramid('2015', '2020'))
sim.run()
age_pyramid = sim['analyzers'][0]
```

initialize(*sim*)

apply(*sim*)

finalize(*sim*)

static reduce(*analyzers*, *use_mean=False*, *bounds=None*, *quantiles=None*)

Create an averaged age pyramid from a list of age pyramid analyzers

plot(*m_color='#4682b4'*, *f_color='#ee7989'*, *fig_args=None*, *axis_args=None*, *data_args=None*, *percentages=True*, *do_save=None*, *fig_path=None*, *do_show=True*, ***kwargs*)

Plot the age pyramids

Parameters

- **m_color** (*hex or rgb*) – the color of the bars for males
- **f_color** (*hex or rgb*) – the color of the bars for females
- **fig_args** (*dict*) – passed to pl.figure()
- **axis_args** (*dict*) – passed to pl.subplots_adjust()
- **data_args** (*dict*) – 'width', 'color', and 'offset' arguments for the data
- **percentages** (*bool*) – whether to plot the pyramid as percentages or numbers
- **do_save** (*bool*) – whether to save

- **fig_path** (*str or filepath*) – filepath to save to
- **do_show** (*bool*) – whether to show the figure
- **kwargs** (*dict*) – passed to `hpv.options.with_style()`; see that function for choices

class age_results(*result_keys*, *die=False*, ***kwargs*)

Bases: *Analyzer*

Constructs results by age at specified points within the sim. Can be used with data

Parameters

- **result_keys** (*dict*) – dictionary with keys of results to generate and associated timepoints/age-bins to generate each result as well as whether to compute_fit
- **die** (*bool*) – whether or not to raise an exception if errors are found
- **kwargs** (*dict*) – passed to *Analyzer*

Example:

```
sim = hpv.Sim(analyzers=hpv.age_results(
result_keys=sc.objdict(
    hpv_prevalence=sc.objdict(
        timepoints=['1990'],
        edges=np.array([0.,20.,25.,30.,40.,45.,50.,55.,65.,100.]),
    ),
    hpv_incidence=sc.objdict(
        timepoints=['1990'],
        edges=np.array([0.,20.,30.,40.,50.,60.,70.,80.,100.])
    )
)
)
sim.run()
age_results = sim['analyzers'][0]
```

initialize(*sim*)

validate_results(*sim*)

convert_rname_stocks(*rname*)

Helper function for converting stock result names to people attributes

convert_rname_flows(*rname*)

Helper function for converting flow result names to people attributes

apply(*sim*)

Calculate age results

finalize(*sim*)

static reduce(*analyzers*, *use_mean=False*, *bounds=None*, *quantiles=None*)

Create an averaged age result from a list of age result analyzers

compute(*key*)

plot(*fig_args=None*, *axis_args=None*, *data_args=None*, *width=0.8*, *do_save=None*, *fig_path=None*, *do_show=True*, ***kwargs*)

Plot the age results

Parameters

- **fig_args** (*dict*) – passed to pl.figure()
- **axis_args** (*dict*) – passed to pl.subplots_adjust()
- **data_args** (*dict*) – 'width', 'color', and 'offset' arguments for the data
- **width** (*float*) – width of the bars
- **do_save** (*bool*) – whether to save
- **fig_path** (*str or filepath*) – filepath to save to
- **do_show** (*bool*) – whether to show the figure
- **kwargs** (*dict*) – passed to `hpv.options.with_style()`; see that function for choices

class age_causal_infection(*start_year=None*, ***kwargs*)

Bases: *Analyzer*

Determine the age at which people with cervical cancer were causally infected and time spent between infection and cancer.

initialize(*sim*)

apply(*sim*)

class cancer_detection(*symp_prob=0.01*, *treat_prob=0.01*, *product=None*, ***kwargs*)

Bases: *Analyzer*

Cancer detection via symptoms

Parameters

- **symp_prob** – Probability of having cancer detected via symptoms, rather than screening
- **treat_prob** – Probability of receiving treatment for those with symptom-detected cancer

initialize(*sim*)

apply(*sim*)

Check for new cancer detection, treat subset of detected cancers

1.4.2.2 hpvsim.base module

Base classes for HPVsim. 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.

class ParsObj(*pars*)

Bases: `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

class Result(*name=None*, *npts=None*, *scale=True*, *color=None*, *n_rows=0*, *n_copies=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)

Example:

```
import hpvsim as hpv
r1 = hpv.Result(name='test1', npts=10)
r1[:5] = 20
print(r1.values)
```

property npts

class BaseSim(**args*, ***kwargs*)

Bases: *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

property n

Count the number of people – if it fails, assume none

get_t(*dates*, *exact_match=False*, *return_date_format=None*)

Convert a string, date/datetime object, or int to a timepoint (int).

Parameters

- **date** (*str, date, int, or list*) – convert any of these objects to a timepoint relative to the simulation's start day
- **exact_match** (*bool*) – whether or not to demand an exact match to the requested date
- **return_date_format** (*None, str*) – if None, do not return dates; otherwise return them as strings or floats as requested

Returns

the time point in the simulation cloesst to the requested date

Return type
t (int or str)

Examples:

```
sim.get_t('2015-03-01') # Get the closest timepoint to the specified date
sim.get_t(3) # Will return 3
sim.get_t('2015') # Can use strings
sim.get_t(['2015.5', '2016.5']) # List of strings, will match as close as␣
↪possible
sim.get_t(['2015.5', '2016.5'], exact_match=True) # Raises an error since these␣
↪dates aren't directly simulated
```

result_keys(*which='all'*)

Get the actual results objects, not other things stored in sim.results.

If which is 'main', return only the main results keys. If 'genotype', return only genotype 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:

```
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 "shrunken" 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 shrunken copy instead

Returns
a Sim object with the listed attributes removed

Return type
shrunken (*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:

```
sim.save() # Saves to a .sim file
```

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 hpv.load()

Returns
the loaded simulation object

Return type
sim (*Sim*)

Example:

```
sim = hpv.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:

```
tp = hpv.test_prob(symp_prob=0.1)
cb1 = hpv.change_beta(days=5, changes=0.3, label='NPI')
cb2 = hpv.change_beta(days=10, changes=0.3, label='Masks')
sim = hpv.Sim(interventions=[tp, cb1, cb2])
cb1, cb2 = sim.get_interventions(hpv.change_beta)
tp, cb2 = sim.get_interventions([0,2])
ind = sim.get_interventions(hpv.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:

```
tp = hpv.test_prob(symp_prob=0.1)
cb = hpv.change_beta(days=5, changes=0.3, label='NPI')
sim = hpv.Sim(interventions=[tp, cb])
cb = sim.get_intervention('NPI')
cb = sim.get_intervention('NP', partial=True)
cb = sim.get_intervention(hpv.change_beta)
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(*pars*)

Bases: `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.

Initialize essential attributes used for filtering

initialize()

Initialize underlying storage and map arrays

set_pars(*pars=None*, *hiv_pars=None*)

Re-link the parameters stored in the people object to the sim containing it, and perform some basic validation.

validate(*sim_pars=None*, *verbose=False*)

Perform validation on the People object.

Parameters

- **sim_pars** (*dict*) – dictionary of parameters from the sim to ensure they match the current People object
- **verbose** (*bool*) – detail to print

lock()

Lock the people object to prevent keys from being added

unlock()

Unlock the people object to allow keys to be added

filter_inds(*inds*)

Store indices to allow for easy filtering of the People object.

Parameters
inds (*array*) – filter by these indices

Returns
A filtered People object, which works just like a normal People object except only operates on a subset of indices.

filter(*criteria*)

Store indices to allow for easy filtering of the People object.

Parameters
criteria (*array*) – a boolean array for the filtering critria

Returns
A filtered People object, which works just like a normal People object except only operates on a subset of indices.

unfilter()

Set main simulation attributes to be views of the underlying data

This method should be called whenever the number of agents required changes (regardless of whether or not the underlying arrays have been resized)

addtoself(*people2*)

Combine two people arrays, avoiding dcp

set(*key*, *value*, *die=True*)

get(*key*)

Convenience method – key can be string or list of strings

property is_female

Boolean array of everyone female

property is_female_alive

Boolean array of everyone female and alive

property is_male

Boolean array of everyone male

property is_male_alive

Boolean array of everyone male and alive

property f_inds

Indices of everyone female

property m_inds

Indices of everyone male

property int_age

Return ages as an integer

property round_age

Rounds age up to the next highest integer

property dt_age

Return ages rounded to the nearest whole timestep

property is_active

Boolean array of everyone sexually active i.e. past debut

property is_female_adult

Boolean array of everyone eligible for screening

property is_virgin

Boolean array of everyone not yet sexually active i.e. pre debut

property alive_inds

Indices of everyone alive

property alive_level0

Indices of everyone alive who is a level 0 agent

property alive_level0_inds

Indices of everyone alive who is a level 0 agent

property n_alive

Number of people alive

property n_alive_level0

Number of people alive

property infected

Boolean array of everyone infected. Union of infectious and inactive. Includes people with cancer, people with latent infections, and people with active infections

property cin

Boolean array of everyone with dysplasia. Union of CIN1, CIN2, CIN3

property precin

Boolean array of everyone infectious with no dysplasia. Includes people with transient infections that will clear on their own plus those where dysplasia isn't established yet

property latent

Boolean array of everyone with latent infection. By definition, these people have no dysplasia and inactive infection status.

true(*key*)

Return indices matching the condition

true_by_genotype(*key*, *genotype*)

Return indices matching genotype-condition

false_by_genotype(*key*, *genotype*)

Return indices not matching genotype-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*, *weighted=True*)

Count the number of people for a given key

count_any(*key*, *weighted=True*)

Count the number of people for a given key for a 2D array if any value matches

count_by_genotype(*key*, *genotype*, *weighted=True*)

Count the number of people for a given key

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)

imm_keys()

Returns keys for different states of a person (e.g., symptomatic)

intv_keys()

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 acts

indices()

The indices of each people array

to_df()

Convert to a Pandas dataframe

to_arr()

Return as numpy array

person(*ind*)

Method to create person from the people

to_list()

Return all people as a list

from_list(*people*)

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:

```
import hpvsim as hpv
import networkx as nx
sim = hpv.Sim(n_agents=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]
nx.draw(G, node_color=node_colors, edge_color=edge_colors, width=edge_weights,␣
↪alpha=0.5)
```

save(*filename=None*, *force=False*, ***kwargs*)

Save to disk as a gzipped pickle.

Note: by default this function raises an exception if trying to save a run or partially run People object, since the changes that happen during a run are usually irreversible.

Parameters

- **filename** (*str or None*) – the name or path of the file to save to; if None, uses stored
- **force** (*bool*) – whether to allow saving even of a run or partially-run People object
- **kwargs** – passed to `sc.makefilepath()`

Returns

the validated absolute path to the saved file

Return type

filename (str)

Example:

```
sim = hpv.Sim()
sim.initialize()
sim.people.save() # Saves to a .ppl file
```

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
- **args** (*list*) – passed to hpv.load()
- **kwargs** (*dict*) – passed to hpv.load()

Returns
the loaded people object

Return type
people (*People*)

Example:

```
people = hpv.people.load('my-people.ppl')
```

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*, *debut=-1*, *partners=None*, *current_partners=None*, *rship_start_dates=None*, *rship_end_dates=None*, *n_rships=None*)

Bases: 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(*data=None*, *layer_keys=None*, ***kwargs*)

Bases: *FlexDict*

A simple (for now) class for storing different contact layers.

Parameters

- **data** (*dict*) – a dictionary that looks like a Contacts object
- **layer_keys** (*list*) – if provided, create an empty Contacts object with these layers
- **kwargs** (*dict*) – additional layer(s), merged with data

add_layer(***kwargs*)

Small method to add one or more layers to the contacts. Layers should be provided as keyword arguments.

Example:

```
hospitals_layer = hpv.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 = hpv.Sim(n_agents=50, pop_type='hybrid').run()
G = sim.people.contacts.to_graph()
nx.draw(G)
```

class Layer(**args*, *label=None*, ***kwargs*)

Bases: *FlexDict*

A small class holding a single layer of contact edges (connections) between people.

The input is typically arrays including: person 1 of the connection, person 2 of the connection, the weight of the connection, the duration and start/end times 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

- **f** (*array*) – an array of N connections, representing people on one side of the connection
- **m** (*array*) – an array of people on the other side of the connection
- **acts** (*array*) – an array of number of acts per timestep for each connection
- **dur** (*array*) – duration of the connection
- **start** (*array*) – start time of the connection
- **end** (*array*) – end time of the 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:

```
# 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 = hpv.Layer(p1=p1, p2=p2, beta=beta, label='rand')
layer = hpv.Layer(dict(p1=p1, p2=p2, beta=beta), label='rand') # Alternate method

# Convert one layer to another with extra columns
index = np.arange(n)
self_conn = p1 == p2
layer2 = hpv.Layer(**layer, index=index, self_conn=self_conn, label=layer.label)
```

property members

Return sorted array of all members

meta_keys()

Return the keys for the layer's meta information – i.e., f, m, beta, any others

validate(*force=True*)

Check the integrity of the layer: right types, right lengths.

If dtype is incorrect, try to convert automatically; if length is incorrect, do not.

get_inds(*inds*, *remove=False*)

Get the specified indices from the edgelist and return them as a dict.

Parameters
inds (*int, array, slice*) – the indices to be removed

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 f,m,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:

```
import networkx as nx
sim = hpv.Sim(n_agents=20, pop_type='hybrid').run()
G = sim.people.contacts['h'].to_graph()
nx.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 HPVsim People object, which is usually used to make new contacts
- **frac** (*float*) – the fraction of contacts to update on each timestep

1.4.2.3 hpvsim.calibration module

Define the calibration class

class Calibration(*sim*, *datafiles*, *calib_pars=None*, *genotype_pars=None*, *fit_args=None*, *par_samplers=None*, *n_trials=None*, *n_workers=None*, *total_trials=None*, *name=None*, *db_name=None*, *keep_db=None*, *storage=None*, *rand_seed=None*, *label=None*, *die=False*, *verbose=True*)

Bases: `prettyobj`

A class to handle calibration of HPVsim 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
- **datafiles** (*list*) – list of datafile strings to calibrate to
- **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
- **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: 'hpvsim_calibration')
- **db_name** (*str*) – the name of the database file (default: 'hpvsim_calibration.db')
- **keep_db** (*bool*) – whether to keep the database after calibration (default: false)
- **storage** (*str*) – the location of the database (default: sqlite)
- **rand_seed** (*int*) – if provided, use this random seed to initialize Optuna runs (for reproducibility)
- **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 hpv.Calibration()

Returns

A Calibration object

Example:

```
sim = hpv.Sim(pars, genotypes=[16, 18])
calib_pars = dict(beta=[0.05, 0.010, 0.20],hpv_control_prob=[.9, 0.5, 1])
calib = hpv.Calibration(sim, calib_pars=calib_pars,
                        datafiles=['test_data/south_africa_hpv_data.xlsx',
                                   'test_data/south_africa_cancer_data.xlsx'],
                        total_trials=10, n_workers=4)
calib.calibrate()
calib.plot()
```

run_sim(*calib_pars=None*, *genotype_pars=None*, *label=None*, *return_sim=False*)

Create and run a simulation

static get_full_pars(*sim=None*, *calib_pars=None*, *genotype_pars=None*)

Make a full pardict from the subset of regular sim parameters and genotype parameters used in calibration

trial_pars_to_sim_pars(*trial_pars=None*, *which_pars=None*, *return_full=True*)

Create genotype_pars and pars dicts from the trial parameters. Note: not used during self.calibrate. :param trial_pars: dictionary of parameters from a single trial. If not provided, best parameters will be used :type trial_pars: dict :param return_full: whether to return a unified par dict ready for use in a sim, or the sim pars and genotype pars separately :type return_full: bool

Example:

```
sim = hpv.Sim(genotypes=[16, 18])
calib_pars = dict(beta=[0.05, 0.010, 0.20],hpv_control_prob=[.9, 0.5, 1])
genotype_pars = dict(hpv16=dict(prog_time=[3, 3, 10]))
calib = hpv.Calibration(sim, calib_pars=calib_pars, genotype_pars=genotype_pars
                    datafiles=['test_data/south_africa_hpv_data.xlsx',
                               'test_data/south_africa_cancer_data.xlsx'],
                    total_trials=10, n_workers=4)
calib.calibrate()
new_pars = calib.trial_pars_to_sim_pars() # Returns best parameters from␣
↪calibration in a format ready for sim running
sim.update_pars(new_pars)
sim.run()
```

sim_to_sample_pars()

Convert sim pars to sample pars

trial_to_sim_pars(*pardict=None*, *trial=None*, *gname=None*)

Take in an optuna trial and sample from pars, after extracting them from the structure they're provided in

run_trial(*trial*, *save=True*)

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.

make_study()

Make a study, deleting one if it already exists

calibrate(*calib_pars=None*, *genotype_pars=None*, *verbose=True*, *load=True*, *tidyup=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.)

parse_study(*study*)

Parse the study into a data frame – called automatically

to_json(*filename=None*, *indent=2*, ***kwargs*)

Convert the data to JSON.

plot(*res_to_plot=None*, *fig_args=None*, *axis_args=None*, *data_args=None*, *do_save=None*, *fig_path=None*, *do_show=True*, *plot_type='sns.boxplot'*, ***kwargs*)

Plot the calibration results

Parameters

- **res_to_plot** (*int*) – number of results to plot. if None, plot them all
- **fig_args** (*dict*) – passed to pl.figure()
- **axis_args** (*dict*) – passed to pl.subplots_adjust()
- **data_args** (*dict*) – 'width', 'color', and 'offset' arguments for the data
- **do_save** (*bool*) – whether to save
- **fig_path** (*str or filepath*) – filepath to save to
- **do_show** (*bool*) – whether to show the figure
- **kwargs** (*dict*) – passed to `hpv.options.with_style()`; see that function for choices

1.4.2.4 hpvsim.defaults module

Set the defaults across each of the different files.

default_float

alias of `float32`

default_int

alias of `int32`

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'

1.4.2.5 hpvsim.immunity module

Defines classes and methods for calculating immunity

init_immunity(*sim*, *create=True*)

Initialize immunity matrices with all genotypes and vaccines in the sim

update_peak_immunity(*people*, *inds*, *imm_pars*, *imm_source*, *offset=None*, *infection=True*)

Update immunity level

This function updates the immunity for individuals when an infection or vaccination occurs.

- individuals that are infected and already have immunity from a previous vaccination/infection have their immunity level;
- **individuals without prior immunity are assigned an initial level drawn from a distribution. This level** depends on whether the immunity is from a natural infection or from a vaccination (and if so, on the type of vaccine).

Parameters

- **people** – A people object

- **inds** – Array of people indices
- **imm_pars** – Parameters from which to draw values for quantities like ['imm_init'] - either sim pars (for natural immunity) or vaccine pars
- **imm_source** – index of either genotype or vaccine where immunity is coming from

Returns: None

check_immunity(*people*)

Calculate people's immunity on this timestep from prior infections. As an example, suppose HPV16 and 18 are in the sim, and the cross-immunity matrix is:

pars['immunity'] = np.array([[1., 0.5],
[0.3, 1.]])

This indicates that people who've had HPV18 have 50% protection against getting 16, and people who've had 16 have 30% protection against getting 18. Now suppose we have 3 people, whose immunity levels are

people.imm = np.array([[0.9, 0.0, 0.0],
[0.0, 0.7, 0.0]])

This indicates that person 1 has a prior HPV16 infection, person 2 has a prior HPV18 infection, and person 3 has no history of infection.

In this function, we take the dot product of pars['immunity'] and people.imm to get:

people.sus_imm = np.array([[0.9 , 0.35, 0.],
[0.27, 0.7 , 0.]])

This indicates that the person 1 has high protection against reinfection with HPV16, and some (30% of 90%) protection against infection with HPV18, and so on.

precompute_waning(*t*, *pars=None*)

Process functional form and parameters into values:

- 'exp_decay' : exponential decay. Parameters should be init_val and half_life (half_life can be None/nan)
- 'linear_decay': linear decay

A custom function can also be supplied.

Parameters

- **length** (*float*) – length of array to return, i.e., for how long waning is calculated
- **pars** (*dict*) – passed to individual immunity functions

Returns
array of length 'length' of values

exp_decay(*t*, *init_val*, *half_life*)

Returns an array of length t with values for the immunity at each time step after recovery

linear_decay(*length*, *init_val*, *slope*)

Calculate linear decay

linear_growth(*length*, *slope*)

Calculate linear growth

1.4.2.6 hpvsim.interventions module

Specify the core interventions. Other interventions can be defined by the user by inheriting from these classes.

class RoutineDelivery(*years=None*, *start_year=None*, *end_year=None*, *prob=None*, *annual_prob=True*)

Bases: Intervention

Base class for any intervention that uses routine delivery; handles interpolation of input years.

initialize(*sim*)

class CampaignDelivery(*years*, *interpolate=None*, *prob=None*, *annual_prob=True*)

Bases: Intervention

Base class for any intervention that uses campaign delivery; handles interpolation of input years.

initialize(*sim*)

class dynamic_pars(*pars=None*, ***kwargs*)

Bases: 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:

```
interv = hpv.dynamic_pars(condoms=dict(timepoints=10, vals={'c':0.9})) # Increase␣
↪condom use amount casual partners to 90%
interv = hpv.dynamic_pars({'beta':{'timepoints':[10, 15], 'vals':[0.005, 0.015]}, #␣
↪At timepoint 10, reduce beta, then increase it again
                           'debut':{'timepoints':10, 'vals':dict(f=dict(dist='normal
↪', par1=20, par2=2.1), m=dict(dist='normal', par1=19.6, par2=1.8))}}) # Increase␣
↪mean age of sexual debut
```

initialize(*sim*)

Initialize with a sim

apply(*sim*)

Loop over the parameters, and then loop over the timepoints, applying them if any are found

class EventSchedule

Bases: Intervention

Run functions on different days

This intervention is a a kind of generalization of `dynamic_pars` to allow more flexibility in triggering multiple, arbitrary operations and to more easily assemble multiple changes at different times. This intervention can be used to implement scale-up or other changes to interventions without needing to implement time-dependency in the intervention itself.

To use the intervention, simply index the intervention by `t` or by date.

Example:

```
>>> iv = EventSchedule()
>>> iv[1] = lambda sim: print(sim.t)
>>> iv['2020-04-02'] = lambda sim: print('foo')
```

initialize(*sim*)

apply(*sim*)

set_intervention_attributes(*sim*, *intervention_name*, ***kwargs*)

class BaseVaccination(*product=None*, *prob=None*, *age_range=None*, *sex=None*, *eligibility=None*, *label=None*, ***kwargs*)

Bases: `Intervention`

Base vaccination class for determining who will receive a vaccine.

Parameters

- **product** (*str/Product*) – the vaccine to use
- **prob** (*float/arr*) – annual probability of eligible population getting vaccinated
- **age_range** (*list/tuple*) – age range to vaccinate
- **sex** (*int/str/list*) – sex to vaccinate - accepts 0/1 or 'f'/'m' or a list of both
- **eligibility** (*inds/callable*) – indices OR callable that returns inds
- **label** (*str*) – the name of vaccination strategy
- **kwargs** (*dict*) – passed to Intervention()

initialize(*sim*)

check_eligibility(*sim*)

Determine who is eligible for vaccination

apply(*sim*)

Perform vaccination by finding who's eligible for vaccination, finding who accepts, and applying the vaccine product.

shrink(*in_place=True*)

Shrink vaccination intervention

class routine_vx(*product=None*, *prob=None*, *age_range=None*, *sex=0*, *eligibility=None*, *start_year=None*, *end_year=None*, *years=None*, ***kwargs*)

Bases: *BaseVaccination*, *RoutineDelivery*

Routine vaccination - an instance of base vaccination combined with routine delivery. See base classes for a description of input arguments.

Examples:

```
vx1 = hpv.routine_vx(product='bivalent', age_range=[9,10], prob=0.9, start_
↪year=2025) # Vaccinate 90% of girls aged 9-10 every year
vx2 = hpv.routine_vx(product='bivalent', age_range=[9,10], prob=0.9, sex=[0,1],␣
↪years=np.arange(2020,2025)) # Screen 90% of girls and boys aged 9-10 every year␣
↪from 2020-2025
vx3 = hpv.routine_vx(product='quadrivalent', prob=np.linspace(0.2,0.8,5), years=np.
↪arange(2020,2025)) # Scale up vaccination over 5 years starting in 2020
```

initialize(*sim*)

class campaign_vx(*product=None*, *prob=None*, *age_range=None*, *sex=0*, *eligibility=None*, *years=None*, *interpolate=True*, ***kwargs*)

Bases: *BaseVaccination*, *CampaignDelivery*

Campaign vaccination - an instance of base vaccination combined with campaign delivery. See base classes for a description of input arguments.

initialize(*sim*)

class BaseTest(*product=None*, *prob=None*, *eligibility=None*, ***kwargs*)

Bases: Intervention

Base class for screening and triage.

Parameters

- **product** (*str/Product*) – the diagnostic to use
- **prob** (*float/arr*) – annual probability of eligible women receiving the diagnostic
- **eligibility** (*inds/callable*) – indices OR callable that returns inds
- **label** (*str*) – the name of screening strategy
- **kwargs** (*dict*) – passed to Intervention()

initialize(*sim*)

deliver(*sim*)

Deliver the diagnostics by finding who's eligible, finding who accepts, and applying the product.

check_eligibility(*sim*)

class BaseScreening(*age_range=None*, ***kwargs*)

Bases: *BaseTest*

Base class for screening.

Parameters

- **age_range** (*list/tuple/arr*) – age range for screening, e.g. [30,50]
- **kwargs** (*dict*) – passed to BaseTest

check_eligibility(*sim*)

Return an array of indices of agents eligible for screening at time t, i.e. sexually active females in age range, plus any additional user-defined eligibility, which often includes the screening interval.

apply(*sim*)

Perform screening by finding who's eligible, finding who accepts, and applying the product.

class routine_screening(*product=None*, *prob=None*, *eligibility=None*, *age_range=None*, *years=None*, *start_year=None*, *end_year=None*, ***kwargs*)

Bases: *BaseScreening*, *RoutineDelivery*

Routine screening - an instance of base screening combined with routine delivery. See base classes for a description of input arguments.

Examples:

```
screen1 = hpv.routine_screening(product='hpv', prob=0.02) # Screen 2% of the␣
↪eligible population every year
screen2 = hpv.routine_screening(product='hpv', prob=0.02, start_year=2020) # Screen␣
↪2% every year starting in 2020
screen3 = hpv.routine_screening(product='hpv', prob=np.linspace(0.005,0.025,5),␣
↪years=np.arange(2020,2025)) # Scale up screening over 5 years starting in 2020
```

initialize(*sim*)

class campaign_screening(*product=None*, *age_range=None*, *sex=None*, *eligibility=None*, *prob=None*, *years=None*, *interpolate=None*, ***kwargs*)

Bases: *BaseScreening*, *CampaignDelivery*

Campaign screening - an instance of base screening combined with campaign delivery. See base classes for a description of input arguments.

Examples:

```
screen1 = hpv.campaign_screening(product='hpv', prob=0.2, years=2030) # Screen 20%␣
↪of the eligible population in 2020
screen2 = hpv.campaign_screening(product='hpv', prob=0.02, years=[2025,2030]) #␣
↪Screen 20% of the eligible population in 2025 and again in 2030
```

initialize(*sim*)

class BaseTriage(***kwargs*)

Bases: *BaseTest*

Base class for triage.

Parameters
kwargs (*dict*) – passed to BaseTest

check_eligibility(*sim*)

apply(*sim*)

class routine_triage(*product=None*, *prob=None*, *eligibility=None*, *age_range=None*, *years=None*, *start_year=None*, *end_year=None*, *annual_prob=None*, ***kwargs*)

Bases: *BaseTriage*, *RoutineDelivery*

Routine triage - an instance of base triage combined with routine delivery. See base classes for a description of input arguments.

Examples:

```
# Example 1: Triage 40% of the eligible population in all years
triage1 = hpv.routine_triage(product='via_triage', prob=0.4)

# Example 2: Triage positive screens into confirmatory testing or theapeutic␣
↪vaccintion
screened_pos = lambda sim: sim.get_intervention('screening').outcomes['positive']
triage2 = hpv.routine_triage(product='pos_screen_assessment', eligibility=screen_
↪pos, prob=0.9, start_year=2030)
```

initialize(*sim*)

class campaign_triage(*product=None*, *age_range=None*, *sex=None*, *eligibility=None*, *prob=None*, *years=None*, *interpolate=None*, *annual_prob=None*, ***kwargs*)

Bases: *BaseTriage*, *CampaignDelivery*

Campaign triage - an instance of base triage combined with campaign delivery. See base classes for a description of input arguments.

Examples:

```
# Example 1: In 2030, triage all positive screens into confirmatory testing or␣
↪therapeutic vaccintion
screened_pos = lambda sim: sim.get_intervention('screening').outcomes['positive']
triage1 = hpv.campaign_triage(product='pos_screen_assessment', eligibility=screen_
↪pos, prob=0.9, years=2030)
```

initialize(*sim*)

class BaseTreatment(*product=None*, *prob=None*, *eligibility=None*, *age_range=None*, ***kwargs*)

Bases: `Intervention`

Base treatment class.

Parameters

- **product** (*str/Product*) – the treatment product to use
- **accept_prob** (*float/arr*) – acceptance rate of treatment - interpreted as the % of women eligble for treatment who accept
- **eligibility** (*inds/callable*) – indices OR callable that returns inds
- **label** (*str*) – the name of treatment strategy
- **kwargs** (*dict*) – passed to Intervention()

initialize(*sim*)

check_eligibility(*sim*)

Check people's eligibility for treatment

get_accept_inds(*sim*)

Get indices of people who will acccept treatment; these people are then added to a queue or scheduled for receiving treatment

get_candidates(*sim*)

Get candidates for treatment on this timestep. Implemented by derived classes.

apply(*sim*)

Perform treatment by getting candidates, checking their eligibility, and then treating them.

class treat_num(*max_capacity=None*, ***kwargs*)

Bases: *BaseTreatment*

Treat a fixed number of people each timestep.

Parameters

max_capacity (*int*) – maximum number who can be treated each timestep

add_to_queue(*sim*)

Add people who are willing to accept treatment to the queue

get_candidates(*sim*)

Get the indices of people who are candidates for treatment

apply(*sim*)

Apply treatment. On each timestep, this method will add eligible people who are willing to accept treatment to a queue, and then will treat as many people in the queue as there is capacity for.

class treat_delay(*delay=None*, ***kwargs*)

Bases: *BaseTreatment*

Treat people after a fixed delay

Parameters
delay (*int*) – years of delay between becoming eligible for treatment and receiving treatment.

add_to_schedule(*sim*)

Add people who are willing to accept treatment to the treatment scehduler

get_candidates(*sim*)

Get the indices of people who are candidates for treatment

apply(*sim*)

Apply treatment. On each timestep, this method will add eligible people who are willing to accept treatment to a scheduler, and then will treat anyone scheduled for treatment on this timestep.

class BaseTxVx(***kwargs*)

Bases: *BaseTreatment*

Base class for therapeutic vaccination

deliver(*sim*)

Deliver the intervention. This applies on a single timestep, whereas apply() methods apply on every timestep and can selectively call this method.

class routine_txvx(*product=None*, *prob=None*, *age_range=None*, *eligibility=None*, *start_year=None*, *end_year=None*, *years=None*, *annual_prob=None*, ***kwargs*)

Bases: *BaseTxVx*, *RoutineDelivery*

Routine delivery of therapeutic vaccine - an instance of treat_num combined
with routine delivery. See base classes for a description of input arguments.

Examples:

```
txvx1 = hpv.routine_txvx(product='txvx1', prob=0.9, age_range=[25,26], start_
↪year=2030) # Vaccinate 90% of 25yo women every year starting 2025
txvx2 = hpv.routine_txvx(product='txvx1', prob=np.linspace(0.2,0.8,5), age_
↪range=[25,26], years=np.arange(2030,2035)) # Scale up vaccination over 5 years␣
↪starting in 2020
```

initialize(*sim*)

apply(*sim*)

class campaign_txvx(*product=None*, *prob=None*, *age_range=None*, *eligibility=None*, *years=None*, *interpolate=True*, *annual_prob=None*, ***kwargs*)

Bases: *BaseTxVx*, *CampaignDelivery*

Campaign delivery of therapeutic vaccine - an instance of treat_num combined with campaign delivery. See base classes for a description of input arguments.

initialize(*sim*)

apply(*sim*)

class linked_txvx(***kwargs*)

Bases: *BaseTxVx*

Deliver therapeutic vaccine. This intervention should be used if TxVx delivery is linked to another program that determines eligibility, e.g. a screening program. Handling of dates is assumed to be handled by the linked intervention.

apply(*sim*)

class dx(*df*, *hierarchy*)

Bases: Product

Testing products are used within screening and triage. Their fundamental proprty is that they classify people into exactly one result state. They do not change anything about the People.

property default_value

administer(*sim*, *inds*, *return_format='dict'*)

Administer a testing product.

Returns

an array of length len(inds) with integer entries that map each person to one of the result_states if return_format==’dict’: a dictionary keyed by result_states with values containing the indices of people classified into this state

Return type

if return_format==’array’

class tx(*df*, *name=None*, *clears_all=False*)

Bases: Product

Treatment products include anything used to treat cancer or precancer, as well as therapeutic vaccination. They change fundamental properties about People, including their prognoses and infectiousness.

get_people_in_state(*state*, *g*, *sim*)

Find people within a given state/genotype. Returns indices

administer(*sim*, *inds*, *return_format='dict'*)

Loop over treatment states to determine those who are successfully treated and clear infection

class vx(*genotype_pars=None*, *imm_init=None*, *imm_boost=None*)

Bases: Product

Vaccine product

administer(*people*, *inds*)

Apply the vaccine to the requested people indices.

class radiation(*dur=None*)

Bases: Product

administer(*sim*, *inds*)

default_dx(*prod_name=None*)

Create default diagnostic products

default_tx(*prod_name=None*)

Create default treatment products

default_vx(*prod_name=None*)

Create default vaccine products

1.4.2.7 hpvsim.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 `sc.readdate()` are similar, and indeed this function calls `sc.readdate()` if the input is a string, in this function an integer is treated as a number of days from start_date, while for `sc.readdate()` it is treated as a timestamp in seconds.

Note: in this and other date functions, arguments work either with or without underscores (e.g. `start_date` or `startdate`)

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"

New in version 2.0.0: support for `np.datetime64` objects

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:

```
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=None*, *end_date=None*, *interval=None*, *inclusive=True*, *as_date=False*, *readformat=None*, *outformat=None*, ***kwargs*)

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.

Note: instead of an end date, can also pass one or more of days, months, weeks, or years, which will be added on to the start date via `sc.datedelta()`.

Parameters

- **start_date** (*int/str/date*) – the starting date, in any format
- **end_date** (*int/str/date*) – the end date, in any format
- **interval** (*int/str/dict*) – if an int, the number of days; if 'week', 'month', or 'year', one of those; if a dict, passed to `dt.relativedelta()`
- **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 (note: you can also use "asdate" instead of "as_date")
- **readformat** (*str*) – passed to date()

- **outformat** (*str*) – passed to date()
- **days** (*int*) – optional

Examples:

```
dates1 = sc.daterange('2020-03-01', '2020-04-04')
dates2 = sc.daterange('2020-03-01', '2022-05-01', interval=dict(months=2),␣
↪asdate=True)
dates3 = sc.daterange('2020-03-01', weeks=5)
```

New in version 1.0.0.

New in version 1.3.0: "interval" argument

New in version 2.0.0: `sc.datedelta()` arguments

load_data(*datafile*, *check_date=False*, *header='infer'*, *calculate=True*, ***kwargs*)

Load data for comparing to the model output, either from file or from a dataframe. Data is expected to be in wide format, with each row representing a year and columns for each variable by genotype/age/sex.

Parameters

- **datafile** (*str/df*) – if a string, the name of the file to load (either Excel or CSV); if a dataframe, use directly
- **start_year** (*int*) – first year with data available
- **kwargs** (*dict*) – passed to pd.read_excel()

Returns

pandas dataframe of the loaded data

Return type

data (dataframe)

load(**args*, *update=True*, *verbose=True*, ***kwargs*)

Convenience method for sc.loadobj() and equivalent to hpv.Sim.load() or hpv.Scenarios.load().

Parameters

- **filename** (*str*) – file to load
- **do_migrate** (*bool*) – whether to migrate if loading an old object
- **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:

```
sim = hpv.load('calib.sim') # Equivalent to hpv.Sim.load('calib.sim')
scens = hpv.load(filename='school-closures.scens', folder='schools')
```

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

Convenience method for sc.saveobj() and equivalent to hpv.Sim.save() or hpv.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:

```
hpv.save('calib.sim', sim) # Equivalent to sim.save('calib.sim')
hpv.save(filename='school-closures.scens', folder='schools', obj=scens)
```

savefig(*filename=None*, *comments=None*, *fig=None*, ***kwargs*)

Wrapper for Matplotlib's `pl.savefig()` function which automatically stores HPVsim metadata in the figure.

By default, saves (git) information from both the HPVsim version and the calling function. Additional comments can be added to the saved file as well. These can be retrieved via `hpv.get_png_metadata()` (or `sciris.sc_plotting.loadmetadata()`). Metadata can also be stored for PDF, but cannot be automatically retrieved.

Parameters

- **filename** (*str/list*) – name of the file to save to (default, timestamp); can also be a list of names
- **comments** (*str/dict*) – additional metadata to save to the figure
- **fig** (*fig/list*) – figure to save (by default, current one); can also be a list of figures
- **kwargs** (*dict*) – passed to `fig.savefig()`

Example:

```
hpv.Sim().run().plot()
hpv.savefig()
```

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 hpv.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:

```
hpv.git_info() # Return information
hpv.git_info(__file__) # Writes to disk
hpv.git_info('hpvsim_version.gitinfo') # Writes to disk
hpv.git_info('hpvsim_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:

```
hpv.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 HPVsim 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:

```
hpv.check_save_version()
hpv.check_save_version('1.3.2', filename='script.gitinfo', comments='This is the␣
↪main analysis script')
hpv.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 HPVsim '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 HPVsim 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 hpv.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:

```
hpv.Sim().run(do_plot=True)
hpv.savefig('hpvsim.png')
hpv.get_png_metadata('hpvsim.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:

```
hpv.get_doubling_time(sim, interval=[3,30]) # returns the doubling time over the␣
↪given interval (single float)
hpv.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:

```
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␣
↪method
e5 = compute_gof(x1, x2, as_scalar='median') # Normalized median absolute error --␣
↪highly robust
```

help(*pattern=None*, *source=False*, *ignorecase=True*, *flags=None*, *context=False*, *output=False*)

Get help on HPVsim in general, or search for a word/expression.

Parameters

- **pattern** (*str*) – the word, phrase, or regex to search for
- **source** (*bool*) – whether to search source code instead of docstrings for matches
- **ignorecase** (*bool*) – whether to ignore case (equivalent to `flags=re.I`)
- **flags** (*list*) – additional flags to pass to `re.findall()`
- **context** (*bool*) – whether to show the line(s) of matches
- **output** (*bool*) – whether to return the dictionary of matches

Examples:

```
hpv.help()
hpv.help('vaccine')
hpv.help('contact', ignorecase=False, context=True)
hpv.help('lognormal', source=True, context=True)
```

New in version 3.1.2.

1.4.2.8 hpvsim.parameters module

Set the parameters for hpvsim.

make_pars(***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 = hpv.Sim() and then inspect sim.pars, rather than calling this function directly.

Parameters

- **version** (*str*) – if supplied, use parameters from this version
- **kwargs** (*dict*) – any additional kwargs are interpreted as parameter names

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

1.4.2.9 hpvsim.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*, *pop_trend=None*, ***kwargs*)

Bases: *BasePeople*

A class to perform all the operations on the people – usually not invoked directly.

This class is usually 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. However, ages, contacts, etc. will need to be created separately – see `hpv.make_people()` instead.

Note that this class handles the mechanics of updating the actual people, while `hpv.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 n_agents
- **strict** (*bool*) – whether or not to only create keys that are already in self.meta.person; otherwise, let any key be set
- **pop_trend** (*dataframe*) – a dataframe of years and population sizes, if available
- **kwargs** (*dict*) – the actual data, e.g. from a popdict, being specified

Examples:

```
ppl1 = hpv.People(2000)

sim = hpv.Sim()
ppl2 = hpv.People(sim.pars)
```

init_flows()

Initialize flows to be zero

scale_flows(*inds*)

Return the scaled versions of the flows – replacement for len(inds) followed by scale factor multiplication

increment_age()

Let people age by one timestep

initialize(*sim_pars=None*, *hiv_pars=None*)

Perform initializations

update_states_pre(*t*, *year=None*)

Perform all state updates at the current timestep

set_prognoses(*inds*, *g*, *dt*, *hiv_pars=None*)

Set prognoses for people following infection. Wrapper method that calls the 4 separate methods for setting and updating prognoses.

set_dysp_status(*inds*, *g*, *dt*)

Use durations and dysplasia rates to determine whether HPV clears or progresses to dysplasia

set_severity(*inds*, *g*, *gpars*, *hiv_prog_rate=None*)

Set dysplasia severity and duration for women who develop dysplasia

set_dysp_rates(*inds*, *g*, *gpars*, *hiv_dysp_rate=None*)

Set dysplasia rates

set_cin_grades(*inds*, *g*, *dt*, *dysp_arrs*)

Set CIN clinical grades and dates of progression

set_hiv_prognoses(*inds*, *year=None*)

Set HIV outcomes (for now only ART)

dissolve_partnerships(*t=None*)

Dissolve partnerships

create_partnerships(*tind*, *mixing*, *layer_probs*, *cross_layer*, *dur_pship*, *acts*, *age_act_pars*, *pref_weight=100*)

Create partnerships. All the hard work of creating the contacts is done by hppop.make_contacts, which in turn relies on hpu.create_edgelist for creating the edgelist. This method is just a light wrapper that passes in the arguments in the right format and the updates relationship info stored in the People class.

check_inds(*current*, *date*, *filter_inds=None*)

Return indices for which the current state is false and which meet the date criterion

check_inds_true(*current*, *date*, *filter_inds=None*)

Return indices for which the current state is true and which meet the date criterion

check_cin1(*genotype*)

Check for new progressions to CIN1

check_cin2(*genotype*)

Check for new progressions to CIN2

check_cin3(*genotype*)

Check for new progressions to CIN3

check_cancer(*genotype*)

Check for new progressions to cancer

check_cancer_deaths()

Check for new deaths from cancer

check_clearance(*genotype*)

Check for HPV clearance.

apply_hiv_rates(*year=None*)

Apply HIV infection rates to population

apply_death_rates(*year=None*)

Apply death rates to remove people from the population NB people are not actually removed to avoid issues with indices

add_births(*year=None*, *new_births=None*)

Add more people to the population

Specify either the year from which to retrieve the birth rate, or the absolute number of new people to add. Must specify one or the other. People are added in-place to the current *People* instance

check_migration(*year=None*)

Check if people need to immigrate/emigrate in order to make the population size correct.

make_naive(*inds*)

Make a set of people naive. This is used during dynamic resampling.

Parameters
inds (*array*) – list of people to make naive

infect(*inds*, *g=None*, *offset=None*, *dur=None*, *layer=None*)

Infect people and determine their eventual outcomes. 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
- **g** (*int*) – int of genotype to infect people with
- **offset** (*array*) – if provided, the infections will occur at the timepoint self.t+offset
- **dur** (*array*) – if provided, the duration of the infections
- **layer** (*str*) – contact layer this infection was transmitted on

Returns
number of people infected

Return type
count (int)

remove_people(*inds*, *cause=None*)

Remove people - used for death and migration

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 = hpv.Sim(pop_type='hybrid', verbose=0)
sim.run()
sim.people.story(12)
sim.people.story(795)
```

1.4.2.10 hpvsim.plotting module

Core plotting functions for simulations, multisims, and scenarios.

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*, *style_args=None*, *n_cols=None*, *grid=True*, *commaticks=True*, *setylim=True*, *log_scale=False*, *colors=None*, *labels=None*, *do_show=None*, *sep_figs=False*, *fig=None*, *ax=None*, *plot_burnin=False*, ***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*, *style_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*, *plot_burnin=False*, ***kwargs*)

Plot the results of a scenario – see Scenarios.plot() for documentation

plot_scen_age_results(*analyzer_ref=0*, *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*, *style_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*, *plot_burnin=False*, *plot_type='sns.boxplot'*, ***kwargs*)

Plot age results of a scenario

plot_result(*key*, *sim=None*, *fig_args=None*, *plot_args=None*, *axis_args=None*, *scatter_args=None*, *date_args=None*, *style_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*, *plot_burnin=False*, ***kwargs*)

Plot a single result – see `hpv.Sim.plot_result()` for documentation

plot_people(*people*, *bins=None*, *width=1.0*, *alpha=0.6*, *fig_args=None*, *axis_args=None*, *plot_args=None*, *style_args=None*, *do_show=None*, *fig=None*)

Plot statistics of a population – see People.plot() for documentation

1.4.2.11 hpvsim.population module

Defines functions for making the population.

make_people(*sim*, *popdict=None*, *reset=False*, *verbose=None*, *use_age_data=True*, *sex_ratio=0.5*, *dt_round_age=True*, *microstructure=None*, ***kwargs*)

Make the people for the simulation.

Usually called via *hpvsim.sim.Sim.initialize()*.

Parameters

- **sim** (Sim) – the simulation object; population parameters are taken from the sim object
- **popdict** (*any*) – if supplied, use this population dictionary instead of generating a new one; can be a dict or People object
- **reset** (*bool*) – whether to force population creation even if self.popdict/self.people exists
- **verbose** (*bool*) – level of detail to print
- **use_age_data** (*bool*) –
- **sex_ratio** (*bool*) –
- **dt_round_age** (*bool*) – whether to round people's ages to the nearest timestep (default true)

Returns
people

Return type
people (*People*)

make_contacts(*lno=None*, *tind=None*, *partners=None*, *current_partners=None*, *sexes=None*, *ages=None*, *debuts=None*, *is_female=None*, *is_active=None*, *mixing=None*, *layer_probs=None*, *cross_layer=None*, *pref_weight=None*, *durations=None*, *acts=None*, *age_act_pars=None*)

Make contacts for a single layer as an edgelist. This will select sexually active male partners for sexually active females using age structure if given.

1.4.2.12 hpvsim.run module

Functions and classes for running multiple HPVsim runs.

make_metapars()

Create default metaparameters for a Scenarios run

class MultiSim(*sims=None*, *base_sim=None*, *label=None*, *initialize=False*, ***kwargs*)

Bases: 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:

```
sim = hpv.Sim() # Create the sim
msim = hpv.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 = hpv.Sim() # Create the sim
msim = hpv.MultiSim(sim, n_runs=11, noise=0.1, keep_people=True) # Set up a␣
↪multisim with noise
msim.run() # Run
msim.reduce() # Compute statistics
msim.plot() # Plot

sims = [hpv.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 = hpv.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:

```
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:

```
msim = hpv.MultiSim(hpv.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:

```
msim = hpv.MultiSim(hpv.Sim())
msim.run()
msim.combine()
msim.summarize()
```

compare(*t=None*, *sim_inds=None*, *output=False*, *do_plot=False*, *show_match=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())
- **show_match** (*bool*) – whether to include a column for whether all sims match
- **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 hpv.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:

```
sim = hpv.Sim()
msim = hpv.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 hpv.load(filename) or hpv.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
scenfile (str)

Example:

```
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 hpv.load()

Returns
the loaded MultiSim object

Return type
msim (*MultiSim*)

Example:

```
msim = hpv.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:

mm1 = hpv.MultiSim.merge(msim1, msim2, base=True) mm2 = hpv.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 that many chunks; if a list return chunks of that many sims

Returns
A list of MultiSim objects

Examples:

```
m1 = hpv.MultiSim(hpv.Sim(label='sim1'), initialize=True)
m2 = hpv.MultiSim(hpv.Sim(label='sim2'), initialize=True)
m3 = hpv.MultiSim.merge(m1, m2)
m3.run()
m1b, m2b = m3.split()

msim = hpv.MultiSim(hpv.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=2) # 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:

```
msim = hpv.MultiSim(hpv.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:

```
msim = hpv.MultiSim(hpv.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:

```
msim = hpv.MultiSim(hpv.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: *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:

```
scens = hpv.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:

```
scenarios = {'base': {'name':'Base','pars': {}}, 'beta': {'name':'Beta', 'pars':
↪ {'beta': 0.020}}}
scens = hpv.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:

```
scens = hpv.Scenarios()
scens.run()
scens.plot()
```

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

Plot all age_results analyzers in a scenario together. :returns: Figure handle :rtype: fig

Example:

```
az = hpv.age_results(timepoints=['2015', '2020'], results=['hpv_incidence',
↪'total_cancers'])
base_sim = hpv.Sim(analyzers=az)
scens = hpv.Scenarios(sim=base_sim)
scens.run()
scens.plot_age_results()
```

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=None*)

Export results as XLSX

Parameters
filename (*str*) – if None, return string; else, write to file

Returns
An sc.Spreadsheet with an Excel file, or writes the file to disk

save(*scenfile=None*, *keep_sims=True*, *keep_people=False*, ***kwargs*)

Save to disk as a gzipped pickle.

Parameters

- **scenfile** (*str*) – the name or path of the file to save to; if None, uses stored

- **keep_sims** (*bool*) – whether or not to store the actual Sim objects in the Scenarios object
- **keep_people** (*bool*) – whether or not to store the population in the Sim objects (NB, very large)
- **kwargs** (*dict*) – passed to makefilepath()

Returns
the validated absolute path to the saved file

Return type
scenfile (str)

Example:

```
scens.save() # Saves to a .scens file with the date and time of creation by␣
↪default
```

static load(*scenfile*, **args*, ***kwargs*)

Load from disk from a gzipped pickle.

Parameters

- **scenfile** (*str*) – the name or path of the file to load from
- **kwargs** – passed to hpv.load()

Returns
the loaded scenarios object

Return type
scens (*Scenarios*)

Example:

```
scens = hpv.Scenarios.load('my-scenarios.scens')
```

disp(*output=False*)

Display a verbose description of the scenarios. See also scenarios.summarize() (medium length output) and scenarios.brief() (short output).

Parameters
output (*bool*) – if true, return a string instead of printing output

Example:

```
scens = hpv.Scenarios(hpv.Sim(), label='Example scenarios')
scens.run(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:

```
scens = hpv.Scenarios(hpv.Sim(), label='Example scenarios')
scens.run(verbose=0) # Run silently
scens.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:

```
scens = hpv.Scenarios(label='Example scenarios')
scens.run()
scens.brief() # Prints one-line output
```

static merge(**args*)

Merge two or more scenarios to create a single scenario object :param args: the Scenarios to merge (either a list, or separate) :type args: Scenarios

Returns
a new Scenario object

Return type
scen (*Scenarios*)

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:

```
import hpvsim as hp
sim = hpv.Sim() # Create a default simulation
sim = hpv.single_run(sim) # Run it, equivalent(ish) to sim.run()
```

multi_run(*sim*, *n_runs=4*, *reseed=None*, *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 (default: true for single, false for list of sims)
- **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:

```
import hpvsim as hpv
sim = hpv.Sim()
sims = hpv.multi_run(sim, n_runs=6, noise=0.2)
```

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

A shortcut to `hpv.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 = hpv.Sim(beta=0.01, label='Low')
s2 = hpv.Sim(beta=0.02, label='High')
hpv.parallel(s1, s2).plot()
msim = hpv.parallel([s1, s2], keep_people=True)
```

1.4.2.13 hpvsim.settings module

Define options for hpvsim, mostly plotting options. All options should be set using set() or directly, e.g.:

```
hpv.options(font_size=18)
```

To reset default options, use:

```
hpv.options('default')
```

Note: "options" is used to refer to the choices available (e.g., DPI), while "settings" is used to refer to the choices made (e.g., DPI=150).

1.4.2.14 hpvsim.sim module

Define core Sim classes

class Sim(*pars=None*, *datafile=None*, *label=None*, *popfile=None*, *people=None*, *version=None*, *hiv_datafile=None*, *art_datafile=None*, ***kwargs*)

Bases: *BaseSim*

load_data(*datafile=None*, ***kwargs*)
Load the data to calibrate against, if provided

load_hiv_data(*location=None*, *hiv_datafile=None*, *art_datafile=None*, ***kwargs*)
Load any data files that are used to create additional parameters, if provided

initialize(*reset=False*, *init_states=True*, ***kwargs*)
Perform all initializations on the sim.

layer_keys()
Attempt to retrieve the current layer keys.

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

validate_init_conditions(*init_hpv_prev*)

Initial prevalence values can be supplied with different amounts of detail. Here we flesh out any missing details so that the initial prev values are by age and genotype. We also check the prevalence values are ok.

init_genotypes()

Initialize the genotype parameters

init_immunity(*create=True*)

Initialize immunity matrices

init_results(*frequency='annual'*, *add_data=True*)

Create the main results structure. The prefix "new" is used for flow variables, i.e. counting new events (infections/deaths) on each timestep The prefix "n" is used for stock variables, i.e. counting the total number in any given state (sus/inf/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

Parameters

- **sim** (*hpv.Sim*) – a sim
- **frequency** (*str or float*) – the frequency with which to save results: accepts 'annual', 'dt', or a float which is interpreted as a fraction of a year, e.g. 0.2 will save results every 0.2 years
- **add_data** (*bool*) – whether or not to add data to the result structures

init_people(*popdict=None*, *init_states=False*, *reset=False*, *verbose=None*, ***kwargs*)

Create the people and the network.

Use `init_states=False` for creating a fresh People object for use in future simulations

Parameters

- **popdict** (*any*) – pre-generated people of various formats.
- **init_states** (*bool*) – whether to initialize states (default false when called directly)
- **reset** (*bool*) – whether to regenerate the people even if they already exist
- **verbose** (*int*) – detail to print
- **kwargs** (*dict*) – passed to hpv.make_people()

init_interventions()

Initialize and validate the interventions

init_analyzers()

Initialize the analyzers

finalize_analyzers()

init_states(*age_brackets=None*, *init_hpv_prev=None*, *init_cin_prev=None*, *init_cancer_prev=None*)

Initialize prior immunity and seed infections

step()

Step through time and update values

run(*do_plot=False*, *until=None*, *restore_pars=True*, *reset_seed=True*, *verbose=None*)

Run the model once

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.

compute_summary(*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

- **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. 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:

```
sim = hpv.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
```

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

Plot the outputs of the model

compute_fit()

Compute fit between model and data.

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.

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:

```
s1 = hpv.Sim(rand_seed=1).run()
s2 = hpv.Sim(rand_seed=2).run()
hpv.diff_sims(s1, s2)
```

1.4.2.15 hpvsim.utils module

Numerical utilities for running hpvsim.

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=∞
- 'beta' : beta distribution with alpha=par1 and beta=par2;
- 'gamma' : gamma distribution with shape=par1 and scale=par2;

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:

```
hpv.sample() # returns Unif(0,1)
hpv.sample(dist='normal', par1=3, par2=0.5) # returns Normal(=3, =0.5)
hpv.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. This function also resets Python's built-in random number generated.

Parameters
seed (*int*) – the random seed

n_binomial(*prob*, *n*)

Perform multiple binomial (Bernolli) 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:

```
outcomes = hpv.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 = hpv.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 = hpv.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 = hpv.n_multinomial(np.ones(6)/6.0, 50)+1 # Return 50 die-rolls
```

poisson(*rate*)

A Poisson trial.

Parameters

rate (*float*) – the rate of the Poisson process

Example:

```
outcome = hpv.poisson(100) # Single Poisson trial with mean 100
```

n_poisson(*rate*, *n*)

An array of Poisson trials.

Parameters

- **rate** (*float*) – the rate of the Poisson process (mean)
- **n** (*int*) – number of trials

Example:

```
outcomes = hpv.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 hpv.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, ∞ = Poisson
- **n** (*int*) – number of trials
- **step** (*float*) – the step size to use if non-integer outputs are desired

Example:

```
outcomes = hpv.n_neg_binomial(100, 1, 50) # 50 negative binomial trials with mean␣
↪100 and dispersion roughly equal to mean (large-mean limit)
outcomes = hpv.n_neg_binomial(1, 100, 20) # 20 negative binomial trials with mean 1␣
↪and dispersion still roughly equal to mean (approximately Poisson)
```

choose(*max_n*, *n*)

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:

```
choices = hpv.choose(5, 2) # choose 2 out of 5 people with equal probability␣
↪(without repeats)
```

choose_r(*max_n*, *n*)

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:

```
choices = hpv.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:

```
choices = hpv.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:

```
inds = hpv.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

Example:

```
inds = hpv.false(np.array([1,0,0,1,1,0,1]))
```

defined(*arr*)

Returns the indices of the values of the array that are not-nan.

Parameters
arr (*array*) – any array

Example:

```
inds = hpv.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

Example:

```
inds = hpv.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

Example:

```
inds = hpv.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

Example:

```
inds = hpv.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

Example:

```
inds = hpv.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:

```
inds = hpv.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:

```
inds = hpv.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:

```
inds = hpv.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:

```
inds = hpv.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:

```
inds = hpv.iundefinedi(np.array([4,np.nan,0,np.nan,np.nan,4,7,4,np.nan]), inds=np.
↪array([0,1,3,5]))
```

dtround(*arr*, *dt*, *ceil=True*)

Rounds the values in the array to the nearest timestep

Parameters

- **arr** (*array*) – any array
- **dt** (*float*) – float, usually representing a timestep in years

Example:

```
dtround = hpv.dtround(np.array([0.23,0.61,20.53])) # Returns array([0.2, 0.6, 20.6])
dtround = hpv.dtround(np.array([0.23,0.61,20.53]),ceil=True) # Returns array([0.4,␣
↪0.8, 20.6])
```

find_cutoff(*duration_cutoffs*, *duration*)

Find which duration bin each ind belongs to.

1.4.2.16 hpvsim.version module

Version and license information.

PYTHON MODULE INDEX

h

INDEX

D

E

F

G

H

I

K

L

M

N

P

Q

R

S

T

U

V

W