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 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.

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](https://asv.readthedocs.io/en/stable/asv.conf.json.html#conf-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. If Conda supports the dependencies you need, that is the recommended method. See [Environments](https://asv.readthedocs.io/en/stable/using.html#environments) for more information.
* matrix: Dependencies you want to preinstall into the environment where benchmarks are run.

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:

**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\_iterkeys(self):

**for** key **in** self.d.iterkeys():

**pass**

**def** time\_range(self):

d = self.d

**for** key **in** range(500):

x = d[key]

**def** time\_xrange(self):

d = self.d

**for** key **in** xrange(500):

x = d[key]

**class** **MemSuite**:

**def** mem\_list(self):

**return** [0] \* 256

You’ll want to replace these benchmarks with your own. See [Writing benchmarks](https://asv.readthedocs.io/en/stable/writing_benchmarks.html#writing-benchmarks) for more information.

Running benchmarks

Benchmarks are run using the asv run subcommand.

Let’s start by just benchmarking the latest commit on the current master branch of the project:

$ asv run

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.

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](https://store.continuum.io/cshop/anaconda/) and [virtualenv](https://pypi.python.org/pypi/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 each of the Python versions specified in the asv.conf.json file available on the PATH. For example, if the asv.conf.json file has:

"pythons": ["2.7", "3.6"]

then it will use the executables named python2.7 and python3.6 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](https://github.com/yyuu/pyenv).

The virtualenv environment also supports [PyPy](http://pypy.org/). 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.

Benchmarking

Finally, the benchmarks are run:

$ asv run

· Cloning project.

· Fetching recent changes

· Creating environments......

· Discovering benchmarks

·· Uninstalling from virtualenv-py2.7

·· Building 4238c44d <master> for virtualenv-py2.7

·· Installing into virtualenv-py2.7.

· Running 10 total benchmarks (1 commits \* 2 environments \* 5 benchmarks)

[ 0.00%] · For project commit 4238c44d <master>:

[ 0.00%] ·· Building for virtualenv-py2.7.

[ 0.00%] ·· Benchmarking virtualenv-py2.7

[ 10.00%] ··· Running (benchmarks.TimeSuite.time\_iterkeys--)....

[ 30.00%] ··· benchmarks.MemSuite.mem\_list 2.42k

[ 35.00%] ··· benchmarks.TimeSuite.time\_iterkeys 11.1±0.01μs

[ 40.00%] ··· benchmarks.TimeSuite.time\_keys 11.2±0.01μs

[ 45.00%] ··· benchmarks.TimeSuite.time\_range 32.9±0.01μs

[ 50.00%] ··· benchmarks.TimeSuite.time\_xrange 30.3±0.01μs

[ 50.00%] ·· Building for virtualenv-py3.6..

[ 50.00%] ·· Benchmarking virtualenv-py3.6

[ 60.00%] ··· Running (benchmarks.TimeSuite.time\_iterkeys--)....

[ 80.00%] ··· benchmarks.MemSuite.mem\_list 2.11k

[ 85.00%] ··· benchmarks.TimeSuite.time\_iterkeys failed

[ 90.00%] ··· benchmarks.TimeSuite.time\_keys 9.07±0.5μs

[ 95.00%] ··· benchmarks.TimeSuite.time\_range 35.5±0.01μs

[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](https://asv.readthedocs.io/en/stable/tuning.html).

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](https://www.kernel.org/pub/software/scm/git/docs/gitrevisions.html), or the [revsets help section](http://www.selenic.com/hg/help/revsets) for Mercurial.

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

asv run master..mybranch

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

asv run v0.1..master

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**

There is a special version of asv run that is useful when developing benchmarks, called asv dev. See [Writing benchmarks](https://asv.readthedocs.io/en/stable/writing_benchmarks.html#writing-benchmarks) for more information.

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](https://asv.readthedocs.io/en/stable/using.html#viewing-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.

Viewing the results

You can use the [asv show](https://asv.readthedocs.io/en/stable/commands.html#cmd-asv-show) command to display results from previous runs on the command line:

$ asv show master

Commit: 4238c44d <master>

benchmarks.MemSuite.mem\_list [mymachine/virtualenv-py2.7]

2.42k

started: 2018-08-19 18:46:47, duration: 1.00s

benchmarks.TimeSuite.time\_iterkeys [mymachine/virtualenv-py2.7]

11.1±0.06μs

started: 2018-08-19 18:46:47, duration: 1.00s

...

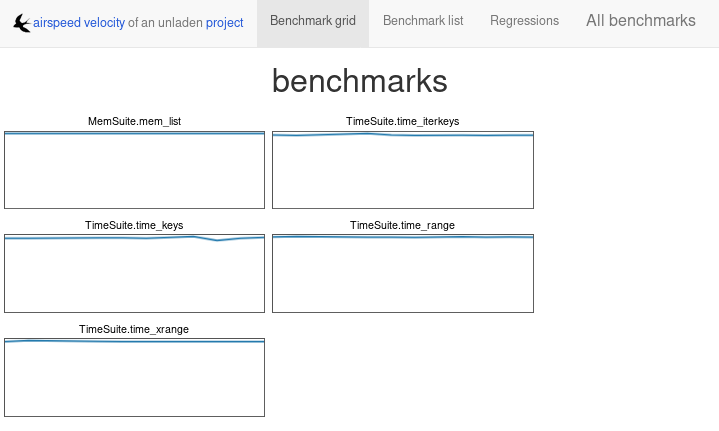
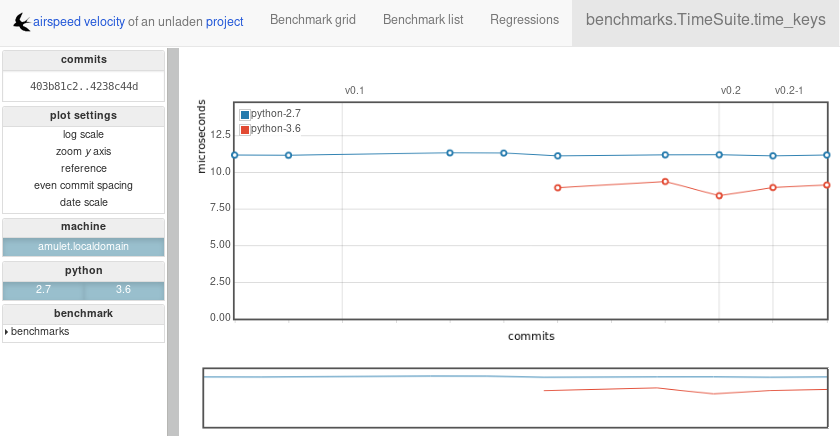
To collate a set of results into a viewable website, run:

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.

[](https://asv.readthedocs.io/en/stable/_images/screenshot-grid.png) [](https://asv.readthedocs.io/en/stable/_images/screenshot-bench.png)

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](https://asv.readthedocs.io/en/stable/commands.html#cmd-asv-gh-pages) for details.

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**](https://docs.python.org/3/library/fnmatch.html#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.

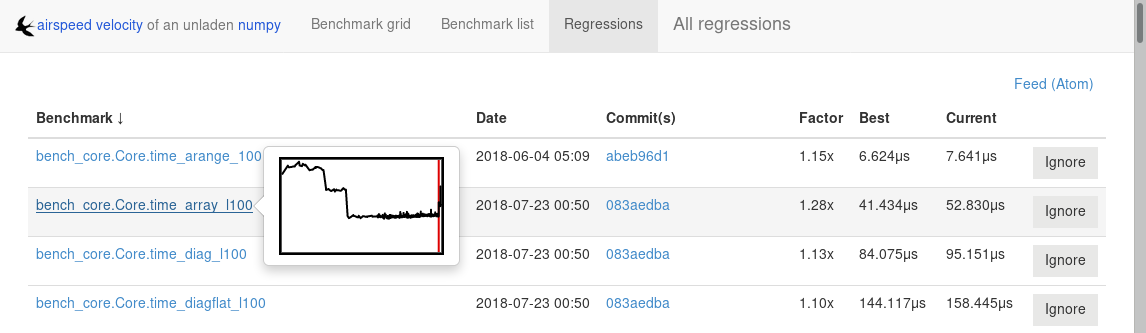
The asv rm command will prompt before performing any operations. Passing the -y option will skip the prompt.

Here is a more complex example, to remove all of the benchmarks on Python 2.7 and the machine named giraffe:

asv rm python=2.7 machine=giraffe

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.

[](https://asv.readthedocs.io/en/stable/_images/screenshot-regressions.png)

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 <----------------------------O----------------------------->

- Testing <-------------O-------------->------------------------------

- Testing --------------<-------O------>------------------------------

- Testing --------------<---O--->-------------------------------------

- Testing --------------<-O->-----------------------------------------

- Testing --------------<O>-------------------------------------------

- 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.

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**](https://docs.python.org/3/library/profile.html#module-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**](https://docs.python.org/3/library/profile.html#module-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/python2.7/timeit.py:201(repeat)

3 0.000 0.000 6.844 2.281 /usr/lib64/python2.7/timeit.py:178(timeit)

3 0.104 0.035 6.844 2.281 /usr/lib64/python2.7/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/python2.7/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/python2.7/timeit.py:143(setup)

1 0.000 0.000 0.000 0.000 /usr/lib64/python2.7/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/python2.7/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](http://www.vrplumber.com/programming/runsnakerun/) or [snakeviz](http://jiffyclub.github.io/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.

See [asv profile](https://asv.readthedocs.io/en/stable/commands.html#cmd-asv-profile) for more options.

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-py2.7-numpy]

failed 35.2m n/a benchmarks.MemSuite.mem\_list [amulet.localdomain/virtualenv-py3.6-numpy]

11.5±0.08μs 11.0±0μs 0.96 benchmarks.TimeSuite.time\_iterkeys [amulet.localdomain/virtualenv-py2.7-numpy]

failed failed n/a benchmarks.TimeSuite.time\_iterkeys [amulet.localdomain/virtualenv-py3.6-numpy]

11.5±1μs 11.2±0.02μs 0.97 benchmarks.TimeSuite.time\_keys [amulet.localdomain/virtualenv-py2.7-numpy]

failed 8.40±0.02μs n/a benchmarks.TimeSuite.time\_keys [amulet.localdomain/virtualenv-py3.6-numpy]

34.6±0.09μs 32.9±0.01μs 0.95 benchmarks.TimeSuite.time\_range [amulet.localdomain/virtualenv-py2.7-numpy]

failed 35.6±0.05μs n/a benchmarks.TimeSuite.time\_range [amulet.localdomain/virtualenv-py3.6-numpy]

31.6±0.1μs 30.2±0.02μs 0.95 benchmarks.TimeSuite.time\_xrange [amulet.localdomain/virtualenv-py2.7-numpy]

failed failed n/a benchmarks.TimeSuite.time\_xrange [amulet.localdomain/virtualenv-py3.6-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](https://asv.readthedocs.io/en/stable/commands.html#cmd-asv-compare) for more.

Writing 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**

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, there is a special command, asv dev, that uses all of these features and is equivalent to:

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

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

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

**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.

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.

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](https://asv.readthedocs.io/en/stable/benchmarks.html).

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.

Benchmark types

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](https://asv.readthedocs.io/en/stable/benchmarks.html).

The default timing function is [**timeit.default\_timer**](https://docs.python.org/3/library/timeit.html#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**](https://docs.python.org/3/library/time.html#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**](https://docs.python.org/3/library/time.html#time.process_time) to measure process time.

**Note**

One consequence of using [**time.process\_time**](https://docs.python.org/3/library/time.html#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):

**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](https://asv.readthedocs.io/en/stable/benchmarks.html).

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](http://pythonhosted.org/Pympler/asizeof.html) 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)](https://asv.readthedocs.io/en/stable/writing_benchmarks.html#tracking) benchmark can be used instead.

For details, see [Benchmark types and attributes](https://asv.readthedocs.io/en/stable/benchmarks.html).

**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.

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](https://asv.readthedocs.io/en/stable/benchmarks.html).

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](https://hg.python.org/cpython/file/tip/Modules/Setup.dist) or brought in by importing the timeit module (which further imports gc, sys, time, and itertools).

Imports

You can use raw timing benchmarks to measure import times.

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](https://asv.readthedocs.io/en/stable/benchmarks.html).

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](https://asv.readthedocs.io/en/stable/benchmarks.html) for reference documentation.

# 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.

## 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

## 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](https://pyperf.readthedocs.io/) project has [documentation on how to tune machines for benchmarking](https://pyperf.readthedocs.io/en/latest/system.html). 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](https://asv.readthedocs.io/en/stable/commands.html#cmd-asv-run).

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.