airspeed velocity Documentation

Release 0.6.2

Michael Droettboom

Feb 12, 2024

CONTENTS

airspeed velocity (`asv`) is a tool for benchmarking Python packages over their lifetime. Runtime, memory consumption and even custom-computed values may be tracked. The results are displayed in an interactive web frontend that requires only a basic static webserver to host.

See examples of Airspeed Velocity websites: astropy, numpy, scipy.

License: BSD three-clause license.

Releases: https://pypi.python.org/pypi/asv

Development: https://github.com/airspeed-velocity/asv

CHAPTER

ONE

INSTALLING AIRSPEED VELOCITY

airspeed velocity is known to work on Linux, Mac OS-X, and Windows. It is known to work with Python 3.7 and higher. It works also with PyPy.

airspeed velocity is a standard Python package, and the latest released version may be installed in the standard way from PyPI:

```
pip install asv
```

The development version can be installed by cloning the source repository and running `pip install .` inside it, or by `pip install git+https://github.com/airspeed-velocity/asv`.

The requirements should be automatically installed. If they aren't installed automatically, for example due to networking restrictions, the `python` requirements are as noted in the `pyproject.toml`.

For managing the environments, one of the following packages is required:

- libmambapy, which is typically part of `mamba`. In this case `conda` must be present too.
- virtualenv, which is required since venv is not compatible with other versions of Python.
- An anaconda or miniconda installation, with the `conda` command available on your path.

Note: `libmambapy` is the fastest for situations where non-pythonic dependencies are required. Anaconda or miniconda is slower but still preferred if the project involves a lot of compiled C/C++ extensions and are available in the `conda` repository, since `conda` will be able to fetch precompiled binaries for these dependencies in many cases. Using `virtualenv`, dependencies without precompiled wheels usually have to be compiled every time the environments are set up.

1.1 Optional optimization

If your project being benchmarked contains C, C++, Objective-C or Cython, consider installing `ccache`. ccache is a compiler cache that speeds up compilation time when the same objects are repeatedly compiled. In **airspeed velocity**, the project being benchmarked is recompiled at many different points in its history, often with only minor changes to the source code, so `ccache` can help speed up the total benchmarking time considerably.

1.2 Running the self-tests

The self tests are based on pytest. If you don't have it installed, and you have a connection to the Internet, it will be installed automatically.

To run **airspeed velocity**'s self tests:

```
pytest
```

CHAPTER

TWO

USING AIRSPEED VELOCITY

airspeed velocity is designed to benchmark a single project over its lifetime using a given set of benchmarks. Below, we use the phrase "project" to refer to the project being benchmarked, and "benchmark suite" to refer to the set of benchmarks – i.e., little snippets of code that are timed – being run against the project.

The benchmark suite may live inside the project's repository, or it may reside in a separate repository – the choice is up to you and is primarily a matter of style or policy. Note also that the result data is stored in JSON files alongside the benchmark suite and may grow quite large, and you may want to plan where to store it.

You can interact with **airspeed velocity** through the `asv` command. Like `git`, the `asv` command has a number of "subcommands" for performing various actions on your benchmarking project.

2.1 Setting up a new benchmarking project

The first thing to do is to set up an **airspeed velocity** benchmark suite for your project. It must contain, at a minimum, a single configuration file, `asv.conf.json`, and a directory tree of Python files containing benchmarks.

The `asv quickstart` command can be used to create a new benchmarking suite. Change to the directory where you would like your new benchmarking suite to be created and run:

```
$ asv quickstart
· Setting up new Airspeed Velocity benchmark suite.

Which of the following template layouts to use:
(1) benchmark suite at the top level of the project repository
(2) benchmark suite in a separate repository

Layout to use? [1/2] 1

· Edit asv.conf.json to get started.
```

Answer '1' if you want a default configuration suitable for putting the benchmark suite on the top level of the same repository where your project is, or '2' to get default configuration for putting it in a separate repository.

Now that you have the bare bones of a benchmarking suite, let's edit the configuration file, `asv.conf.json`. Like most files that **airspeed velocity** uses and generates, it is a JSON file.

There are comments in the file describing what each of the elements do, and there is also a *asv.conf.json reference* with more details. The values that will most likely need to be changed for any benchmarking suite are:

- `project`: The Python package name of the project being benchmarked.
- `project_url`: The project's homepage.

- `repo`: The URL or path to the DVCS repository for the project. This should be a read-only URL so that anyone, even those without commit rights to the repository, can run the benchmarks. For a project on github, for example, the URL would look like: `https://github.com/airspeed-velocity/asv.git`

  The value can also be a path, relative to the location of the configuration file. For example, if the benchmarks are stored in the same repository as the project itself, and the configuration file is located at `benchmarks/asv.conf.json` inside the repository, you can set `"repo": ".."` to use the local repository.

- `show_commit_url`: The base of URLs used to display commits for the project. This allows users to click on a commit in the web interface and have it display the contents of that commit. For a github project, the URL is of the form `http://github.com/$OWNER/$REPO/commit/`.

- `environment_type`: The tool used to create environments. May be `conda` or `virtualenv` or `mamba`. If Conda supports the dependencies you need, that is the recommended method. Mamba is faster but needs a newer Python version (3.8 or greater). See *Environments* for more information.

- `matrix`: Dependencies you want to preinstall into the environment where benchmarks are run.

- `default_benchmark_timeout`: The maximum time in seconds that a benchmark is allowed to run before it is terminated. This is useful to prevent benchmarks from running indefinitely if they get stuck. The default is 60 seconds.

The rest of the values can usually be left to their defaults, unless you want to benchmark against multiple versions of Python or multiple versions of third-party dependencies, or if your package needs nonstandard installation commands.

Once you've set up the project's configuration, you'll need to write some benchmarks. The benchmarks live in Python files in the `benchmarks` directory. The `quickstart` command has created a single example benchmark file already in `benchmarks/benchmarks.py`:

```
# Write the benchmarking functions here.
# See "Writing benchmarks" in the asv docs for more information.


class TimeSuite:
    """
    An example benchmark that times the performance of various kinds
    of iterating over dictionaries in Python.
    """
    def setup(self):
        self.d = {}
        for x in range(500):
            self.d[x] = None

    def time_keys(self):
        for key in self.d.keys():
            pass

    def time_values(self):
        for value in self.d.values():
            pass

    def time_range(self):
        d = self.d
        for key in range(500):
            d[key]
```

(continues on next page)

(continued from previous page)

```
class MemSuite:
    def mem_list(self):
        return [0] * 256
```

You'll want to replace these benchmarks with your own. See *Writing benchmarks* for more information.

2.2 Running benchmarks

Benchmarks are run using the `asv run` subcommand.

Let's start by just benchmarking the latest commit on the current `main` branch of the project:

```
$ asv run
```

2.2.1 Machine information

If this is the first time using `asv run` on a given machine, (which it probably is, if you're following along), you will be prompted for information about the machine, such as its platform, cpu and memory. **airspeed velocity** will try to make reasonable guesses, so it's usually ok to just press `Enter` to accept each default value. This information is stored in the `~/.asv-machine.json` file in your home directory:

```
I will now ask you some questions about this machine to identify
it in the benchmarks.

1. machine: A unique name to identify this machine in the results.
   May be anything, as long as it is unique across all the
   machines used to benchmark this project.  NOTE: If changed from
   the default, it will no longer match the hostname of this
   machine, and you may need to explicitly use the --machine
   argument to asv.
machine [cheetah]:
2. os: The OS type and version of this machine.  For example,
   'Macintosh OS-X 10.8'.
os [Linux 3.17.6-300.fc21.x86_64]:
3. arch: The generic CPU architecture of this machine.  For
   example, 'i386' or 'x86_64'.
arch [x86_64]:
4. cpu: A specific description of the CPU of this machine,
   including its speed and class.  For example, 'Intel(R) Core(TM)
   i5-2520M CPU @ 2.50GHz (4 cores)'.
cpu [Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz]:
5. ram: The amount of physical RAM on this machine.  For example,
   '4GB'.
ram [8055476]:
```

Note: If you ever need to update the machine information later, you can run `asv machine`.

Note: By default, the name of the machine is determined from your hostname. If you have a hostname that frequently

changes, and your `~/.asv-machine.json` file contains more than one entry, you will need to use the `--machine` argument to `asv run` and similar commands.

2.2.2 Environments

Next, the Python environments to run the benchmarks are set up. `asv` always runs its benchmarks in an environment that it creates, in order to not change any of your existing Python environments. One environment will be set up for each of the combinations of Python versions and the matrix of project dependencies, if any. The first time this is run, this may take some time, as many files are copied over and dependencies are installed into the environment. The environments are stored in the `env` directory so that the next time the benchmarks are run, things will start much faster.

Environments can be created using different tools. By default, `asv` ships with support for anaconda, mamba, and virtualenv, though plugins may be installed to support other environment tools. The `environment_type` key in `asv.conf.json` is used to select the tool used to create environments.

When using `virtualenv`, `asv` does not build Python interpreters for you, but it expects to find the Python versions specified in the `asv.conf.json` file available on the `PATH`. For example, if the `asv.conf.json` file has:

```
"pythons": ["3.7", "3.12"]
```

then it will use the executables named `python3.7` and `python3.12` on the path. There are many ways to get multiple versions of Python installed – your package manager, `apt-get`, `yum`, `MacPorts` or `homebrew` probably has them, or you can also use pyenv.

The `virtualenv` environment also supports PyPy. You can specify `"pypy"` or `"pypy3"` as a Python version number in the `"pythons"` list. Note that PyPy must also be installed and available on your `PATH`.

2.2.3 Benchmarking

Finally, the benchmarks are run:

```
$ asv run
· Cloning project.
· Fetching recent changes
· Creating environments......
· Discovering benchmarks
·· Uninstalling from virtualenv-py3.7
·· Building 4238c44d <main> for virtualenv-py3.7
·· Installing into virtualenv-py3.7.
· Running 10 total benchmarks (1 commits * 2 environments * 5 benchmarks)
[  0.00%] · For project commit 4238c44d <main>:
[  0.00%] ·· Building for virtualenv-py3.7.
[  0.00%] ·· Benchmarking virtualenv-py3.7
[ 10.00%] ··· Running (benchmarks.TimeSuite.time_iterkeys--)....
[ 30.00%] ··· benchmarks.MemSuite.mem_list                              2.42k
[ 35.00%] ··· benchmarks.TimeSuite.time_iterkeys                    11.1±0.01s
[ 40.00%] ··· benchmarks.TimeSuite.time_keys                        11.2±0.01s
[ 45.00%] ··· benchmarks.TimeSuite.time_range                       32.9±0.01s
[ 50.00%] ··· benchmarks.TimeSuite.time_xrange                      30.3±0.01s
[ 50.00%] ·· Building for virtualenv-py3.12..
[ 50.00%] ·· Benchmarking virtualenv-py3.12
[ 60.00%] ··· Running (benchmarks.TimeSuite.time_iterkeys--)....
```

(continues on next page)

(continued from previous page)

```
[ 80.00%] ··· benchmarks.MemSuite.mem_list                          2.11k
[ 85.00%] ··· benchmarks.TimeSuite.time_iterkeys                    failed
[ 90.00%] ··· benchmarks.TimeSuite.time_keys                      9.07±0.5s
[ 95.00%] ··· benchmarks.TimeSuite.time_range                    35.5±0.01s
[100.00%] ··· benchmarks.TimeSuite.time_xrange                      failed
```

To improve reproducibility, each benchmark is run in its own process.

The results of each benchmark are displayed in the output and also recorded on disk. For timing benchmarks, the median and interquartile range of collected measurements are displayed. Note that the results may vary on slow time scales due to CPU frequency scaling, heat management, and system load, and this variability is not necessarily captured by a single run. How to deal with this is discussed in *Tuning timing measurements*.

The killer feature of **airspeed velocity** is that it can track the benchmark performance of your project over time. The `range` argument to `asv run` specifies a range of commits that should be benchmarked. The value of this argument is passed directly to either `git log` or to the Mercurial log command to get the set of commits, so it actually has a very powerful syntax defined in the gitrevisions manpage, or the revsets help section for Mercurial.

For example, in a Git repository, one can test a range of commits on a particular branch since branching off main:

```
asv run main..mybranch
```

Or, to benchmark all of the commits since a particular tag (`v0.1`):

```
asv run v0.1..main
```

To benchmark a single commit, or tag, use `^!` (git):

```
asv run v0.1^!
```

Corresponding examples for Mercurial using the revsets specification are also possible.

In many cases, this may result in more commits than you are able to benchmark in a reasonable amount of time. In that case, the `--steps` argument is helpful. It specifies the maximum number of commits you want to test, and it will evenly space them over the specified range.

You can benchmark all commits in the repository by using:

```
asv run ALL
```

You may also want to benchmark every commit that has already been benchmarked on all the other machines. For that, use:

```
asv run EXISTING
```

You can benchmark all commits since the last one that was benchmarked on this machine. This is useful for running in nightly cron jobs:

```
asv run NEW
```

You can also benchmark a specific set of commits listed explicitly in a file (one commit hash per line):

```
asv run HASHFILE:hashestobenchmark.txt
```

Finally, you can also benchmark all commits that have not yet been benchmarked for this machine:

```
asv run --skip-existing-commits ALL
```

Note: You can also do a validity check for the benchmark suite without running benchmarks, using `asv check`.

The results are stored as JSON files in the directory `results/$MACHINE`, where `$MACHINE` is the unique machine name that was set up in your `~/.asv-machine.json` file. In order to combine results from multiple machines, you can for example store the results in separate repositories, for example git submodules, alongside the results from other machines. These results are then collated and "published" altogether into a single interactive website for viewing (see *Viewing the results*).

You can also continue to generate benchmark results for other commits, or for new benchmarks and continue to throw them in the `results` directory. **airspeed velocity** is designed from the ground up to handle missing data where certain benchmarks have yet to be performed – it's entirely up to you how often you want to generate results, and on which commits and in which configurations.

2.3 Viewing the results

You can use the *asv show* command to display results from previous runs on the command line:

```
$ asv show main
Commit: 4238c44d <main>

benchmarks.MemSuite.mem_list [mymachine/virtualenv-py3.7]
  2.42k
  started: 2018-08-19 18:46:47, duration: 1.00s

benchmarks.TimeSuite.time_iterkeys [mymachine/virtualenv-py3.7]
  11.1±0.06s
  started: 2018-08-19 18:46:47, duration: 1.00s

...
```

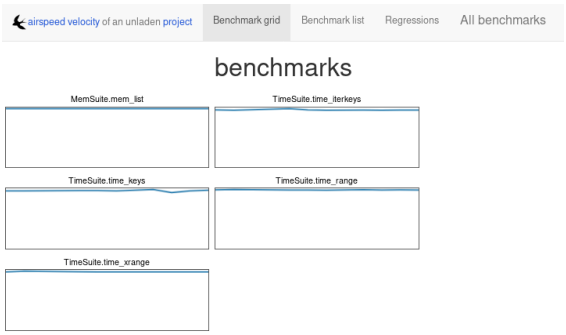
To collate a set of results into a viewable website, run:

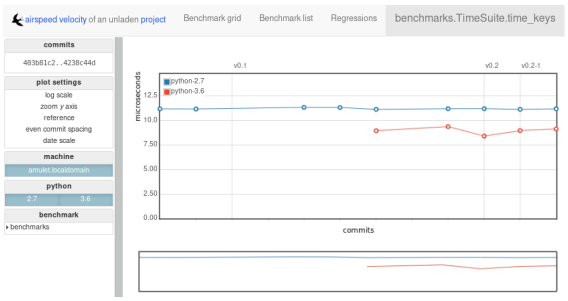
```
asv publish
```

This will put a tree of files in the `html` directory. This website can not be viewed directly from the local filesystem, since web browsers do not support AJAX requests to the local filesystem. Instead, **airspeed velocity** provides a simple static webserver that can be used to preview the website. Just run:

```
asv preview
```

and open the URL that is displayed at the console. Press Ctrl+C to stop serving.





To share the website on the open internet, simply put the files in the `html` directory on any webserver that can serve static content. Github Pages works quite well, for example. For using Github Pages, `asv` includes the convenience command `asv gh-pages` to put the results to the `gh-pages` branch and push them to Github. See *asv gh-pages –help* for details.

2.4 Managing the results database

The `asv rm` command can be used to remove benchmarks from the database. The command takes an arbitrary number of `key=value` entries that are "and"ed together to determine which benchmarks to remove.

The keys may be one of:

- `benchmark`: A benchmark name
- `python`: The version of python
- `commit_hash`: The commit hash
- machine-related: `machine`, `arch`, `cpu`, `os`, `ram`
- environment-related: a name of a dependency, e.g. `numpy`

The values are glob patterns, as supported by the Python standard library module `fnmatch`. So, for example, to remove all benchmarks in the `time_units` module:

```
asv rm "benchmark=time_units.*"
```

Note the double quotes around the entry to prevent the shell from expanding the * itself.

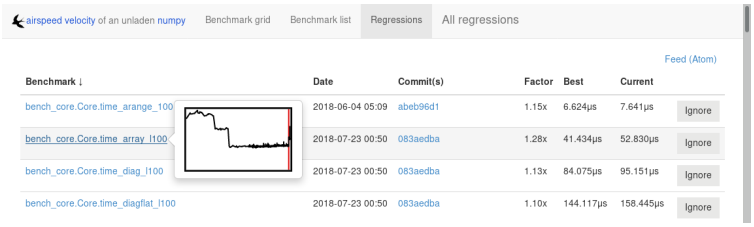
The `asv rm` command will prompt before performing any operations. Passing the `-y` option will skip the prompt.

Here is a more complex example, to remove all of the benchmarks on Python 3.7 and the machine named `giraffe`:

```
asv rm python=3.7 machine=giraffe
```

2.5 Finding a commit that produces a large regression

airspeed velocity detects statistically significant decreases of performance automatically based on the available data when you run `asv publish`. The results can be inspected via the web interface, clicking the "Regressions" tab on the web site. The results include links to each benchmark graph deemed to contain a decrease in performance, the commits where the regressions were estimated to occur, and other potentially useful information.



However, since benchmarking can be rather time consuming, it's likely that you're only benchmarking a subset of all commits in the repository. When you discover from the graph that the runtime between commit A and commit B suddenly doubles, you don't necessarily know which particular commit in that range is the likely culprit. `asv find` can be used to help find a commit within that range that produced a large regression using a binary search. You can select a range of commits easily from the web interface by dragging a box around the commits in question. The commit hashes associated with that range is then displayed in the "commits" section of the sidebar. We'll copy and paste this commit range into the commandline arguments of the `asv find` command, along with the name of a single benchmark to use. The output below is truncated to show how the search progresses:

```
$ asv find 05d4f83d..b96fcc53 time_coordinates.time_latitude
- Running approximately 10 benchmarks within 1156 commits
- Testing <----------------------------0---------------------------->
- Testing <-------------0-------------->------------------------------
- Testing --------------<-------0------>------------------------------
- Testing --------------<---0--->--------------------------------------
- Testing --------------<-0->----------------------------------------
- Testing --------------<0>------------------------------------------
- Testing --------------<>--------------------------------------------
- Greatest regression found: 2918f61e
```

The result, `2918f61e` is the commit found with the largest regression, using the binary search.

Note: The binary search used by `asv find` will only be effective when the runtimes over the range are more-or-less monotonic. If there is a lot of variation within that range, it may find only a local maximum, rather than the global maximum. For best results, use a reasonably small commit range.

2.6 Running a benchmark in the profiler

airspeed velocity can oftentimes tell you *if* something got slower, but it can't really tell you *why* it got slower. That's where a profiler comes in. **airspeed velocity** has features to easily run a given benchmark in the Python standard library's `cProfile` profiler, and then open the profiling data in the tool of your choice.

The `asv profile` command profiles a given benchmark on a given revision of the project.

Note: You can also pass the `--profile` option to `asv run`. In addition to running the benchmarks as usual, it also runs them again in the `cProfile` profiler and save the results. `asv profile` will use this data, if found, rather than needing to profile the benchmark each time. However, it's important to note that profiler data contains absolute paths to the source code, so they are generally not portable between machines.

`asv profile` takes as arguments the name of the benchmark and the hash, tag or branch of the project to run it in. Below is a real world example of testing the `astropy` project. By default, a simple table summary of profiling results is displayed:

```
> asv profile time_units.time_very_simple_unit_parse 10fc29cb

    8700042 function calls in 6.844 seconds

 Ordered by: cumulative time

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.000    0.000    6.844    6.844 asv/benchmark.py:171(method_caller)
      1    0.000    0.000    6.844    6.844 asv/benchmark.py:197(run)
      1    0.000    0.000    6.844    6.844 /usr/lib64/python3.7/timeit.py:201(repeat)
      3    0.000    0.000    6.844    2.281 /usr/lib64/python3.7/timeit.py:178(timeit)
      3    0.104    0.035    6.844    2.281 /usr/lib64/python3.7/timeit.py:96(inner)
 300000    0.398    0.000    6.740    0.000 benchmarks/time_units.py:20(time_very_simple_
↪unit_parse)
 300000    1.550    0.000    6.342    0.000 astropy/units/core.py:1673(__call__)
 300000    0.495    0.000    2.416    0.000 astropy/units/format/generic.py:361(parse)
 300000    1.023    0.000    1.841    0.000 astropy/units/format/__init__.py:31(get_
↪format)
 300000    0.168    0.000    1.283    0.000 astropy/units/format/generic.py:374(_do_
↪parse)
 300000    0.986    0.000    1.115    0.000 astropy/units/format/generic.py:345(_parse_
↪unit)
3000002    0.735    0.000    0.735    0.000 {isinstance}
 300000    0.403    0.000    0.403    0.000 {method 'decode' of 'str' objects}
 300000    0.216    0.000    0.216    0.000 astropy/units/format/generic.py:32(__init__)
 300000    0.152    0.000    0.188    0.000 /usr/lib64/python3.7/inspect.py:59(isclass)
 900000    0.170    0.000    0.170    0.000 {method 'lower' of 'unicode' objects}
 300000    0.133    0.000    0.133    0.000 {method 'count' of 'unicode' objects}
 300000    0.078    0.000    0.078    0.000 astropy/units/core.py:272(get_current_unit_
↪registry)
 300000    0.076    0.000    0.076    0.000 {issubclass}
 300000    0.052    0.000    0.052    0.000 astropy/units/core.py:131(registry)
 300000    0.038    0.000    0.038    0.000 {method 'strip' of 'str' objects}
 300003    0.037    0.000    0.037    0.000 {globals}
 300000    0.033    0.000    0.033    0.000 {len}
      3    0.000    0.000    0.000    0.000 /usr/lib64/python3.7/timeit.py:143(setup)
      1    0.000    0.000    0.000    0.000 /usr/lib64/python3.7/timeit.py:121(__init__)
      6    0.000    0.000    0.000    0.000 {time.time}
      1    0.000    0.000    0.000    0.000 {min}
      1    0.000    0.000    0.000    0.000 {range}
      1    0.000    0.000    0.000    0.000 {hasattr}
      1    0.000    0.000    0.000    0.000 /usr/lib64/python3.7/timeit.py:94(_template_
↪func)
      3    0.000    0.000    0.000    0.000 {gc.enable}
      3    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
      3    0.000    0.000    0.000    0.000 {gc.disable}
      1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'␣
↪objects}
      3    0.000    0.000    0.000    0.000 {gc.isenabled}
      1    0.000    0.000    0.000    0.000 <string>:1(<module>)
```

Navigating these sorts of results can be tricky, and generally you want to open the results in a GUI tool, such as RunSnakeRun or snakeviz. For example, by passing the `--gui=runsnake` to `asv profile`, the profile is collected

(or extracted) and opened in the RunSnakeRun tool.

Note: To make sure the line numbers in the profiling data correctly match the source files being viewed, the correct revision of the project is checked out before opening it in the external GUI tool.

You can also get the raw profiling data by using the `--output` argument to `asv profile`.

Note: Since the method name is passed as a regex, parenthesis need to be escaped, e.g. `asv profile 'benchmarks.MyBench.time_sort\(500\)' HEAD --gui snakeviz`

See *asv profile* for more options.

To extract information from `--profile` runs of `asv`:

```
$ python
import asv
results_asv = asv.results.iter_results(".asv")
res_objects = list(results_asv)
prof_data = res_objects[0].get_profile_stats('benchmarks.MyBench.time_sort')
prof_data.strip_dirs() # Remove machine specific info
prof_data.sort_stats('cumulative').print_stats()
```

Where different benchmarks may be used. A specific `json` may also be loaded directly with `asv.results.Results.load(<json_path>)`, after which `get_profile_stats` can be used.

2.7 Comparing the benchmarking results for two revisions

In some cases, you may want to directly compare the results for two specific revisions of the project. You can do so with the `compare` command:

```
$ asv compare v0.1 v0.2
All benchmarks:

       before           after         ratio
     [3bfda9c6]       [bf719488]
     <v0.1>           <v0.2>
            40.4m            40.4m     1.00  benchmarks.MemSuite.mem_list [amulet.
↪localdomain/virtualenv-py3.7-numpy]
           failed            35.2m      n/a  benchmarks.MemSuite.mem_list [amulet.
↪localdomain/virtualenv-py3.12-numpy]
      11.5±0.08s         11.0±0s     0.96  benchmarks.TimeSuite.time_iterkeys [amulet.
↪localdomain/virtualenv-py3.7-numpy]
           failed          failed      n/a  benchmarks.TimeSuite.time_iterkeys [amulet.
↪localdomain/virtualenv-py3.12-numpy]
         11.5±1s      11.2±0.02s     0.97  benchmarks.TimeSuite.time_keys [amulet.
↪localdomain/virtualenv-py3.7-numpy]
           failed      8.40±0.02s      n/a  benchmarks.TimeSuite.time_keys [amulet.
↪localdomain/virtualenv-py3.12-numpy]
      34.6±0.09s      32.9±0.01s     0.95  benchmarks.TimeSuite.time_range [amulet.
↪localdomain/virtualenv-py3.7-numpy]
```

(continues on next page)

(continued from previous page)

```
          failed      35.6±0.05s      n/a  benchmarks.TimeSuite.time_range [amulet.
↪localdomain/virtualenv-py3.12-numpy]
      31.6±0.1s      30.2±0.02s     0.95  benchmarks.TimeSuite.time_xrange [amulet.
↪localdomain/virtualenv-py3.7-numpy]
          failed           failed      n/a  benchmarks.TimeSuite.time_xrange [amulet.
↪localdomain/virtualenv-py3.12-numpy]
```

This will show the times for each benchmark for the first and second revision, and the ratio of the second to the first. In addition, the benchmarks will be color coded green and red if the benchmark improves or worsens more than a certain threshold factor, which defaults to 1.1 (that is, benchmarks that improve by more than 10% or worsen by 10% are color coded). The threshold can be set with the `--factor=value` option. Finally, the benchmarks can be split into ones that have improved, stayed the same, and worsened, using the same threshold using the `--split` option. See *asv compare* for more.

CHAPTER

THREE

WRITING BENCHMARKS

Note: The asv_samples repository has complete examples of benchmarks along with continuous integration and can serve as a reference for writing and working with benchmarks.

Benchmarks are stored in a Python package, i.e. collection of `.py` files in the benchmark suite's `benchmark` directory (as defined by `benchmark_dir` in the `asv.conf.json` file). The package may contain arbitrarily nested subpackages, contents of which will also be used, regardless of the file names.

Within each `.py` file, each benchmark is a function or method. The name of the function must have a special prefix, depending on the type of benchmark. `asv` understands how to handle the prefix in either `CamelCase` or lowercase with underscores. For example, to create a timing benchmark, the following are equivalent:

```
def time_range():
    for i in range(1000):
        pass

def TimeRange():
    for i in range(1000):
        pass
```

Benchmarks may be organized into methods of classes if desired:

```
class Suite:
    def time_range(self):
        for i in range(1000):
            pass

    def time_xrange(self):
        for i in xrange(1000):
            pass
```

3.1 Running benchmarks during development

There are some options to `asv run` that may be useful when writing benchmarks.

You may find that `asv run` spends a lot of time setting up the environment each time. You can have `asv run` use an existing Python environment that already has the benchmarked project and all of its dependencies installed. Use the `--python` argument to specify a Python environment to use:

```
asv run --python=python
```

If you don't care about getting accurate timings, but just want to ensure the code is running, you can add the `--quick` argument, which will run each benchmark only once:

```
asv run --quick
```

In order to display the standard error output (this includes exception tracebacks) that your benchmarks may produce, pass the `--show-stderr` flag:

```
asv run --show-stderr
```

Finally, a quick way to test out the benchmark suite before doing a full run is to use all of these features together with:

```
asv run --python=same --quick --show-stderr --dry-run
```

Changed in version 0.6.0: This replaces the now removed `asv dev` command.

You may also want to only do a basic check whether the benchmark suite is well-formatted, without actually running any benchmarks:

```
asv check --python=same
```

3.2 Setup and teardown functions

If initialization needs to be performed that should not be included in the timing of the benchmark, include that code in a `setup` method on the class, or add an attribute called `setup` to a free function.

For example:

```
class Suite:
    def setup(self):
        # load data from a file
        with open("/usr/share/words.txt", "r") as fd:
            self.words = fd.readlines()

    def time_upper(self):
        for word in self.words:
            word.upper()

# or equivalently...

words = []
def my_setup():
    global words
```

(continues on next page)

(continued from previous page)

```
    with open("/usr/share/words.txt", "r") as fd:
        words = fd.readlines()

def time_upper():
    for word in words:
        word.upper()
time_upper.setup = my_setup
```

You can also include a module-level `setup` function, which will be run for every benchmark within the module, prior to any `setup` assigned specifically to each function.

Similarly, benchmarks can also have a `teardown` function that is run after the benchmark. This is useful if, for example, you need to clean up any changes made to the filesystem.

Note that although different benchmarks run in separate processes, for a given benchmark repeated measurement (cf. `repeat` attribute) and profiling occur within the same process. For these cases, the setup and teardown routines are run multiple times in the same process.

If `setup` raises a `NotImplementedError`, the benchmark is marked as skipped.

Note: For `asv` versions before 0.5 it was possible to raise `NotImplementedError` from any existing benchmark during its execution and the benchmark would be marked as skipped. This behavior was deprecated from 0.5 onwards.

Changed in version 0.6.0: To keep compatibility with earlier versions, it is possible to raise `asv_runner.benchmark.mark.SkipNotImplemented` anywhere within a Benchmark, though users are advised to use the skip decorators instead as they are faster and do not execute the `setup` function. See *Skipping benchmarks* for more details.

The `setup` method is run multiple times, for each benchmark and for each repeat. If the `setup` is especially expensive, the `setup_cache` method may be used instead, which only performs the setup calculation once and then caches the result to disk. It is run only once also for repeated benchmarks and profiling, unlike `setup`. `setup_cache` can persist the data for the benchmarks it applies to in two ways:

- Returning a data structure, which `asv` pickles to disk, and then loads and passes it as the first argument to each benchmark.
- Saving files to the current working directory (which is a temporary directory managed by `asv`) which are then explicitly loaded in each benchmark process. It is probably best to load the data in a `setup` method so the loading time is not included in the timing of the benchmark.

A separate cache is used for each environment and each commit of the project being tested and is thrown out between benchmark runs.

For example, caching data in a pickle:

```
class Suite:
    def setup_cache(self):
        fib = [1, 1]
        for i in range(100):
            fib.append(fib[-2] + fib[-1])
        return fib

    def track_fib(self, fib):
        return fib[-1]
```

As another example, explicitly saving data in a file:

```
class Suite:
    def setup_cache(self):
        with open("test.dat", "wb") as fd:
            for i in range(100):
                fd.write('{0}\n'.format(i))

    def setup(self):
        with open("test.dat", "rb") as fd:
            self.data = [int(x) for x in fd.readlines()]

    def track_numbers(self):
        return len(self.data)
```

The `setup_cache` timeout can be specified by setting the `.timeout` attribute of the `setup_cache` function. The default value is the maximum of the timeouts of the benchmarks using it.

Note: Changed in version 0.6.0: The configuration option `default_benchmark_timeout` can also be set for a project-wide timeout.

3.3 Benchmark attributes

Each benchmark can have a number of arbitrary attributes assigned to it. The attributes that `asv` understands depends on the type of benchmark and are defined below. For free functions, just assign the attribute to the function. For methods, include the attribute at the class level. For example, the following are equivalent:

```
def time_range():
    for i in range(1000):
        pass
time_range.timeout = 120.0

class Suite:
    timeout = 120.0

    def time_range(self):
        for i in range(1000):
            pass
```

For the list of attributes, see *Benchmark types and attributes*.

3.4 Parameterized benchmarks

You might want to run a single benchmark for multiple values of some parameter. This can be done by adding a `params` attribute to the benchmark object:

```
def time_range(n):
   for i in range(n):
       pass
time_range.params = [0, 10, 20, 30]
```

This will also make the setup and teardown functions parameterized:

```
class Suite:
    params = [0, 10, 20]

    def setup(self, n):
        self.obj = range(n)

    def teardown(self, n):
        del self.obj

    def time_range_iter(self, n):
        for i in self.obj:
            pass
```

If `setup` raises a `NotImplementedError`, the benchmark is marked as skipped for the parameter values in question.

The parameter values can be any Python objects. However, it is often best to use only strings or numbers, because these have simple unambiguous text representations. In the event the `repr()` output is non-unique, the representations will be made unique by suffixing an integer identifier corresponding to the order of appearance.

When you have multiple parameters, the test is run for all of their combinations:

```
def time_ranges(n, func_name):
    f = {'range': range, 'arange': numpy.arange}[func_name]
    for i in f(n):
        pass

time_ranges.params = ([10, 1000], ['range', 'arange'])
```

The test will be run for parameters `(10, 'range')`, `(10, 'arange')`, `(1000, 'range')`, `(1000, 'arange')`.

You can also provide informative names for the parameters:

```
time_ranges.param_names = ['n', 'function']
```

These will appear in the test output; if not provided you get default names such as "param1", "param2".

Note that `setup_cache` is not parameterized.

3.5 Skipping benchmarks

Note: This section is only applicable from version `0.6.0` on-wards

Conversely, it is possible (typically due to high setup times) that one might want to skip some benchmarks all-together, or just for some sets of parameters. This is accomplished by an attribute `skip_params`, which can be used with the decorator `@skip_for_params` as:

```
from asv_runner.benchmarks.mark import skip_for_params
@skip_for_params([(10, 'arange'), (1000, 'range')])
def time_ranges(n, func_name):
    f = {'range': range, 'arange': np.arange}[func_name]
```

(continues on next page)

(continued from previous page)

```
    for i in f(n):
        pass
```

Benchmarks may also be conditionally skipped based on a boolean with `@skip_benchmark_if`:

```
from asv_runner.benchmarks.mark import skip_benchmark_if
import datetime

# Skip if not before midday
@skip_benchmark_if(
    datetime.datetime.now(datetime.timezone.utc).hour >= 12
)
def time_ranges(n, func_name):
    f = {'range': range, 'arange': np.arange}[func_name]
    for i in f(n):
        pass
```

Similarly, for parameters we have `@skip_params_if`:

```
from asv_runner.benchmarks.mark import skip_params_if
import datetime

class TimeSuite:
    params = [100, 200, 300, 400, 500]
    param_names = ["size"]

    def setup(self, size):
        self.d = {}
        for x in range(size):
            self.d[x] = None

    # Skip benchmarking when size is either 100 or 200
    # and the current hour is 12 or later.
    @skip_params_if(
        [(100,), (200,)],
        datetime.datetime.now(datetime.timezone.utc).hour >= 12
    )
    def time_dict_update(self, size):
        d = self.d
        for i in range(size):
            d[i] = i
```

Warning: The skips discussed so far, using the decorators will ignore both the benchmark, and the `setup` function, however, `setup_cache` will not be affected.

If the onus of preparing the exact parameter sets for `skip_for_params` is too complicated and the `setup` function is not too expensive, or if a benchmark needs to be skipped conditionally but `skip_*_if` are not the right choice, there is also the `SkipNotImplemented` exception which can be raised anywhere during a benchmark run for it to be marked as skipped (`n/a` in the output table). This may be used as:

```
from asv_runner.benchmarks.mark import SkipNotImplemented
class SimpleSlow:
    params = ([False, True])
    param_names = ["ok"]
    def time_failure(self, ok):
        if ok:
            x = 34.2**4.2
        else:
            raise SkipNotImplemented(f"{ok} is skipped")
```

3.6 Benchmark types

3.6.1 Timing

Timing benchmarks have the prefix `time`.

How ASV runs benchmarks is as follows (pseudocode for main idea):

```
for round in range(`rounds`):
   for benchmark in benchmarks:
       with new process:
           <calibrate `number` if not manually set>
           for j in range(`repeat`):
               <setup `benchmark`>
               sample = timing_function(<run benchmark `number` times>) / `number`
               <teardown `benchmark`>
```

where the actual `rounds`, `repeat`, and `number` are *attributes of the benchmark*.

The default timing function is `timeit.default_timer`, which uses the highest resolution clock available on a given platform to measure the elapsed wall time. This has the consequence of being more susceptible to noise from other processes, but the increase in resolution is more significant for shorter duration tests (particularly on Windows).

Process timing is provided by the function `time.process_time` (POSIX `CLOCK_PROCESS_CPUTIME`), which measures the CPU time used only by the current process. You can change the timer by setting the benchmark's `timer` attribute, for example to `time.process_time` to measure process time.

Note: One consequence of using `time.process_time` is that the time spent in child processes of the benchmark is not included. Multithreaded benchmarks also return the total CPU time counting all CPUs. In these cases you may want to measure the wall clock time, by setting the `timer = timeit.default_timer` benchmark attribute.

For best results, the benchmark function should contain as little as possible, with as much extraneous setup moved to a `setup` function:

```
class Suite:
    def setup(self):
        # load data from a file
        with open("/usr/share/words.txt", "r") as fd:
            self.words = fd.readlines()

    def time_upper(self):
```

(continues on next page)

(continued from previous page)

```
        for word in self.words:
            word.upper()
```

How `setup` and `teardown` behave for timing benchmarks is similar to the Python `timeit` module, and the behavior is controlled by the `number` and `repeat` attributes.

For the list of benchmark attributes, see *Benchmark types and attributes*.

3.6.2 Memory

Memory benchmarks have the prefix `mem`.

Memory benchmarks track the size of Python objects. To write a memory benchmark, write a function that returns the object you want to track:

```
def mem_list():
    return [0] * 256
```

The asizeof module is used to determine the size of Python objects. Since `asizeof` includes the memory of all of an object's dependencies (including the modules in which their classes are defined), a memory benchmark instead calculates the incremental memory of a copy of the object, which in most cases is probably a more useful indicator of how much space *each additional* object will use. If you need to do something more specific, a generic *Tracking (Generic)* benchmark can be used instead.

For details, see *Benchmark types and attributes*.

Note: The memory benchmarking feature is still experimental. `asizeof` may not be the most appropriate metric to use.

Note: The memory benchmarks are not supported on PyPy.

3.6.3 Peak Memory

Peak memory benchmarks have the prefix `peakmem`.

Peak memory benchmark tracks the maximum resident size (in bytes) of the process in memory. This does not necessarily count memory paged on-disk, or that used by memory-mapped files. To write a peak memory benchmark, write a function that does the operation whose maximum memory usage you want to track:

```
def peakmem_list():
    [0] * 165536
```

Note: The peak memory benchmark also counts memory usage during the `setup` routine, which may confound the benchmark results. One way to avoid this is to use `setup_cache` instead.

For details, see *Benchmark types and attributes*.

3.6.4 Raw timing benchmarks

For some timing benchmarks, for example measuring the time it takes to import a module, it is important that they are run separately in a new Python process.

Measuring execution time for benchmarks run once in a new Python process can be done with `timeraw_*` timing benchmarks:

```
def timeraw_import_inspect():
    return """
    import inspect
    """
```

Note that these benchmark functions should return a string, corresponding to the code that will be run.

Importing a module takes a meaningful amount of time only the first time it is executed, therefore a fresh interpreter is used for each iteration of the benchmark. The string returned by the benchmark function is executed in a subprocess.

Note that the setup and setup_cache are performed in the base benchmark process, so that the setup done by them is not available in the benchmark code. To perform setup also in the benchmark itself, you can return a second string:

def timeraw_import_inspect():
code = "import inspect" setup = "import ast" return code, setup

The raw timing benchmarks have the same parameters as ordinary timing benchmarks, but `number` is by default 1, and `timer` is ignored.

Note: Timing standard library modules is possible as long as they are not built-in or brought in by importing the `timeit` module (which further imports `gc`, `sys`, `time`, and `itertools`).

3.6.5 Imports

You can use raw timing benchmarks to measure import times.

3.6.6 Tracking (Generic)

It is also possible to use `asv` to track any arbitrary numerical value. "Tracking" benchmarks can be used for this purpose and use the prefix `track`. These functions simply need to return a numeric value. For example, to track the number of objects known to the garbage collector at a given state:

```
import gc

def track_num_objects():
    return len(gc.get_objects())
track_num_objects.unit = "objects"
```

For details, see *Benchmark types and attributes*.

3.7 Benchmark versioning

When you edit benchmark's code in the benchmark suite, this often changes what is measured, and previously measured results should be discarded.

Airspeed Velocity records with each benchmark measurement a "version number" for the benchmark. By default, it is computed by hashing the benchmark source code text, including any `setup` and `setup_cache` routines. If there are changes in the source code of the benchmark in the benchmark suite, the version number changes, and `asv` will ignore results whose version number is different from the current one.

It is also possible to control the versioning of benchmark results manually, by setting the `.version` attribute for the benchmark. The version number, i.e. content of the attribute, can be any Python string. `asv` only checks whether the version recorded with a measurement matches the current version, so you can use any versioning scheme.

See *Benchmark types and attributes* for reference documentation.

CHAPTER

FOUR

TUNING TIMING MEASUREMENTS

The results from timing benchmarks are generally variable.

Performance variations occur on different time scales. For timing benchmarks repeated immediately after each other, there is always some jitter in the results, due to operating system scheduling and other sources. For timing benchmarks run at more widely separated times, systematic differences changing on long time scales can appear, for example from changes in the background system load or built-in CPU mechanisms for power and heat management.

Airspeed Velocity has mechanisms to deal with these variations. For dealing with short-time variations, you can use the `sample_time`, `number` and `repeat` attributes of timing benchmarks to control how results are sampled and averaged. For long-time variations, you can use the `rounds` attribute and `--interleave-rounds`, `--append-samples`, and `-a rounds=4` command line options to run timing benchmarks at more widely spaced times, in order to average over long-time performance variations.

If you are planning to capture historical benchmark data for most commits, very accurate timings are not necessary. The detection of regressions in historical benchmark data used in `asv` is designed to be statistically robust and tolerates fair amounts of noise. However, if you are planning to use `asv continuous` and `asv compare`, accurate results are more important.

4.1 Library settings

If your code uses 3rd party libraries, you may want to check their settings before benchmarking. In particular, such libraries may use automatic multithreading, which may affect runtime performance in surprising ways. If you are using libraries such as OpenBLAS, Intel MKL, or OpenMP, benchmark results may become easier to understand by forcing single-threaded operation. For these three, this can be typically done by setting environment variables:

```
OPENBLAS_NUM_THREADS=1
MKL_NUM_THREADS=1
OMP_NUM_THREADS=1
```

4.2 Tuning machines for benchmarking

Especially if you are using a laptop computer for which the heat and power management is an issue, getting reliable results may require too long averaging times to be practical.

To improve the situation it is possible to optimize the usage and settings of your machine to minimize the variability in timing benchmarks. Generally, while running benchmarks there should not be other applications actively using CPU, or you can run `asv` pinned to a CPU core not used by other processes. You should also force the CPU frequency or power level settings to a fixed value.

The pyperf project has documentation on how to tune machines for benchmarking. The simplest way to apply basic tuning on Linux using `pyperf` is to run:

```
sudo python -mpyperf system tune
```

This will modify system settings that can be only changed as root, and you should read the `pyperf` documentation on what it precisely does. This system tuning also improves results for `asv`. To achieve CPU affinity pinning with `asv` (e.g. to an isolated CPU), you should use *the –cpu-affinity option*.

It is also useful to note that configuration changes and operating system upgrades on the benchmarking machine can change the baseline performance of the machine. For absolutely best results, you may then want to use a dedicated benchmarking machine that is not used for anything else. You may also want to carefully select a long-term supported operating system, such that you can only choose to install security upgrades.

CHAPTER

FIVE

REFERENCE

5.1 Benchmark types and attributes

Contents

Warning: Changed in version 0.6.0: The code for these have now been moved to be in `asv_runner`, and the rest of the documentation may be outdated.

5.1.1 Benchmark types

The following benchmark types are recognized:

- `def time_*()`: measure time taken by the function. See *Timing*.
- `def timeraw_*()`: measure time taken by the function, after interpreter start. See *Raw timing benchmarks*.
- `def mem_*()`: measure memory size of the object returned. See *Memory*.
- `def peakmem_*()`: measure peak memory size of the process when calling the function. See *Peak Memory*.
- `def track_*()`: use the returned numerical value as the benchmark result See *Tracking (Generic)*.

Note: New in version 0.6.2: External benchmarks may be defined through `asv_runner` and a list of benchmark plugins (like `asv_bench_memray`) may be found here, with samples at asv_samples.

5.1.2 Benchmark attributes

Benchmark attributes can either be applied directly to the benchmark function:

```
def time_something():
    pass

time_something.timeout = 123
```

or appear as class attributes:

```
class SomeBenchmarks:
    timeout = 123

    def time_something(self):
        pass
```

Different benchmark types have their own sets of applicable attributes. Moreover, the following attributes are applicable to all benchmark types:

- `timeout`: The amount of time, in seconds, to give the benchmark to run before forcibly killing it. Defaults to 60 seconds.
- `benchmark_name`: If given, used as benchmark function name instead of generated one `<module>.<class>.<function>`.
- `pretty_name`: If given, used to display the benchmark name instead of the benchmark function name.
- `pretty_source`: If given, used to display a custom version of the benchmark source.
- `version`: Used to determine when to invalidate old benchmark results. Benchmark results produced with a different value of the version than the current value will be ignored. The value can be any Python string (or other object, `str()` will be taken).

  Default (if `version=None` or not given): hash of the source code of the benchmark function and setup and setup_cache methods. If the source code of any of these changes, old results become invalidated.
- `setup`: function to be called as a setup function for the benchmark See *Setup and teardown functions* for discussion.
- `teardown`: function to be called as a teardown function for the benchmark See *Setup and teardown functions* for discussion.
- `setup_cache`: function to be called as a cache setup function. See *Setup and teardown functions* for discussion.
- `param_names`: list of parameter names See *Parameterized benchmarks* for discussion.
- `params`: list of lists of parameter values. If there is only a single parameter, may also be a list of parameter values. See *Parameterized benchmarks* for discussion.

  Example:

```
def setup_func(n, func):
    print(n, func)

def teardown_func(n, func):
    print(n, func)

def time_ranges(n, func):
    for i in func(n):
```

(continues on next page)

(continued from previous page)

```
        pass

time_ranges.setup = setup_func
time_ranges.param_names = ['n', 'func']
time_ranges.params = ([10, 1000], [range, numpy.arange])
```

The benchmark will be run for parameters `(10, range)`, `(10, numpy.arange)`, `(1000, range)`, `(1000, numpy.arange)`. The setup and teardown functions will also obtain these parameters.

Note that `setup_cache` is not parameterized.

For the purposes of identifying benchmarks in the UI, `repr()` is called on the elements of `params`. In the event these strings contain memory addresses, those adresses are stripped to allow comparison across runs. Additionally, if this results in a non-unique mapping, each duplicated element will be suffixed with a distinct integer identifier corresponding to order of appearance.

Timing benchmarks

- `warmup_time`: `asv` will spend this time (in seconds) in calling the benchmarked function repeatedly, before starting to run the actual benchmark. If not specified, `warmup_time` defaults to 0.1 seconds (on PyPy, the default is 1.0 sec).
- `rounds`: How many rounds to run the benchmark in (default: 2). The rounds run different timing benchmarks in an interleaved order, allowing to sample over longer periods of background performance variations (e.g. CPU power levels).
- `repeat`: The number measurement samples to collect per round. Each sample consists of running the benchmark `number` times. The median time from all samples collected in all roudns is used as the final measurement result.

  `repeat` can be a tuple `(min_repeat, max_repeat, max_time)`. In this case, the measurement first collects at least `min_repeat` samples, and continues until either `max_repeat` samples are collected or the collection time exceeds `max_time`.

  When not provided (`repeat` set to 0), the default value is `(1, 10, 20.0)` if `rounds`==1 and `(1, 5, 10.0)` otherwise.
- `number`: Manually choose the number of iterations in each sample. If `number` is specified, `sample_time` is ignored. Note that `setup` and `teardown` are not run between iterations: `setup` runs first, then the timed benchmark routine is called `number` times, and after that `teardown` runs.
- `sample_time`: `asv` will automatically select `number` so that each sample takes approximatively `sample_time` seconds. If not specified, `sample_time` defaults to 10 milliseconds.
- `min_run_count`: the function is run at least this many times during benchmark. Default: 2
- `timer`: The timing function to use, which can be any source of monotonically increasing numbers, such as `time.clock`, `time.time` or `time.process_time`. If it's not provided, it defaults to `timeit.default_timer`, but other useful values are `process_time`, for which `asv` provides a backported version for versions of Python prior to 3.3.

  Changed in version 0.4: Previously, the default timer measured process time, which was chosen to minimize noise from other processes. However, on Windows, this is only available at a resolution of 15.6ms, which is greater than the recommended benchmark runtime of 10ms. Therefore, we default to the highest resolution clock on any platform.

The `sample_time`, `number`, `repeat`, and `timer` attributes can be adjusted in the `setup()` routine, which can be useful for parameterized benchmarks.

Tracking benchmarks

- `unit`: The unit of the values returned by the benchmark. Used for display in the web interface.

5.1.3 Environment variables

When `asv` runs benchmarks, several environment variables are defined, see *ASV environment variables*.

5.2 `asv.conf.json` reference

The `asv.conf.json` file contains information about a particular benchmarking project. The following describes each of the keys in this file and their expected values.

Note: The GitHub repository at asv_samples serves as a comprehensive showcase for integrating Air Speed Velocity (ASV) with a wide array of Python project configurations. This includes various build systems and advanced benchmarking features like custom parameterizations and ASV plugins, aiming to benchmark Python code performance across diverse setups.

The repository is structured with dedicated branches for each build system and feature demonstration, providing insights into the impacts of different build systems and ASV's extensible features on performance metrics.

Contents

- *html_dir*
- *hash_length*
- *plugins*
- *build_cache_size*
- *regressions_first_commits*
- *regressions_thresholds*

5.2.1 project

The name of the project being benchmarked.

5.2.2 project_url

The URL to the homepage of the project. This can point to anywhere, really, as it's only used for the link at the top of the benchmark results page back to your project.

5.2.3 repo

The URL to the repository for the project.

The value can also be a path, relative to the location of the configuration file. For example, if the benchmarks are stored in the same repository as the project itself, and the configuration file is located at `benchmarks/asv.conf.json` inside the repository, you can set `"repo": ".."` to use the local repository.

Currently, only `git` and `hg` repositories are supported, so this must be a URL that `git` or `hg` know how to clone from, for example:

- git@github.com:airspeed-velocity/asv.git
- https://github.com/airspeed-velocity/asv.git
- ssh://hg@bitbucket.org/yt_analysis/yt
- hg+https://bitbucket.org/yt_analysis/yt

The repository may be readonly.

5.2.4 repo_subdir

The relative path to your Python project inside the repository. This is where its `setup.py` file is located.

If empty or omitted, the project is assumed to be located at the root of the repository.

5.2.5 build_command, install_command, uninstall_command

Airspeed Velocity rebuilds the project as needed, using these commands.

The defaults are:

```
"install_command":
["in-dir={env_dir} python -mpip install {wheel_file}"],

"uninstall_command":
["return-code=any python -mpip uninstall -y {project}"],

"build_command":
["python setup.py build",
 "PIP_NO_BUILD_ISOLATION=false python -mpip wheel --no-deps --no-index -w {build_cache_
↪dir} {build_dir}"],
```

Note: Changed in version 0.6.2: The default build command now assume network connectivity is not prohibited. The `build_command` is now:

```
"build_command":
["python setup.py build",
 "python -mpip wheel -w {build_cache_dir} {build_dir}"],
```

The install commands should install the project in the active Python environment (virtualenv/conda), so that it can be used by the benchmark code.

The uninstall commands should uninstall the project from the environment.

Note: Changed in version 0.6.0: If a build command is not specified in the `asv.conf.json`, the default assumes the build system requirements are defined in a `setup.py` file. `pyproject.toml` is the preferred file format to define the build system requirements of Python projects (PEP518), and this approach will be the default from `asv v0.6.0` onwards.

The build commands can optionally be used to cache build results in the cache directory `{build_cache_dir}`, which is commit and environment-specific. If the cache directory contains any files after `build_command` finishes with exit code 0, `asv` assumes it contains a cached build. When a cached build is available, `asv` will only call `install_command` but not `build_command`. (The number of cached builds retained at any time is determined by the `build_cache_size` configuration option.)

The `install_command` and `build_command` are by default launched in `{build_dir}`. The `uninstall_command` is launched in the environment root directory.

The commands are specified in typical POSIX shell syntax (Python shlex), but are not run in a shell, so that e.g. `cd` has no effect on subsequent commands, and wildcard or environment variable expansion is not done. The substituted variables `{variable_name}` do not need to be quoted. The commands may contain environment variable specifications in in form `VARNAME=value` at the beginning. In addition, valid return codes can be specified via `return-code=0,1,2` and `return-code=any`.

The `in-dir=somedir` specification changes the working directory for the command.

The commands can be supplied with the arguments:

- `{project}`: the project name from the configuration file

- {env_name}: name of the currently active environment
- {env_type}: type of the currently active environment
- {env_dir}: full path to the currently active environment root
- {conf_dir}: full path to the directory where asv.conf.json is
- {build_dir}: full path to the build directory (checked-out source path + repo_subdir)
- {build_cache_dir}: full path to the build cache directory
- {commit}: commit hash of currently installed project
- {wheel_file}: absolute path to a *.whl file in {build_cache_dir} (defined only if there is exactly one existing wheel file in the directory).

Several *environment variables* are also defined.

5.2.6 branches

Branches to generate benchmark results for.

This controls how the benchmark results are displayed, and what benchmarks asv run ALL and asv run NEW run.

If not provided, "main" (Git) or "default" (Mercurial) is chosen.

5.2.7 show_commit_url

The base URL to show information about a particular commit. The commit hash will be added to the end of this URL and then opened in a new tab when a data point is clicked on in the web interface.

For example, if using Github to host your repository, the show_commit_url should be:

http://github.com/owner/project/commit/

5.2.8 pythons

The versions of Python to run the benchmarks in. If not provided, it will to default to the version of Python that the asv command (main) is being run under.

If provided, it should be a list of strings. It may be one of the following:

- a Python version string, e.g. "3.7", in which case:
  - if conda is found, conda will be used to create an environment for that version of Python via a temporary environment.yml file
  - if virtualenv is installed, asv will search for that version of Python on the PATH and create a new virtual environment based on it. asv does not handle downloading and installing different versions of Python for you. They must already be installed and on the path. Depending on your platform, you can install multiple versions of Python using your package manager or using pyenv.
- an executable name on the PATH or an absolute path to an executable. In this case, the environment is assumed to be already fully loaded and read-only. Thus, the benchmarked project must already be installed, and it will not be possible to benchmark multiple revisions of the project.

5.2.9 conda_environment_file

A path to a `conda` environment file to use as source for the dependencies. For example:

```
"conda_environment_file": "environment.yml"
```

The environment file should generally install `wheel` and `pip`, since those are required by the default `asv` build commands. If there are packages present in `matrix`, an additional `conda env update` call is used to install them after the environment is created.

Note: Changed in version 0.6.0: If an `environment.yml` file is present where `asv` is run, it will be used. To turn off this behavior, `conda_environment_file` can be set to `IGNORE`.

This option will cause `asv` to ignore the Python version in the environment creation, which is then assumed to be fixed by the environment file.

5.2.10 conda_channels

A list of `conda` channel names (strings) to use in the provided order as the source channels for the dependencies. For example:

```
"conda_channels": ["conda-forge", "defaults"]
```

The channels will be parsed by `asv` to populate the `channels` section of a temporary environment.yml file used to build the benchmarking environment.

5.2.11 matrix

Defines a matrix of third-party dependencies and environment variables to run the benchmarks with.

If provided, it must be a dictionary, containing some of the keys "req", "env", "env_nobuild". For example:

```
"matrix": {
    "req": {
        "numpy": ["1.25", "1.26"],
        "Cython": []
        "six": ["", null]
    },
    "env": {
        "FOO": "bar"
    }
}
```

The keys of the `"req"` are the names of dependencies, and the values are lists of versions (as strings) of that dependency. An empty string means the "latest" version of that dependency available on PyPI. Value of `null` means the package will not be installed.

If the list is empty, it is equivalent to `[""]`, in other words, the "latest" version.

For example, the following will test with two different versions of Numpy, the latest version of Cython, and six installed as the latest version and not installed at all:

```
"matrix": {
    "req": {
        "numpy": ["1.25", "1.26"],
        "Cython": []
        "six": ["", null],
    }
}
```

The matrix dependencies are installed *before* any dependencies that the project being benchmarked may specify in its `setup.py` file.

Note: At present, this functionality only supports dependencies that are installable via `pip` or `conda` or `mamba` (depending on which environment is used). If `conda/mamba` is specified as `environment_type` and you wish to install the package via `pip`, then preface the package name with `pip+`. For example, `emcee` is only available from `pip`, so the package name to be used is `pip+emcee`.

The `env` and `env_nobuild` dictionaries can be used to set also environment variables:

```
"matrix": {
    "env": {
        "ENV_VAR_1": ["val1", "val2"],
        "ENV_VAR_2": ["val3", null],
    },
    "env_nobuild": {
        "ENV_VAR_3": ["val4", "val5"],
    }
}
```

Variables in "no_build" will be passed to every environment during the test phase, but will not trigger a new build. A value of `null` means that the variable will not be set for the current combination.

The above matrix will result in 4 different builds with the following additional environment variables and values:

- [("ENV_VAR_1", "val1"), ("ENV_VAR_2", "val3")]
- [("ENV_VAR_1", "val1")]
- [("ENV_VAR_1", "val2"), ("ENV_VAR_2", "val3")]
- [("ENV_VAR_1", "val2")]

It will generate 8 different test environments based on those 4 builds with the following environment variables and values:

- [("ENV_VAR_1", "val1"), ("ENV_VAR_2", "val3"), ("ENV_VAR_3", "val4")]
- [("ENV_VAR_1", "val1"), ("ENV_VAR_2", "val3"), ("ENV_VAR_3", "val5")]
- [("ENV_VAR_1", "val1"), ("ENV_VAR_3", "val4")]
- [("ENV_VAR_1", "val1"), ("ENV_VAR_3", "val5")]
- [("ENV_VAR_1", "val2"), ("ENV_VAR_2", "val3"), ("ENV_VAR_3", "val4")]
- [("ENV_VAR_1", "val2"), ("ENV_VAR_2", "val3"), ("ENV_VAR_3", "val5")]
- [("ENV_VAR_1", "val2"), ("ENV_VAR_3", "val4")]
- [("ENV_VAR_1", "val2"), ("ENV_VAR_3", "val5")]

5.2.12 exclude

Combinations of libraries, Python versions, or platforms to be excluded from the combination matrix. If provided, must be a list of dictionaries, each specifying an exclude rule.

An exclude rule consists of key-value pairs, specifying matching rules `matrix[key] ~ value`. The values are strings containing regular expressions that should match whole strings. The exclude rule matches if all of the items in it match.

Each exclude rule can contain the following keys:

- `python`: Python version (from `pythons`)
- `sys_platform`: Current platform, as in `sys.platform`. Common values are: `linux2`, `win32`, `cygwin`, `darwin`.
- `environment_type`: The environment type in use (from `environment_type`).
- `req`: dictionary of rules vs. the requirements
- `env`: dictionary of rules vs. environment variables
- `env_nobuild`: : dictionary of rules vs. the non-build environment variables

For example:

```
"pythons": ["3.8", "3.9"],
"matrix": {
    "req": {
        "numpy": ["1.25", "1.26"],
        "Cython": ["", null],
        "colorama": ["", null]
    },
    "env": {"FOO": ["1", "2"]},
},
"exclude": [
    {"python": "3.8", "req": {"numpy": "1.25"}},
    {"sys_platform": "(?!win32).*", "req": {"colorama": ""}},
    {"sys_platform": "win32", "req": {"colorama": null}},
    {"env": {"FOO": "1"}},
]
```

This will generate all combinations of Python version and items in the matrix, except those with Python 3.8 and Numpy 3.9. In other words, the combinations:

```
python==3.8 numpy==1.26 Cython==latest (colorama==latest) FOO=2
python==3.8 numpy==1.26 (colorama==latest) FOO=2
python==3.9 numpy==1.25 Cython==latest (colorama==latest) FOO=2
python==3.9 numpy==1.25 (colorama==latest) FOO=2
python==3.9 numpy==1.26 Cython==latest (colorama==latest) FOO=2
python==3.9 numpy==1.26 (colorama==latest) FOO=2
```

The `colorama` package will be installed only if the current platform is Windows.

5.2.13 include

Additional package combinations to be included as environments.

If specified, must be a list of dictionaries, indicating the versions of packages and other environment configuration to be installed. The dictionary must also include a `python` key specifying the Python version.

Similarly as for the matrix, the `"req"`, `"env"` and `"env_nobuild"` entries specify dictionaries containing requirements and environment variables. In contrast to the matrix, the values are not lists, but a single value only.

In addition, the following keys can be present: `sys_platform`, `environment_type`. If present, the include rule is active only if the values match, using same matching rules as explained for `exclude` above.

The exclude rules are not applied to includes.

For example:

```
"include": [
    {"python": "3.9", "req": {"numpy": "1.26"}, "env": {"FOO": "true"}},
    {"platform": "win32", "environment_type": "conda",
     "req": {"python": "3.12", "libpython": ""}}
]
```

This corresponds to two additional environments. One runs on Python 3.9 and including the specified version of Numpy. The second is active only for Conda on Windows, and installs the latest version of `libpython`.

5.2.14 benchmark_dir

The directory, relative to the current directory, that benchmarks are stored in. Should rarely need to be overridden. If not provided, defaults to `"benchmarks"`.

5.2.15 environment_type

Specifies the tool to use to create environments. May be “conda”, “virtualenv”, “mamba” or another value depending on the plugins in use. If missing or the empty string, the tool will be automatically determined by looking for tools on the `PATH` environment variable.

5.2.16 env_dir

The directory, relative to the current directory, to cache the Python environments in. If not provided, defaults to `"env"`.

5.2.17 results_dir

The directory, relative to the current directory, that the raw results are stored in. If not provided, defaults to `"results"`.

5.2.18 html_dir

The directory, relative to the current directory, to save the website content in. If not provided, defaults to `"html"`.

5.2.19 hash_length

The number of characters to retain in the commit hashes when displayed in the web interface. The default value of 8 should be more than enough for most projects, but projects with extremely large history may need to increase this value. This does not affect the storage of results, where the full commit hash is always retained.

5.2.20 plugins

A list of modules to import containing asv plugins.

5.2.21 build_cache_size

The number of builds to cache for each environment.

5.2.22 regressions_first_commits

The commits after which the regression search in *asv publish* should start looking for regressions.

The value is a dictionary mapping benchmark identifier regexps to commits after which to look for regressions. The benchmark identifiers are of the form `benchmark_name(parameters)@branch`, where `(parameters)` is present only for parameterized benchmarks. If the commit identifier is *null*, regression detection for the matching benchmark is skipped. The default is to start from the first commit with results.

Example:

```
"regressions_first_commits": {
    ".*": "v0.1.0",
    "benchmark_1": "80fca08d",
    "benchmark_2@main": null,
}
```

In this case, regressions are detected only for commits after tag `v0.1.0` for all benchmarks. For `benchmark_1`, regression detection is further limited to commits after the commit given, and for `benchmark_2`, regression detection is skipped completely in the `main` branch.

5.2.23 regressions_thresholds

The minimum relative change required before *asv publish* reports a regression.

The value is a dictionary, similar to `regressions_first_commits`. If multiple entries match, the largest threshold is taken. If no entry matches, the default threshold is `0.05` (iow. 5%).

Example:

```
"regressions_thresholds": {
    ".*": 0.01,
    "benchmark_1": 0.2,
}
```

In this case, the reporting threshold is 1% for all benchmarks, except `benchmark_1` which uses a threshold of 20%.

5.3 Commands

Contents

5.3.1 asv help

```
usage: asv help [-h]

options:
  -h, --help  show this help message and exit
```

5.3.2 asv quickstart

```
usage: asv quickstart [-h] [--dest DEST] [--top-level | --no-top-level]
                      [--verbose] [--config CONFIG] [--version]

Creates a new benchmarking suite

options:
  -h, --help            show this help message and exit
  --dest DEST, -d DEST  The destination directory for the new benchmarking
                        suite
  --top-level           Use layout suitable for putting the benchmark suite on
                        the top level of the project's repository
  --no-top-level        Use layout suitable for putting the benchmark suite in
                        a separate repository
  --verbose, -v         Increase verbosity
  --config CONFIG       Benchmark configuration file
  --version             Print program version
```

5.3.3 asv machine

```
usage: asv machine [-h] [--machine MACHINE] [--os OS] [--arch ARCH]
                   [--cpu CPU] [--num_cpu NUM_CPU] [--ram RAM] [--yes]
                   [--verbose] [--config CONFIG] [--version]

Defines information about this machine. If no arguments are provided, an
interactive console session will be used to ask questions about the machine.

options:
  -h, --help          show this help message and exit
  --machine MACHINE   A unique name to identify this machine in the results.
                      May be anything, as long as it is unique across all the
                      machines used to benchmark this project. NOTE: If changed
                      from the default, it will no longer match the hostname of
                      this machine, and you may need to explicitly use the
                      --machine argument to asv.
  --os OS             The OS type and version of this machine. For example,
                      'Macintosh OS-X 10.8'.
  --arch ARCH         The generic CPU architecture of this machine. For
                      example, 'i386' or 'x86_64'.
  --cpu CPU           A specific description of the CPU of this machine,
                      including its speed and class. For example, 'Intel(R)
                      Core(TM) i5-2520M CPU @ 2.50GHz (4 cores)'.
  --num_cpu NUM_CPU   The number of CPUs in the system. For example, '4'.
  --ram RAM           The amount of physical RAM on this machine. For example,
                      '4GB'.
  --yes               Accept all questions
  --verbose, -v       Increase verbosity
  --config CONFIG     Benchmark configuration file
  --version           Print program version
```

5.3.4 asv setup

```
usage: asv setup [-h] [--parallel [PARALLEL]] [-E ENV_SPEC] [--python PYTHON]
                 [--verbose] [--config CONFIG] [--version]

Setup virtual environments for each combination of Python version and third-
party requirement. This is called by the ``run`` command implicitly, and isn't
generally required to be run on its own.

options:
  -h, --help            show this help message and exit
  --parallel [PARALLEL], -j [PARALLEL]
                        Build (but don't benchmark) in parallel. The value is
                        the number of CPUs to use, or if no number provided,
                        use the number of cores on this machine.
  -E ENV_SPEC, --environment ENV_SPEC
                        Specify the environment and Python versions for
                        running the benchmarks. String of the format
                        'environment_type:python_version', for example
                        'conda:3.12'. If the Python version is not specified,
                        all those listed in the configuration file are run.
                        The special environment type
                        'existing:/path/to/python' runs the benchmarks using
                        the given Python interpreter; if the path is omitted,
                        the Python running asv is used. For 'existing', the
                        benchmarked project must be already installed,
                        including all dependencies. By default, uses the
                        values specified in the configuration file.
  --python PYTHON       Same as --environment=:PYTHON
  --verbose, -v         Increase verbosity
  --config CONFIG       Benchmark configuration file
  --version             Print program version
```

5.3.5 asv run

```
usage: asv run [-h] [--date-period DATE_PERIOD] [--steps STEPS]
               [--bench BENCH] [--attribute ATTRIBUTE]
               [--cpu-affinity ATTRIBUTE] [--profile] [--parallel [PARALLEL]]
               [--show-stderr] [--durations [N]] [--quick] [-E ENV_SPEC]
               [--python PYTHON] [--set-commit-hash SET_COMMIT_HASH]
               [--launch-method {auto,spawn,forkserver}] [--dry-run]
               [--machine MACHINE] [--skip-existing-successful]
               [--skip-existing-failed] [--skip-existing-commits]
               [--skip-existing] [--record-samples] [--append-samples]
               [--interleave-rounds] [--no-interleave-rounds] [--no-pull]
               [--verbose] [--config CONFIG] [--version]
               [range]

Run a benchmark suite.

examples:
```

(continues on next page)

(continued from previous page)

```
  asv run main             run for one branch
  asv run main^!           run for one commit (git)
  asv run "--merges main"  run for only merge commits (git)

positional arguments:
  range                 Range of commits to benchmark. For a git repository,
                        this is passed as the first argument to ``git rev-
                        list``; or Mercurial log command. See 'specifying
                        ranges' section of the `gitrevisions` manpage, or 'hg
                        help revisions', for more info. Also accepts the
                        special values 'NEW', 'ALL', 'EXISTING', and
                        'HASHFILE:xxx'. 'NEW' will benchmark all commits since
                        the latest benchmarked on this machine. 'ALL' will
                        benchmark all commits in the project. 'EXISTING' will
                        benchmark against all commits for which there are
                        existing benchmarks on any machine. 'HASHFILE:xxx'
                        will benchmark only a specific set of hashes given in
                        the file named 'xxx' ('-' means stdin), which must
                        have one hash per line. By default, will benchmark the
                        head of each configured of the branches.

options:
  -h, --help            show this help message and exit
  --date-period DATE_PERIOD
                        Pick only one commit in each given time period. For
                        example: 1d (daily), 1w (weekly), 1y (yearly).
  --steps STEPS, -s STEPS
                        Maximum number of steps to benchmark. This is used to
                        subsample the commits determined by range to a
                        reasonable number.
  --bench BENCH, -b BENCH
                        Regular expression(s) for benchmark to run. When not
                        provided, all benchmarks are run.
  --attribute ATTRIBUTE, -a ATTRIBUTE
                        Override a benchmark attribute, e.g. `-a repeat=10`.
  --cpu-affinity ATTRIBUTE
                        Set CPU affinity for running the benchmark, in format:
                        0 or 0,1,2 or 0-3. Default: not set
  --profile, -p         In addition to timing, run the benchmarks through the
                        `cProfile` profiler and store the results.
  --parallel [PARALLEL], -j [PARALLEL]
                        Build (but don't benchmark) in parallel. The value is
                        the number of CPUs to use, or if no number provided,
                        use the number of cores on this machine.
  --show-stderr, -e     Display the stderr output from the benchmarks.
  --durations [N]       Display total duration for N (or 'all') slowest
                        benchmarks
  --quick, -q           Do a "quick" run, where each benchmark function is run
                        only once. This is useful to find basic errors in the
                        benchmark functions faster. The results are unlikely
                        to be useful, and thus are not saved.
  -E ENV_SPEC, --environment ENV_SPEC
```

(continues on next page)

(continued from previous page)

```
                      Specify the environment and Python versions for
                      running the benchmarks. String of the format
                      'environment_type:python_version', for example
                      'conda:3.12'. If the Python version is not specified,
                      all those listed in the configuration file are run.
                      The special environment type
                      'existing:/path/to/python' runs the benchmarks using
                      the given Python interpreter; if the path is omitted,
                      the Python running asv is used. For 'existing', the
                      benchmarked project must be already installed,
                      including all dependencies. By default, uses the
                      values specified in the configuration file.
--python PYTHON       Same as --environment=:PYTHON
--set-commit-hash SET_COMMIT_HASH
                      Set the commit hash to use when recording benchmark
                      results. This makes results to be saved also when
                      using an existing environment.
--launch-method {auto,spawn,forkserver}
                      How to launch benchmarks. Choices: auto, spawn,
                      forkserver
--dry-run, -n         Do not save any results to disk.
--machine MACHINE, -m MACHINE
                      Use the given name to retrieve machine information. If
                      not provided, the hostname is used. If no entry with
                      that name is found, and there is only one entry in
                      ~/.asv-machine.json, that one entry will be used.
--skip-existing-successful
                      Skip running benchmarks that have previous successful
                      results
--skip-existing-failed
                      Skip running benchmarks that have previous failed
                      results
--skip-existing-commits
                      Skip running benchmarks for commits that have existing
                      results
--skip-existing, -k   Skip running benchmarks that have previous successful
                      or failed results
--record-samples      Store raw measurement samples, not only statistics
--append-samples      Combine new measurement samples with previous results,
                      instead of discarding old results. Implies --record-
                      samples. The previous run must also have been run with
                      --record/append-samples.
--interleave-rounds   Interleave benchmarks with multiple rounds across
                      commits. This can avoid measurement biases from commit
                      ordering, can take longer.
--no-interleave-rounds
--no-pull             Do not pull the repository
--verbose, -v         Increase verbosity
--config CONFIG       Benchmark configuration file
--version             Print program version
```

5.3.6 asv dev

```
usage: asv dev [-h] [--date-period DATE_PERIOD] [--steps STEPS]
               [--bench BENCH] [--attribute ATTRIBUTE]
               [--cpu-affinity ATTRIBUTE] [--profile] [--parallel [PARALLEL]]
               [--show-stderr] [--durations [N]] [--quick] [-E ENV_SPEC]
               [--python PYTHON] [--set-commit-hash SET_COMMIT_HASH]
               [--launch-method {auto,spawn,forkserver}] [--dry-run]
               [--machine MACHINE] [--skip-existing-successful]
               [--skip-existing-failed] [--skip-existing-commits]
               [--skip-existing] [--record-samples] [--append-samples]
               [--interleave-rounds] [--no-interleave-rounds] [--no-pull]
               [--verbose] [--config CONFIG] [--version]
               [range]

This runs a benchmark suite in a mode that is useful during development. It is
equivalent to ``asv run --python=same``

positional arguments:
  range                 Range of commits to benchmark. For a git repository,
                        this is passed as the first argument to ``git rev-
                        list``; or Mercurial log command. See 'specifying
                        ranges' section of the `gitrevisions` manpage, or 'hg
                        help revisions', for more info. Also accepts the
                        special values 'NEW', 'ALL', 'EXISTING', and
                        'HASHFILE:xxx'. 'NEW' will benchmark all commits since
                        the latest benchmarked on this machine. 'ALL' will
                        benchmark all commits in the project. 'EXISTING' will
                        benchmark against all commits for which there are
                        existing benchmarks on any machine. 'HASHFILE:xxx'
                        will benchmark only a specific set of hashes given in
                        the file named 'xxx' ('-' means stdin), which must
                        have one hash per line. By default, will benchmark the
                        head of each configured of the branches.

options:
  -h, --help            show this help message and exit
  --date-period DATE_PERIOD
                        Pick only one commit in each given time period. For
                        example: 1d (daily), 1w (weekly), 1y (yearly).
  --steps STEPS, -s STEPS
                        Maximum number of steps to benchmark. This is used to
                        subsample the commits determined by range to a
                        reasonable number.
  --bench BENCH, -b BENCH
                        Regular expression(s) for benchmark to run. When not
                        provided, all benchmarks are run.
  --attribute ATTRIBUTE, -a ATTRIBUTE
                        Override a benchmark attribute, e.g. `-a repeat=10`.
  --cpu-affinity ATTRIBUTE
                        Set CPU affinity for running the benchmark, in format:
                        0 or 0,1,2 or 0-3. Default: not set
  --profile, -p         In addition to timing, run the benchmarks through the
```

(continues on next page)

(continued from previous page)

```
                      `cProfile` profiler and store the results.
--parallel [PARALLEL], -j [PARALLEL]
                      Build (but don't benchmark) in parallel. The value is
                      the number of CPUs to use, or if no number provided,
                      use the number of cores on this machine.
--show-stderr, -e     Display the stderr output from the benchmarks.
--durations [N]       Display total duration for N (or 'all') slowest
                      benchmarks
--quick, -q           Do a "quick" run, where each benchmark function is run
                      only once. This is useful to find basic errors in the
                      benchmark functions faster. The results are unlikely
                      to be useful, and thus are not saved.
-E ENV_SPEC, --environment ENV_SPEC
                      Specify the environment and Python versions for
                      running the benchmarks. String of the format
                      'environment_type:python_version', for example
                      'conda:3.12'. If the Python version is not specified,
                      all those listed in the configuration file are run.
                      The special environment type
                      'existing:/path/to/python' runs the benchmarks using
                      the given Python interpreter; if the path is omitted,
                      the Python running asv is used. For 'existing', the
                      benchmarked project must be already installed,
                      including all dependencies. The default value is
                      'existing:same'
--python PYTHON       Same as --environment=:PYTHON
--set-commit-hash SET_COMMIT_HASH
                      Set the commit hash to use when recording benchmark
                      results. This makes results to be saved also when
                      using an existing environment.
--launch-method {auto,spawn,forkserver}
                      How to launch benchmarks. Choices: auto, spawn,
                      forkserver
--dry-run, -n         Do not save any results to disk.
--machine MACHINE, -m MACHINE
                      Use the given name to retrieve machine information. If
                      not provided, the hostname is used. If no entry with
                      that name is found, and there is only one entry in
                      ~/.asv-machine.json, that one entry will be used.
--skip-existing-successful
                      Skip running benchmarks that have previous successful
                      results
--skip-existing-failed
                      Skip running benchmarks that have previous failed
                      results
--skip-existing-commits
                      Skip running benchmarks for commits that have existing
                      results
--skip-existing, -k   Skip running benchmarks that have previous successful
                      or failed results
--record-samples      Store raw measurement samples, not only statistics
--append-samples      Combine new measurement samples with previous results,
```

(continues on next page)

(continued from previous page)

```
                        instead of discarding old results. Implies --record-
                        samples. The previous run must also have been run with
                        --record/append-samples.
  --interleave-rounds   Interleave benchmarks with multiple rounds across
                        commits. This can avoid measurement biases from commit
                        ordering, can take longer.
  --no-interleave-rounds
  --no-pull             Do not pull the repository
  --verbose, -v         Increase verbosity
  --config CONFIG       Benchmark configuration file
  --version             Print program version
```

5.3.7 asv continuous

```
usage: asv continuous [-h] [--no-record-samples] [--append-samples] [--quick]
                      [--interleave-rounds] [--no-interleave-rounds]
                      [--factor FACTOR] [--no-stats] [--split]
                      [--only-changed] [--no-only-changed]
                      [--sort {name,ratio,default}] [--show-stderr]
                      [--bench BENCH] [--attribute ATTRIBUTE]
                      [--cpu-affinity ATTRIBUTE] [--machine MACHINE]
                      [-E ENV_SPEC] [--python PYTHON]
                      [--launch-method {auto,spawn,forkserver}] [--verbose]
                      [--config CONFIG] [--version]
                      [base] branch

Run a side-by-side comparison of two commits for continuous integration.

positional arguments:
  base                  The commit/branch to compare against. By default, the
                        parent of the tested commit.
  branch                The commit/branch to test. By default, the first
                        configured branch.

options:
  -h, --help            show this help message and exit
  --no-record-samples   Do not store raw measurement samples, but only
                        statistics
  --append-samples      Combine new measurement samples with previous results,
                        instead of discarding old results. Implies --record-
                        samples. The previous run must also have been run with
                        --record/append-samples.
  --quick, -q           Do a "quick" run, where each benchmark function is run
                        only once. This is useful to find basic errors in the
                        benchmark functions faster. The results are unlikely
                        to be useful, and thus are not saved.
  --interleave-rounds   Interleave benchmarks with multiple rounds across
                        commits. This can avoid measurement biases from commit
                        ordering, can take longer.
  --no-interleave-rounds
```

(continues on next page)

(continued from previous page)

```
--factor FACTOR, -f FACTOR
                      The factor above or below which a result is considered
                      problematic. For example, with a factor of 1.1 (the
                      default value), if a benchmark gets 10% slower or
                      faster, it will be displayed in the results list.
--no-stats            Do not use result statistics in comparisons, only
                      `factor` and the median result.
--split, -s           Split the output into a table of benchmarks that have
                      improved, stayed the same, and gotten worse.
--only-changed        Whether to show only changed results.
--no-only-changed
--sort {name,ratio,default}
                      Sort order
--show-stderr, -e     Display the stderr output from the benchmarks.
--bench BENCH, -b BENCH
                      Regular expression(s) for benchmark to run. When not
                      provided, all benchmarks are run.
--attribute ATTRIBUTE, -a ATTRIBUTE
                      Override a benchmark attribute, e.g. `-a repeat=10`.
--cpu-affinity ATTRIBUTE
                      Set CPU affinity for running the benchmark, in format:
                      0 or 0,1,2 or 0-3. Default: not set
--machine MACHINE, -m MACHINE
                      Use the given name to retrieve machine information. If
                      not provided, the hostname is used. If no entry with
                      that name is found, and there is only one entry in
                      ~/.asv-machine.json, that one entry will be used.
-E ENV_SPEC, --environment ENV_SPEC
                      Specify the environment and Python versions for
                      running the benchmarks. String of the format
                      'environment_type:python_version', for example
                      'conda:3.12'. If the Python version is not specified,
                      all those listed in the configuration file are run.
                      The special environment type
                      'existing:/path/to/python' runs the benchmarks using
                      the given Python interpreter; if the path is omitted,
                      the Python running asv is used. For 'existing', the
                      benchmarked project must be already installed,
                      including all dependencies. By default, uses the
                      values specified in the configuration file.
--python PYTHON       Same as --environment=:PYTHON
--launch-method {auto,spawn,forkserver}
                      How to launch benchmarks. Choices: auto, spawn,
                      forkserver
--verbose, -v         Increase verbosity
--config CONFIG       Benchmark configuration file
--version             Print program version
```

5.3.8 asv find

```
usage: asv find [-h] [--invert] [--skip-save] [--parallel [PARALLEL]]
                [--show-stderr] [--machine MACHINE] [-E ENV_SPEC]
                [--python PYTHON] [--launch-method {auto,spawn,forkserver}]
                [--verbose] [--config CONFIG] [--version]
                from..to benchmark_name

Adaptively searches a range of commits for one that produces a large
regression. This only works well when the regression in the range is mostly
monotonic.

positional arguments:
  from..to              Range of commits to search. For a git repository, this
                        is passed as the first argument to ``git log``. See
                        'specifying ranges' section of the `gitrevisions`
                        manpage for more info.
  benchmark_name        Name of benchmark to use in search.

options:
  -h, --help            show this help message and exit
  --invert, -i          Search for a decrease in the benchmark value, rather
                        than an increase.
  --skip-save           Do not save intermediate results from the search
  --parallel [PARALLEL], -j [PARALLEL]
                        Build (but don't benchmark) in parallel. The value is
                        the number of CPUs to use, or if no number provided,
                        use the number of cores on this machine.
  --show-stderr, -e     Display the stderr output from the benchmarks.
  --machine MACHINE, -m MACHINE
                        Use the given name to retrieve machine information. If
                        not provided, the hostname is used. If no entry with
                        that name is found, and there is only one entry in
                        ~/.asv-machine.json, that one entry will be used.
  -E ENV_SPEC, --environment ENV_SPEC
                        Specify the environment and Python versions for
                        running the benchmarks. String of the format
                        'environment_type:python_version', for example
                        'conda:3.12'. If the Python version is not specified,
                        all those listed in the configuration file are run.
                        The special environment type
                        'existing:/path/to/python' runs the benchmarks using
                        the given Python interpreter; if the path is omitted,
                        the Python running asv is used. For 'existing', the
                        benchmarked project must be already installed,
                        including all dependencies. By default, uses the
                        values specified in the configuration file.
  --python PYTHON       Same as --environment=:PYTHON
  --launch-method {auto,spawn,forkserver}
                        How to launch benchmarks. Choices: auto, spawn,
                        forkserver
  --verbose, -v         Increase verbosity
  --config CONFIG       Benchmark configuration file
```

(continues on next page)

(continued from previous page)

```
  --version             Print program version
```

5.3.9 asv rm

```
usage: asv rm [-h] [-y] [--verbose] [--config CONFIG] [--version]
              patterns [patterns ...]

Removes entries from the results database.

positional arguments:
  patterns         Pattern(s) to match, each of the form X=Y. X may be one of
                   "benchmark", "commit_hash", "python" or any of the machine
                   or environment params. Y is a case-sensitive glob pattern.

options:
  -h, --help       show this help message and exit
  -y               Don't prompt for confirmation.
  --verbose, -v    Increase verbosity
  --config CONFIG  Benchmark configuration file
  --version        Print program version
```

5.3.10 asv publish

```
usage: asv publish [-h] [--no-pull] [--html-dir HTML_DIR] [--verbose]
                   [--config CONFIG] [--version]
                   [range]

Collate all results into a website. This website will be written to the
``html_dir`` given in the ``asv.conf.json`` file, and may be served using any
static web server.

positional arguments:
  range                 Optional commit range to consider

options:
  -h, --help            show this help message and exit
  --no-pull             Do not pull the repository
  --html-dir HTML_DIR, -o HTML_DIR
                        Optional output directory. Default is 'html_dir' from
                        asv config
  --verbose, -v         Increase verbosity
  --config CONFIG       Benchmark configuration file
  --version             Print program version
```

5.3.11 asv preview

```
usage: asv preview [-h] [--port PORT] [--browser] [--html-dir HTML_DIR]
                   [--verbose] [--config CONFIG] [--version]

Preview the results using a local web server

options:
  -h, --help            show this help message and exit
  --port PORT, -p PORT  Port to run webserver on. [8080]
  --browser, -b         Open in webbrowser
  --html-dir HTML_DIR, -o HTML_DIR
                        Optional output directory. Default is 'html_dir' from
                        asv config
  --verbose, -v         Increase verbosity
  --config CONFIG       Benchmark configuration file
  --version             Print program version
```

5.3.12 asv profile

```
usage: asv profile [-h] [--gui GUI] [--output OUTPUT] [--force] [-E ENV_SPEC]
                   [--python PYTHON] [--launch-method {auto,spawn,forkserver}]
                   [--verbose] [--config CONFIG] [--version]
                   benchmark [revision]

Profile a benchmark

positional arguments:
  benchmark             The benchmark to profile. Must be a fully-specified
                        benchmark name. For parameterized benchmark, it must
                        include the parameter combination to use, e.g.:
                        benchmark_name\(param0, param1, ...\)
  revision              The revision of the project to profile. May be a
                        commit hash, or a tag or branch name.

options:
  -h, --help            show this help message and exit
  --gui GUI, -g GUI     Display the profile in the given gui. Use --gui=list
                        to list available guis.
  --output OUTPUT, -o OUTPUT
                        Save the profiling information to the given file. This
                        file is in the format written by the `cProfile`
                        standard library module. If not provided, prints a
                        simple text-based profiling report to the console.
  --force, -f           Forcibly re-run the profile, even if the data already
                        exists in the results database.
  -E ENV_SPEC, --environment ENV_SPEC
                        Specify the environment and Python versions for
                        running the benchmarks. String of the format
                        'environment_type:python_version', for example
                        'conda:3.12'. If the Python version is not specified,
```

(continues on next page)

(continued from previous page)

```
                        all those listed in the configuration file are run.
                        The special environment type
                        'existing:/path/to/python' runs the benchmarks using
                        the given Python interpreter; if the path is omitted,
                        the Python running asv is used. For 'existing', the
                        benchmarked project must be already installed,
                        including all dependencies. By default, uses the
                        values specified in the configuration file.
  --python PYTHON       Same as --environment=:PYTHON
  --launch-method {auto,spawn,forkserver}
                        How to launch benchmarks. Choices: auto, spawn,
                        forkserver
  --verbose, -v         Increase verbosity
  --config CONFIG       Benchmark configuration file
  --version             Print program version
```

5.3.13 asv update

```
usage: asv update [-h] [--verbose] [--config CONFIG] [--version]

Update the results and config files to the current version

options:
  -h, --help       show this help message and exit
  --verbose, -v    Increase verbosity
  --config CONFIG  Benchmark configuration file
  --version        Print program version
```

5.3.14 asv show

```
usage: asv show [-h] [--details] [--durations] [--bench BENCH]
                [--attribute ATTRIBUTE] [--cpu-affinity ATTRIBUTE]
                [--machine MACHINE] [-E ENV_SPEC] [--python PYTHON]
                [--verbose] [--config CONFIG] [--version]
                [commit]

Print saved benchmark results.

positional arguments:
  commit                The commit to show data for

options:
  -h, --help            show this help message and exit
  --details             Show all result details
  --durations           Show only run durations
  --bench BENCH, -b BENCH
                        Regular expression(s) for benchmark to run. When not
                        provided, all benchmarks are run.
  --attribute ATTRIBUTE, -a ATTRIBUTE
```

(continues on next page)

(continued from previous page)

```
                        Override a benchmark attribute, e.g. `-a repeat=10`.
  --cpu-affinity ATTRIBUTE
                        Set CPU affinity for running the benchmark, in format:
                        0 or 0,1,2 or 0-3. Default: not set
  --machine MACHINE, -m MACHINE
                        Use the given name to retrieve machine information. If
                        not provided, the hostname is used. If no entry with
                        that name is found, and there is only one entry in
                        ~/.asv-machine.json, that one entry will be used.
  -E ENV_SPEC, --environment ENV_SPEC
                        Specify the environment and Python versions for
                        running the benchmarks. String of the format
                        'environment_type:python_version', for example
                        'conda:3.12'. If the Python version is not specified,
                        all those listed in the configuration file are run.
                        The special environment type
                        'existing:/path/to/python' runs the benchmarks using
                        the given Python interpreter; if the path is omitted,
                        the Python running asv is used. For 'existing', the
                        benchmarked project must be already installed,
                        including all dependencies. By default, uses the
                        values specified in the configuration file.
  --python PYTHON       Same as --environment=:PYTHON
  --verbose, -v         Increase verbosity
  --config CONFIG       Benchmark configuration file
  --version             Print program version
```

5.3.15 asv compare

```
usage: asv compare [-h] [--factor FACTOR] [--no-stats] [--split]
                   [--only-changed] [--no-only-changed]
                   [--sort {name,ratio,default}] [--machine MACHINE]
                   [-E ENV_SPEC] [--python PYTHON] [--verbose]
                   [--config CONFIG] [--version]
                   revision1 revision2

Compare two sets of results

positional arguments:
  revision1             The reference revision.
  revision2             The revision being compared.

options:
  -h, --help            show this help message and exit
  --factor FACTOR, -f FACTOR
                        The factor above or below which a result is considered
                        problematic. For example, with a factor of 1.1 (the
                        default value), if a benchmark gets 10% slower or
                        faster, it will be displayed in the results list.
  --no-stats            Do not use result statistics in comparisons, only
```

(continues on next page)

(continued from previous page)

```
                        `factor` and the median result.
  --split, -s           Split the output into a table of benchmarks that have
                        improved, stayed the same, and gotten worse.
  --only-changed        Whether to show only changed results.
  --no-only-changed
  --sort {name,ratio,default}
                        Sort order
  --machine MACHINE, -m MACHINE
                        The machine to compare the revisions for.
  -E ENV_SPEC, --environment ENV_SPEC
                        Specify the environment and Python versions for
                        running the benchmarks. String of the format
                        'environment_type:python_version', for example
                        'conda:3.12'. If the Python version is not specified,
                        all those listed in the configuration file are run.
                        The special environment type
                        'existing:/path/to/python' runs the benchmarks using
                        the given Python interpreter; if the path is omitted,
                        the Python running asv is used. For 'existing', the
                        benchmarked project must be already installed,
                        including all dependencies. By default, uses the
                        values specified in the configuration file.
  --python PYTHON       Same as --environment=:PYTHON
  --verbose, -v         Increase verbosity
  --config CONFIG       Benchmark configuration file
  --version             Print program version
```

5.3.16 asv check

```
usage: asv check [-h] [-E ENV_SPEC] [--python PYTHON] [--verbose]
                 [--config CONFIG] [--version]

This imports and checks basic validity of the benchmark suite, but does not
run the benchmark target code

options:
  -h, --help            show this help message and exit
  -E ENV_SPEC, --environment ENV_SPEC
                        Specify the environment and Python versions for
                        running the benchmarks. String of the format
                        'environment_type:python_version', for example
                        'conda:3.12'. If the Python version is not specified,
                        all those listed in the configuration file are run.
                        The special environment type
                        'existing:/path/to/python' runs the benchmarks using
                        the given Python interpreter; if the path is omitted,
                        the Python running asv is used. For 'existing', the
                        benchmarked project must be already installed,
                        including all dependencies. By default, uses the
                        values specified in the configuration file.
```

(continues on next page)

(continued from previous page)

```
  --python PYTHON       Same as --environment=:PYTHON
  --verbose, -v         Increase verbosity
  --config CONFIG       Benchmark configuration file
  --version             Print program version
```

5.3.17 asv gh-pages

```
usage: asv gh-pages [-h] [--no-push] [--rewrite] [--verbose] [--config CONFIG]
                    [--version]

Publish the results to github pages Updates the 'gh-pages' branch in the
current repository, and pushes it to 'origin'.

options:
  -h, --help       show this help message and exit
  --no-push        Update local gh-pages branch but don't push
  --rewrite        Rewrite gh-pages branch to contain only a single commit,
                   instead of adding a new commit
  --verbose, -v    Increase verbosity
  --config CONFIG  Benchmark configuration file
  --version        Print program version
```

5.4 ASV environment variables

Benchmarking and build commands are run with the following environment variables available:

- `ASV`: `true`
- `ASV_PROJECT`: the project name from the configuration file
- `ASV_ENV_NAME`: name of the currently active environment
- `ASV_ENV_TYPE`: type of the currently active environment
- `ASV_ENV_DIR`: full path to the currently active environment root
- `ASV_CONF_DIR`: full path to the directory where `asv.conf.json` is
- `ASV_BUILD_DIR`: full path to the build directory (checked-out source path + `repo_subdir`)
- `ASV_BUILD_CACHE_DIR`: full path to the build cache directory
- `ASV_COMMIT`: commit hash of currently installed project

If there is no asv-managed environment, build, or cache directory, or commit hash, those environment variables are unset.

The following environment variables controlling Python and other behavior are also set:

- `PATH`: environment-specific binary directories prepended
- `PIP_USER`: `false`
- `PYTHONNOUSERSITE`: `True` (for conda environments only)
- `PYTHONPATH`: unset (if really needed, can be overridden by setting `ASV_PYTHONPATH`)

Note:

5.4.1 Custom environment variables

You can send custom environment variables to build and benchmarking commands by configuring the `matrix` setting in `asv.conf.json`.

CHAPTER

SIX

DEVELOPER DOCS

This section describes some things that may be of interest to developers and other people interested in internals of `asv`.

Note: From version `0.6.0` on-wards, functionality in `asv` has been split into the section needed by the code being benchmarked (`asv_runner`) and the rest of `asv`. This means that the `asv` documentation covers setting up environments, loading plugins, and collecting the results of the benchmarks run with `asv_runner`.

Contents

6.1 Development setup

The required packages to the full `asv` test suite, are listed in `requirements-dev.txt`. The minimal set of packages required for testing are: `pytest virtualenv filelock six pip setuptools wheel`.

6.2 Separation of concerns

asv consists of the following steps:

- Setting up an environment for the project
- Building the project within the environment
- Running the benchmarks
- Collecting results and visualizing them after analysis for regressions

---

Note: Conceptually there are two separate parts to this process. There is the `main` process which orchestrates the environment creation. This is followed by a subprocess which essentially runs the project benchmarks. This subprocess must have only minimal dependencies, ideally nothing beyond the minimum Python version needed to run `asv` along with the dependencies of the project itself.

Changed in version 0.6.0: To clarify this, starting from `v0.6.0`, `asv` has been split into `asv_runner`, which is responsible for loading benchmark types, discovering them and running them within an environment, while the `asv` repository handles the remaining tasks.

6.3 Benchmark suite layout and file formats

A benchmark suite directory has the following layout. The `$`-prefixed variables refer to values in the `asv.conf.json` file.

- `asv.conf.json`: The configuration file. See *asv.conf.json reference*.
- `$benchmark_dir`: Contains the benchmark code, created by the user. Each subdirectory needs an `__init__.py`.
- `$project/`: A clone of the project being benchmarked. Information about the history is grabbed from here, but the actual building happens in the environment-specific clones described below.
- `$env_dir/`: Contains the environments used for building and benchmarking. There is one environment in here for each specific combination of Python version and library dependency. Generally, the dependencies are only installed once, and then reused on subsequent runs of `asv`, but the project itself needs to be rebuilt for each commit being benchmarked.
  - `$ENVIRONMENT_HASH/`: The directory name of each environment is the md5hash of the list of dependencies and the Python version. This is not very user friendly, but this keeps the filename within reasonable limits.
    - `asv-env-info.json`: Contains information about the environment, mainly the Python version, dependencies and build environment variables used.
    - `project/`: An environment-specific clone of the project repository. Each environment has its own clone so that builds can be run in parallel without fear of clobbering (particularly for projects that generate source files outside of the `build/` directory. These clones are created from the main `$project/` directory using the `--shared` option to `git clone` so that the repository history is stored in one place to save on disk space.

      The project is built in this directory with the standard `python setup.py build` command. This means repeated builds happen in the same place and ccache is able to cache and reuse many of the build products.
    - `wheels/`: If `build_cache_size` in `asv.conf.json` is set to something other than 0, this contains Wheels of the last N project builds for this environment. In this way, if a build for a particular commit

has already been performed and cached, it can be restored much more quickly. Each subdirectory is a commit hash, containing one `.whl` file and a timestamp.

* `usr/`, `lib/`, `bin/` etc.: These are the virtualenv or Conda environment directories that we install the project into and then run benchmarks from.

- `$results_dir/`: This is the "database" of results from benchmark runs.
  - `benchmarks.json`: Contains metadata about all of the benchmarks in the suite. It is a dictionary from benchmark names (a fully-qualified dot-separated path) to dictionaries containing information about that benchmark. Useful keys include:
    * `code`: The Python code of the benchmark
    * `params`: List of lists describing parameter values of a parameterized benchmark. If benchmark is not parameterized, an empty list. Otherwise, the n-th entry of the list is a list of the Python `repr()` strings for the values the n-th parameter should loop over.
    * `param_names`: Names for parameters for a parameterized benchmark. Must be of the same length as the `params` list.
    * `version`: An arbitrary string identifying the benchmark version. Default value is hash of `code`, but user can override.

    Other keys are specific to the kind of benchmark, and correspond to *Benchmark attributes*.
  - `MACHINE/`: Within the results directory is a directory for each machine. Putting results from different machines in separate directories makes the results trivial to merge, which is useful when benchmarking across different platforms or architectures.
    * `HASH-pythonX.X-depA-depB.json`: Each JSON file within a particular machine represents a run of benchmarks for a particular project commit in a particular environment. Contains the keys:
      · `version`: the value `2`.
      · `commit_hash`: The project commit that the benchmarks were run on.
      · `env_name`: Name of the environment the benchmarks were run in.
      · `date`: A JavaScript date stamp of the date of the commit (not when the benchmarks were run).
      · `params`: Information about the machine the benchmarks were run on.
      · `python`: Python version of the environment.
      · `requirements`: Requirements dictionary of the environment.
      · `env_vars`: Environment variable dictionary of the environment.
      · `durations`: Duration information for build and setup-cache timings.
      · `result_columns`: List of column names for the `results` dictionary. It is `["result", "params", "version", "started_at", "duration", "stats_ci_99_a", "stats_ci_99_b", "stats_q_25", "stats_q_75", "stats_number", "stats_repeat", "samples", "profile"]` currently.
      · `results`: A dictionary from benchmark names to benchmark results. The keys are benchmark names, and values are lists such that `dict(zip(result_columns, results[benchmark_name]))` pairs the appropriate keys with the values; in particular, trailing columns with missing values can be dropped.

        Some items, marked with "(param-list)" below, are lists with items corresponding to results from a parametrized benchmark (see `params` below). Non-parametrized benchmarks then have lists with a single item.

Values except `params` can be `null`, indicating missing data.

Floating-point numbers in `stats_*` and `duration` are truncated to 5 significant base-10 digits when saving, in order to produce smaller JSON files.

- `result`: (param-list) contains the summarized result value(s) of the benchmark. The values are float, NaN or null.

  The values are either numbers indicating result from successful run, `null` indicating a failed benchmark, or `NaN` indicating a benchmark explicitly skipped by the benchmark suite.

- `params`: contains a copy of the parameter values of the benchmark, as described above. If the user has modified the benchmark after the benchmark was run, these may differ from the current values. The `result` value is a list of results. Each entry corresponds to one combination of the parameter values. The n-th entry in the list corresponds to the parameter combination `itertools.product(*params)[n]`, i.e., the results appear in cartesian product order, with the last parameters varying fastest.

  For non-parametrized benchmarks, `[]`.

- `version`: string, a benchmark version identifier. Results whose version is not equal to the current version of the benchmark are ignored. If the value is missing, no version comparisons are done (backward compatibility).

- `started_at`: Javascript timestamp indicating start time of latest benchmark run.

- `duration`: float, indicating the duration of a benchmark run in seconds.

- `stats_*`: (param-list) dictionary containing various statistical indicators. Possible `*` are `ci_99_a`, `ci_99_b` (confidence interval estimate lower/upper values), `q_25` (lower quartile), `q_75` (upper quartile), `repeat`, and `number`.

- `profile`: string, zlib-compressed and base64-encoded Python profile dump.

- `samples`: (param-list) List of samples obtained for a benchmark. The samples are in the order they were measured in.

- `$html_dir/`: The output of `asv publish`, that turns the raw results in `$results_dir/` into something viewable in a web browser. It is an important feature of `asv` that the results can be shared on a static web server, so there is no server side component, and the result data is accessed through AJAX calls from JavaScript. Most of the files at the root of `$html_dir/` are completely static and are just copied verbatim from `asv/www/` in the source tree.

  - `index.json`: Contains an index into the benchmark data, describing what is available. Important keys include:

    * `benchmarks`: A dictionary of benchmarks. At the moment, this is identical to the content in `$results_dir/benchmarks.json`.

    * `revision_to_hash`: A dictionary mapping revision number to commit hash. This allows to show commits tooltip in graph and commits involved in a regression.

    * `revision_to_date`: A dictionary mapping JavaScript date stamps to revisions (including tags). This allows the x-scale of a plot to be scaled by date.

    * `machines`: Describes the machines used for testing.

    * `params`: A dictionary of parameters against which benchmark results can be selected. Each entry is a list of valid values for that parameter.

    * `tags`: A dictionary of git tags and their revisions, so this information can be displayed in the plot.

- `graphs/`: This is a nested tree of directories where each level is a parameter from the `params` dictionary, in asciibetical order. The web interface, given a set of parameters that are set, get easily grab the associated graph.
  - `BENCHMARK_NAME.json`: At the leaves of this tree are the actual benchmark graphs. It contains a list of pairs, where each pair is of the form `(timestamp, result_value)`. For parameterized benchmarks, `result_value` is a list of results, corresponding to `itertools.product` iteration over the parameter combinations, similarly as in the result files. For non-parameterized benchmarks, it is directly the result. Missing values (eg. failed and skipped benchmarks) are represented by `null`.

6.4 Full-stack testing

For full-stack testing, we use Selenium WebDriver and its Python bindings. Additional documentation for Selenium Python bindings is here.

The browser back-end can be selected via:

```
pytest --webdriver=PhantomJS
```

The allowed values include `None` (default), `PhantomJS`, `Chrome`, `Firefox`, `ChromeHeadless`, `FirefoxHeadless`, or arbitrary Python code initializing a Selenium webdriver instance.

To use them, at least one of the following needs to be installed:

- Firefox GeckoDriver: Firefox-based controllable browser.
- ChromeDriver: Chrome-based controllable browser. On Ubuntu, install via `apt-get install chromium-chromedriver`, on Fedora via `dnf install chromedriver`.
- PhantomJS: Headless web browser (discontinued, prefer using Firefox or Chrome).

For other options regarding the webdriver to use, see `py.test --help`.

6.5 Step detection

Regression detection in ASV is based on detecting stepwise changes in the graphs. The assumptions on the data are as follows: the curves are piecewise constant plus random noise. We don't know the scaling of the data or the amplitude of the noise, but assume the relative weight of the noise amplitude is known for each data point.

ASV measures the noise amplitude of each data point, based on a number of samples. We use this information for weighting the different data points:

$$\sigma_j = \sigma \mathrm{CI}_{99} = \sigma / w_j$$

i.e., we assume the uncertainty in each measurement point is proportional to the estimated confidence interval for each data point. Their inverses are taken as the relative weights `w_j`. If `w_j=0` or undefined, we replace it with the median weight, or with 1 if all are undefined. The step detection algorithm determines the absolute noise amplitude itself based on all available data, which is more robust than relying on the individual measurements.

Step detection is a well-studied problem. In this implementation, we mainly follow a variant of the approach outlined in [Friedrich2008] and elsewhere. This provides a fast algorithm for solving the piecewise weighted fitting problem

$$\underset{k,\{j\},\{\mu\}}{\operatorname{argmin}} \gamma k + \sum_{r=1}^{k} \sum_{i=j_{r-1}}^{j_r} w_i |y_i - \mu_r| \tag{6.1}$$

The differences are: as we do not need exact solutions, we add additional heuristics to work around the $\mathcal{O}(n^2)$ scaling, which is too harsh for pure-Python code. For details, see `asv.step_detect.solve_potts_approx`. Moreover, we follow a slightly different approach on obtaining a suitable number of intervals, by selecting an optimal value for γ, based on a variant of the information criterion problem discussed in [Yao1988].

6.5.1 Bayesian information

To proceed, we need an argument by which to select a suitable γ in (6.1). Some of the literature on step detection, e.g. [Yao1988], suggests results based on Schwarz information criteria,

$$\mathrm{SC} = \frac{m}{2} \ln \sigma^2 + k \ln m = \mathrm{min!} \tag{6.2}$$

where σ^2 is maximum likelihood variance estimator (if noise is gaussian). For the implementation, see `asv.step_detect.solve_potts_autogamma`.

What follows is a handwaving plausibility argument why such an objective function makes sense, and how to end up with l_1 rather than gaussians. Better approaches are probably to be found in step detection literature. If you have a better formulation, contributions/corrections are welcome!

We assume a Bayesian model:

$$P(\{y_i\}_{i=1}^m | \sigma, k, \{\mu_i\}_{i=1}^k, \{j_i\}_{i=1}^{k-1}) = N \sigma^{-m} \exp(-\sigma^{-1} \sum_{r=1}^{k} \sum_{i=j_{r-1}+1}^{j_r} w_i |y_i - \mu_r|) \tag{6.3}$$

Here, y_i are the m data points at hand, k is the number of intervals, μ_i are the values of the function at the intervals, j_i are the interval breakpoints; $j_0 = 0$, $j_k = m$, $j_{r-1} < j_r$. The noise is assumed Laplace rather than gaussian, which results to the more robust l_1 norm fitting rather than l_2. The noise amplitude σ is not known. N is a normalization constant that depends on m but not on the other parameters.

The optimal k comes from Bayesian reasoning: $\hat{k} = \mathrm{argmax}_k \, P(k|\{y\})$, where

$$P(k|\{y\}) = \frac{\pi(k)}{\pi(\{y\})} \int d\sigma (d\mu)^k \sum_{\{j\}} P(\{y\}|\sigma, k, \{\mu\}, \{j\}) \pi(\sigma, \{\mu\}, \{j\}|k) \tag{6.4}$$

The prior $\pi(\{y\})$ does not matter for $\hat{k}$; the other priors are assumed flat. We would need to estimate the behavior of the integral in the limit $m \to \infty$. We do not succeed in doing this rigorously here, although it might be done in the literature.

Consider first saddle-point integration over $\{\mu\}$, expanding around the max-likelihood values μ_r^*. The max-likelihood estimates are the weighted medians of the data points in each interval. Change in the exponent when μ is perturbed is

$$\Delta = -\sigma^{-1} \sum_{r=1}^{k} \sum_{i=j_{r-1}+1}^{j_r} w_i [|y_i - \mu_r^* - \delta\mu_r| - |y_i - \mu_r^*|] \tag{6.5}$$

Note that $\sum_{i=j_{r-1}+1}^{j_r} w_i \mathrm{sgn}(y_i - \mu_r^*) = 0$, so that response to small variations $\delta\mu_r$ is m-independent. For larger variations, we have

$$\Delta = -\sigma^{-1} \sum_{r=1}^{k} N_r(\delta\mu_r) |\delta\mu_r| \tag{6.6}$$

where $N_r(\delta\mu) = \sum_i w_i s_i$ where $s_i = \pm 1$ depending on whether y_i is above or below the perturbed median. Let us assume that in a typical case, $N_r(\delta\mu) \sim m_r \bar{W}_r^2 \delta\mu/\sigma$ where $\bar{W}_r = \frac{1}{m_r} \sum_i w_i$ is the average weight of the interval and m_r the number of points in the interval. This recovers a result we would have obtained in the gaussian noise case

$$\Delta \sim -\sigma^{-2} \sum_r W_r^2 m_r |\delta\mu_r|^2 \tag{6.7}$$

For the gaussian case, this would not have required any questionable assumptions. After integration over $\{\delta\mu\}$ we are left with

$$\int(\ldots) \propto \int d\sigma \sum_{\{j\}} (2\pi)^{k/2} \sigma^k [\bar{W}_1 \cdots \bar{W}_k]^{-1} [m_1 \cdots m_k]^{-1/2} P(\{y\}|\sigma, k, \{\mu_*\}, \{j\}) \pi(\sigma, \{j\}|k) \tag{6.8}$$

We now approximate the rest of the integrals/sums with only the max-likelihood terms, and assume $m_j^* \sim m/k$. Then,

$$\begin{aligned} \ln P(k|\{y\}) &\simeq C_1(m) + C_2(k) + \frac{k}{2}\ln(2\pi) - \frac{k}{2}\ln(m/k) - k\ln\bar{W} + \ln P(\{y\}|\sigma_*, k, \{\mu_*\}, \{j_*\}) \\ &\approx \tilde{C}_1(m) + \tilde{C}_2(k) - \frac{k}{2}\ln m + \ln P(\{y\}|\sigma_*, k, \{\mu_*\}, \{j_*\}) \end{aligned} \tag{6.9}$$

where we neglect terms that don't affect asymptotics for $m \to \infty$, and C are some constants not depending on both m, k. The result is of course the Schwarz criterion for k free model parameters. We can suspect that the factor $k/2$ should be replaced by a different number, since we have $2k$ parameters. If also the other integrals/sums can be approximated in the same way as the $\{\mu\}$ ones, we should obtain the missing part.

Substituting in the max-likelihood value

$$\sigma_* = \frac{1}{m}\sum_{r=1}^{k}\sum_{i=j_{r-1}^*+1}^{j_r^*} w_i|y_i - \mu_r^*| \tag{6.10}$$

we get

$$\ln P(k|\{y\}) \sim C - \frac{k}{2}\ln m - m\ln\sum_{r=1}^{k}\sum_{i=j_{r-1}^*}^{j_r^*} w_i|y_i - \mu_r^*| \tag{6.11}$$

This is now similar to (6.2), apart from numerical prefactors. The final fitting problem then becomes

$$\operatorname*{argmin}_{k,\{j\},\{\mu\}} r(m)k + \ln\sum_{r=1}^{k}\sum_{i=j_{r-1}}^{j_r} w_i|y_i - \mu_r| \tag{6.12}$$

with $r(m) = \frac{\ln m}{2m}$. Note that it is invariant vs. rescaling of weights $w_i \mapsto \alpha w_i$, i.e., the invariance of the original problem is retained. As we know this function $r(m)$ is not necessarily completely correct, and it seems doing the calculation rigorously requires more effort than can be justified by the requirements of the application, we now take a pragmatic view and fudge the function to $r(m) = \beta\frac{\ln m}{m}$ with β chosen so that things appear to work in practice for the problem at hand.

According to [Friedrich2008], problem (6.12) can be solved in $\mathcal{O}(n^3)$ time. This is too slow, however. We can however approach this on the basis of the easier problem (6.1). It produces a family of solutions $[k^*(\gamma), \{\mu^*(\gamma)\}, \{j^*(\gamma)\}]$. We now evaluate (6.12) restricted to the curve parameterized by γ. In particular, $[\{\mu^*(\gamma)\}, \{j^*(\gamma)\}]$ solves (6.12) under the constraint $k = k^*(\gamma)$. If $k^*(\gamma)$ obtains all values in the set $\{1, \ldots, m\}$ when γ is varied, the original problem is solved completely. This probably is not a far-fetched assumption; in practice it appears such Bayesian information criterion provides a reasonable way for selecting a suitable γ.

6.5.2 Overfitting

It's possible to fit any data perfectly by choosing size-1 intervals, one per each data point. For such a fit, the logarithm (6.12) gives $-\infty$ which then always minimizes SC. This artifact of the model above needs special handling.

Indeed, for $\sigma \to 0$, (6.3) reduces to

$$P(\{y_i\}_{i=1}^m|\sigma, k, \{\mu_i\}_{i=1}^k, \{j_i\}_{i=1}^{k-1}) = \prod_{r=1}^{k}\prod_{i=j_{r-1}+1}^{j_r}\delta(y_i - \mu_r) \tag{6.13}$$

which in (6.4) gives a contribution (assuming no repeated y-values)

$$P(k|\{y\}) = \frac{\pi(n)}{\pi(\{y\})}\delta_{n,k}\int d\sigma\pi(\sigma,\{y\},\{i\}|n)f(\sigma) + \ldots \tag{6.14}$$

with $f(\sigma) \to 1$ for $\sigma \to 0$. A similar situation occurs also in other cases where perfect fitting occurs (repeated y-values). With the flat, scale-free prior $\pi(\ldots) \propto 1/\sigma$ used above, the result is undefined.

A simple fix is to give up complete scale free-ness of the results, i.e., fixing a minimal noise level $\pi(\sigma,\{\mu\},\{j\}|k) \propto \theta(\sigma-\sigma_0)/\sigma$ with some $\sigma_0(\{\mu\},\{j\},k) > 0$. The effect in the σ integral is cutting off the log-divergence, so that with sufficient accuracy we can in (6.12) replace

$$\ln\sigma \mapsto \ln(\sigma_0+\sigma) \tag{6.15}$$

Here, we fix a measurement accuracy floor with the following guess: `sigma_0 = 0.1 * w0 * min(abs(diff(mu)))` and `sigma_0 = 0.001 * w0 * abs(mu)` when there is only a single interval. Here, `w0` is the median weight.

6.5.3 Autocorrelated noise

Practical experience shows that the noise in the benchmark results can be correlated. Often benchmarks are run for multiple commits at once, for example the new commits at a given time, and the benchmark machine does something else between the runs. Alternatively, the background load from other processes on the machine varies with time.

To give a basic model for the noise correlations, we include AR(1) Laplace noise in (6.3),

$$P(\{y_i\}_{i=1}^m|\sigma,\rho,k,\{\mu_i\}_{i=1}^k,\{j_i\}_{i=1}^{k-1}) = N\sigma^{-m}\exp(-\sigma^{-1}\sum_{r=1}^{k}\sum_{i=j_{r-1}+1}^{j_r}|\epsilon_{i,r}-\rho\epsilon_{i-1,r}|) \tag{6.16}$$

where $\epsilon_{i,r} = y_i - \mu_r$ with $\epsilon_{j_{r-1},r} = y_{j_{r-1}} - \mu_{r-1}$ and $\epsilon_{j_0,1} = 0$ are the deviations from the stepwise model. The correlation measure ρ is unknown, but assumed to be constant in $(-1,1)$.

Since the parameter ρ is global, it does not change the parameter counting part of the Schwarz criterion. The maximum likelihood term however does depend on ρ, so that the problem becomes:

$$\operatorname*{argmin}_{k,\rho,\{j\},\{\mu\}} r(m)k + \ln\sum_{r=1}^{k}\sum_{i=j_{r-1}}^{j_r}|\epsilon_{i,r}-\rho\epsilon_{i-1,r}| \tag{6.17}$$

To save computation time, we do not solve this optimization problem exactly. Instead, we again minimize along the $\mu_r^*(\gamma)$, $j_r^*(\gamma)$ curve provided by the solution to (6.1), and use (6.17) only in selecting the optimal value of the γ parameter.

The minimization vs. ρ can be done numerically for given $\mu_r^*(\gamma)$, $j_r^*(\gamma)$. This minimization step is computationally cheap compared to the piecewise fit, so including it will not significantly change the runtime of the total algorithm.

6.5.4 Postprocessing

For the purposes of regression detection, we do not report all steps the above approach provides. For details, see `asv.step_detect.detect_regressions`.

6.5.5 Making use of measured variance

`asv` measures also variance in the timings. This information is currently used to provide relative data weighting (see above).

CHAPTER

SEVEN

CHANGELOG

7.1 0.6.2 (2024-02-12)

7.1.1 New Features

- Partially skipped benchmarks will still have their results displayed. (#1351)
- `asv` will now correctly prepare all the build backend dependencies into `base_requirements` and the default `build_command` has been modified to allow fetching from PyPI. (#1377)

7.1.2 Bug Fixes

- The `asv` package no longer prepends the script execution directory unconditionally. Now we check for and remove the path only if it matches the directory that the runner script resides in. (#1346)
- The `bdist` wheels no longer include `benchmarks` and `test`. (#1349)
- The `mamba` plugin works correctly for newer versions (>=1.5) of `libmambapy` (#1372)
- The `mamba` plugin respects the `MAMBARC` environment if set, taking channels and channel priority from the file in the environment variable. (#1373)
- Fixed a bug where `matrix` requirements were dropped if an environment file was specified. (#1373)
- `conda-forge` is no longer a default channel for `mamba`. (#1373)

7.1.3 Other Changes and Additions

- `asv` now depends on `virtualenv` (#1379)

7.2 0.6.1 (2023-09-11)

7.2.1 Bug Fixes

- `pip` dependencies in `environment.yml` files for the `mamba` plugin are handled correctly (#1326)
- `asv.config.json` matrix requirements no longer need `pip+` set explicitly for calling the `pip` solver for `virtualenv`
- `asv` will now use `conda_environment_file` if it exists (#1325)

7.2.2 Other Changes and Additions

- `asv` timestamps via `datetime` are now Python 3.12 compatible (#1331)
- `asv` now provides `asv[virtualenv]` as an installable target
- `asv` now uses Github Actions exclusively for Windows and Linux

7.3 0.6.0 (2023-08-20)

7.3.1 New Features

- `asv_runner` is now used internally, making the addition of custom benchmark types viable (#1287)
- Benchmarks can be skipped, both wholly and in part using new decorators `skip_benchmark_if` and `skip_params_if` (#1309)
- Benchmarks can be skipped during their execution (after setup) by raising `SkipNotImplemented` (#1307)
- Added `default_benchmark_timeout` to the configuration object, can also be passed via `-a timeout=NUMBER` (#1308)
- `ASV_RUNNER_PATH` can be set from the terminal to test newer versions of `asv_runner` (#1312)

7.3.2 API Changes

- Removed `asv dev` in favor of using `asv run` with the right arguments (#1200)
- `asv run` and `asv continuous` don't implement the `--strict` option anymore, and they will always return a non-zero (i.e. 2) exit status if any benchmark fail.

7.3.3 Bug Fixes

- Fixed `install_timeout` for `conda` (#1310)
- Fixed handling of local `pip` matrix (#1312)
- Fixed the deadlock when mamba is used with an environment file. (#1300)
- Fixed environment file usage with mamba and recognizes default `environment.yml`. (#1303)

7.3.4 Other Changes and Additions

- `mamba` and `conda` use `environment.yml` if it exists
- `virtualenv` now requires `packaging` due to `distutils` deprecations (#1240)
- Wheels are now built for CPython `3.8, 3.9, 3.10, 3.11`

7.4 0.5.1 (2021-02-06)

7.4.1 Bug Fixes

- Packaging `requirements-dev.txt` file, used in `setup.py`. (#1013)

7.5 0.5 (2021-02-05)

7.5.1 New Features

- Adding environment variables to build and benchmark commands. (#809, #833)
- Added `--strict` option to `asv run` to set exit code on failure. (#865)
- Added `--no-stats` option to `asv compare` and `asv continuous`. (#879)
- Added `--durations` option to `asv run` and `asv show` for displaying benchmark run durations. (#838)
- Added `--date-period` option to `asv run` for running benchmarks for commits separated by a constant time interval. (#835)
- Web UI button to group regressions by benchmark. (#869)
- Space-saving v2 file format for storing results. (#847)
- `timeraw_*` benchmarks for measuring e.g. import times. (#832)
- Support for using conda environment files for env setup. (#793)

7.5.2 API Changes

- Results file format change requires `asv update` to update old data to v2 format.
- The configuration syntax for "matrix", "exclude", and "include" in `asv.conf.json` has changed. The old syntax is still supported, unless you are installing packages named `req`, `env`, `env_nobuild`.

7.5.3 Bug Fixes

- When an `asv find` step fails due to timeout, assume runtime equal to timeout to allow bisection to proceed (#768)
- Minor fixes and improvements (#897, #896, #888, #881, #877, #876, #875, #861, #870, #868, #867, #866, #864, #863, #857, #786, #854, #855, #852, #850, #844, #843, #842, #839, #841, #840, #837, #836, #834, #831, #830, #829, #828, #826, #825, #824)

7.5.4 Other Changes and Additions

- Uniqueness of `repr()` for `param` objects is now guaranteed by suffixing unique identifier corresponding to order of appearance. (#771)
- Memory addresses are now stripped from the `repr()` of `param` elements, allowing comparison across multiple runs. (#771)
- `asv dev` is now equivalent to `asv run` with `--python=same` default. (#874)
- `asv continuous` by default now records measurement samples, for better comparison statistics. (#878)
- ASV now uses PEP 518 `pyproject.toml` in packaging. (#853)

7.6 0.4.1 (2019-05-30)

- Change wheel installation default command to chdir away from build directory instead of `--force-install`. (#823)

7.7 0.4 (2019-05-26)

7.7.1 New Features

- `asv check` command for a quick check of benchmark suite validity. (#782)
- `asv run HASHFILE:filename` can read commit hashes to run from file or stdin (#768)
- `--set-commit-hash` option to `asv run`, which allows recording results from runs in "existing" environments not managed by asv (#794)
- `--cpu-affinity` option to `asv run` and others, to set CPU affinity (#769)
- "Hide legend" option in web UI (#807)
- `pretty_source` benchmark attribute for customizing source code shown (#810)
- Record number of cores in machine information (#761)

7.7.2 API Changes

- Default timer changed from `process_time()` to `timeit.default_timer()` to fix resolution issues on Windows. The old behavior can be restored by setting `Benchmark.timer = time.process_time` (#780)

7.7.3 Bug Fixes

- Fix pip command line in `install_command` (#806)
- Python 3.8 compatibility (#814)
- Minor fixes and improvements (#759, #764, #767, #772, #779, #783, #784, #787, #790, #795, #799, #804, #812, #813, #815, #816, #817, #818, #820)

7.7.4 Other Changes and Additions

- In case of significant changes `asv continuous` message now reports if performance decreased or increased.

7.8 0.3.1 (2018-10-20)

Minor bugfixes and improvements.

- Use measured uncertainties to weigh step detection. (#753)
- Detect also single-commit regressions, if significant. (#745)
- Use proper two-sample test when raw results available. (#754)
- Use a better regression "badness" measure. (#744)
- Display verbose command output immediately, not when command completes. (#747)
- Fix handling of benchmark suite import failures in forkserver and benchmark discovery. (#743, #742)
- Fix forkserver child process handling.
- In asv test suite, use dummy conda packages. (#738)
- Other minor fixes (#756, #750, #749, #746)

7.9 0.3 (2018-09-09)

Major release with several new features.

7.9.1 New Features

- Revised timing benchmarking. `asv` will display and record the median and interquartile ranges of timing measurement results. The information is also used by `asv compare` and `asv continuous` in determining what changes are significant. The `asv run` command has new options for collecting samples. Timing benchmarks have new benchmarking parameters for controlling how timing works, including `processes` attribute for collect data by running benchmarks in different sequential processes. The defaults are adjusted to obtain faster benchmarking. (#707, #698, #695, #689, #683, #665, #652, #575, #503, #493)
- Interleaved benchmark running. Timing benchmarks can be run in interleaved order via `asv run --interleave-processes`, to obtain better sampling over long-time background performance variations. (#697, #694, #647)
- Customization of build/install/uninstall commands. (#699)
- Launching benchmarks via a fork server (on Unix-based systems). Reduces the import time overheads in launching new benchmarks. Default on Linux. (#666, #709, #730)
- Benchmark versioning. Invalidate old benchmark results when benchmarks change, via a benchmark `version` attribute. User-configurable, by default based on source code. (#509)
- Setting benchmark attributes on command line, via `--attribute`. (#647)
- `asv show` command for displaying results on command line. (#711)
- Support for Conda channels. (#539)
- Provide ASV-specific environment variables to launched commands. (#624)

- Show branch/tag names in addition to commit hashes. (#705)
- Support for projects in repository subdirectories. (#611)
- Way to run specific parametrized benchmarks. (#593)
- Group benchmarks in the web benchmark grid (#557)
- Make the web interface URL addresses more copypasteable. (#608, #605, #580)
- Allow customizing benchmark display names (#484)
- Don't reinstall project if it is already installed (#708)

7.9.2 API Changes

- The `goal_time` attribute in timing benchmarks is removed (and now ignored). See documentation on how to tune timing benchmarks now.
- `asv publish` may ask you to run `asv update` once after upgrading, to regenerate `benchmarks.json` if `asv run` was not yet run.
- If you are using `asv` plugins, check their compatibility. The internal APIs in `asv` are not guaranteed to be backward compatible.

7.9.3 Bug Fixes

- Fixes in 0.2.1 and 0.2.2 are also included in 0.3.
- Make `asv compare` accept named commits (#704)
- Fix `asv profile --python=same` (#702)
- Make `asv compare` behave correctly with multiple machines/envs (#687)
- Avoid making too long result file names (#675)
- Fix saving profile data (#680)
- Ignore missing branches during benchmark discovery (#674)
- Perform benchmark discovery only when necessary (#568)
- Fix benchmark skipping to operate on a per-environment basis (#603)
- Allow putting `asv.conf.json` to benchmark suite directory (#717)
- Miscellaneous minor fixes (#735, #734, #733, #729, #728, #727, #726, #723, #721, #719, #718, #716, #715, #714, #713, #706, #701, #691, #688, #684, #682, #660, #634, #615, #600, #573, #556)

7.9.4 Other Changes and Additions

- www: display regressions separately, one per commit (#720)
- Internal changes. (#712, #700, #681, #663, #662, #637, #613, #606, #572)
- CI/etc changes. (#585, #570)
- Added internal debugging command `asv.benchmarks` (#685)
- Make tests not require network connection, except with Conda (#696)
- Drop support for end-of-lifed Python versions 2.6 & 3.2 & 3.3 (#548)

7.10 0.3b1 (2018-08-29)

Prerelease. Same as 0.3rc1, minus #721–

7.11 0.2.2 (2018-07-14)

Bugfix release with minor feature additions.

7.11.1 New Features

- Add a `--no-pull` option to `asv publish` and `asv run` (#592)
- Add a `--rewrite` option to `asv gh-pages` and fix bugs (#578, #529)
- Add a `--html-dir` option to `asv publish` (#545)
- Add a `--yes` option to `asv machine` (#540)
- Enable running via `python -masv` (#538)

7.11.2 Bug Fixes

- Fix support for mercurial >= 4.5 (#643)
- Fix detection of git subrepositories (#642)
- Find conda executable in the "official" way (#646)
- Hide tracebacks in testing functions (#601)
- Launch virtualenv in a more sensible way (#555)
- Disable user site directory also when using conda (#553)
- Set PIP_USER to false when running an executable (#524)
- Set PATH for commands launched inside environments (#541)
- os.environ can only contain bytes on Win/py2 (#528)
- Fix hglib encoding issues on Python 3 (#508)
- Set GIT_CEILING_DIRECTORIES for Git (#636)
- Run pip via python -mpip to avoid shebang limits (#569)
- Always use https URLs (#583)
- Add a min-height on graphs to avoid a flot traceback (#596)
- Escape label html text in plot legends (#614)
- Disable pip build isolation in wheel_cache (#670)
- Fixup CI, test, etc issues (#616, #552, #601, #586, #554, #549, #571, #527, #560, #565)

7.12 0.2.2rc1 (2018-07-09)

Same as 0.2.2, minus #670.

7.13 0.2.1 (2017-06-22)

7.13.1 Bug Fixes

- Use process groups on Windows (#489)
- Sanitize html filenames (#498)
- Fix incorrect date formatting + default sort order in web ui (#504)

7.14 0.2 (2016-10-22)

7.14.1 New Features

- Automatic detection and listing of performance regressions. (#236)
- Support for Windows. (#282)
- New `setup_cache` method. (#277)
- Exclude/include rules in configuration matrix. (#329)
- Command-line option for selecting environments. (#352)
- Possibility to include packages via pip in conda environments. (#373)
- The `pretty_name` attribute can be used to change the display name of benchmarks. (#425)
- Git submodules are supported. (#426)
- The time when benchmarks were run is tracked. (#428)
- New summary web page showing a list of benchmarks. (#437)
- Atom feed for regressions. (#447)
- PyPy support. (#452)

7.14.2 API Changes

- The parent directory of the benchmark suite is no longer inserted into `sys.path`. (#307)
- Repository mirrors are no longer created for local repositories. (#314)
- In asv.conf.json matrix, `null` previously meant (undocumented) the latest version. Now it means that the package is to not be installed. (#329)
- Previously, the `setup` and `teardown` methods were run only once even when the benchmark method was run multiple times, for example due to `repeat > 1` being present in timing benchmarks. This is now changed so that also they are run multiple times. (#316)
- The default branch for Mercurial is now `default`, not `tip`. (#394)

- Benchmark results are now by default ordered by commit, not by date. (#429)
- When `asv run` and other commands are called without specifying revisions, the default values are taken from the branches in `asv.conf.json`. (#430)
- The default value for `--factor` in `asv continuous` and `asv compare` was changed from 2.0 to 1.1 (#469).

7.14.3 Bug Fixes

- Output will display on non-Unicode consoles. (#313, #318, #336)
- Longer default install timeout. (#342)
- Many other bugfixes and minor improvements.

7.15 0.2rc2 (2016-10-17)

Same as 0.2.

7.16 0.1.1 (2015-05-05)

First full release.

7.17 0.1rc3 (2015-05-01)

7.17.1 Bug Fixes

- Display version correctly in docs.
- Include pip_requirements.txt.

7.18 0.1rc2 (2015-05-01)

No significant changes.

7.19 0.1rc1 (2015-05-01)

No significant changes.

CHAPTER

EIGHT

CREDITS

- Michael Droettboom (founder)
- Pauli Virtanen

The rest of the contributors are listed in alphabetical order.

- Aaron Meurer
- @afragner
- Akihiro Nitta
- Andrew Nelson
- Andrew Thomas
- Antoine Pitrou
- Antony Lee
- Ariel Silvio Norberto RAMOS
- Arne Neumann
- Boris Feld
- Chiara Marmo
- Chris Beaumont
- Christoph Deil
- Christopher Whelan
- Colin Carroll
- Daniel Andres Pinto
- David Stansby
- Dieter Werthmüller
- Dorothy Kabarozi
- @DWesl
- Edison Gustavo Muenz
- Elliott Sales de Andrade
- Eric Dill
- Erik M. Bray

- Erik Tollerud
- Erwan Pannier
- Fangchen Li
- Hans Moritz Günther
- Isuru Fernando
- Jarrod Millman
- @jbrockmendel
- jeremie du boisberranger
- John Kirkham
- Josh Soref
- Juan Nunez-Iglesias
- Julian Rüth
- Kacper Kowalik
- Kacper Kowalik (Xarthisius)
- Kevin Anderson
- @Leenkiz
- Lucy Jiménez
- Marc Garcia
- @mariamadronah
- Mark Harfouche
- Markus Mohrhard
- Matthew Treinish
- Matthew Turk
- Matti Picus
- Mike Sarahan
- Min RK
- Nathan Goldbaum
- Nick Crews
- Nicole Franco León
- @pawel
- Paweł Redzyński
- Philippe Pepiot
- Pierre Glaser
- P. L. Lim
- Raphaël Gomès
- Richard Hattersley

- Rohit Goswami
- Rok Mihevc
- Sayed Adel
- serge-sans-paille
- Sourcery AI
- Thomas Pfaff
- Thomas Robitaille
- Tim Felgentreff
- Tom Augspurger
- Tushabe Catherine
- Tyler Reddy
- Valentin Haenel
- @Warbo
- Wojtek Ruszczewski
- Yaroslav Halchenko
- Zach Burnett

BIBLIOGRAPHY

[Friedrich2008] F. Friedrich et al., ''Complexity Penalized M-Estimation: Fast Computation", Journal of Computational and Graphical Statistics 17.1, 201-224 (2008). http://dx.doi.org/10.1198/106186008X285591

[Yao1988] Y.-C. Yao, ''Estimating the number of change-points via Schwarz criterion", Statistics & Probability Letters 6, 181-189 (1988). http://dx.doi.org/10.1016/0167-7152(88)90118-6

PYTHON MODULE INDEX

a

INDEX