airspeed velocity Documentation

Release 0.1.1

Michael Droettboom

May 26, 2019
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installing airspeed velocity</td>
<td>3</td>
</tr>
<tr>
<td>Using airspeed velocity</td>
<td>5</td>
</tr>
<tr>
<td>Writing benchmarks</td>
<td>15</td>
</tr>
<tr>
<td>Reference</td>
<td>21</td>
</tr>
<tr>
<td>Developer Docs</td>
<td>31</td>
</tr>
<tr>
<td>Changelog</td>
<td>35</td>
</tr>
<tr>
<td>Credits</td>
<td>37</td>
</tr>
<tr>
<td>Python Module Index</td>
<td>39</td>
</tr>
</tbody>
</table>
airspeed velocity \((\texttt{asv})\) 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 an example airspeed velocity site.

License: BSD three-clause license.

| Warning: | airspeed velocity is pre-release software. It is most certainly going to change without notice, and may eat kittens. |
Installing airspeed velocity

airspeed velocity is known to work on Linux and Mac OS-X. It’s highly unlikely that it works on Microsoft Windows. It is known to work with Python 2.6, 2.7, 3.2, 3.3 and 3.4.

airspeed velocity is a standard Python package, with its installation based on setuptools, and can be installed using:

```
python setup.py install
```

The requirements should be automatically installed. If they aren’t installed automatically, for example due to networking restrictions, the requirements are:

- six, 1.4 or later

One of the following:

- virtualenv, 1.10 or later (this is true even with Python 3.3, where virtualenv is included as venv, since venv is not compatible with other versions of Python).

  Note that virtualenv 1.11.0 will not work, as it contains a bug in setuptools that prevents its installation in a clean virtual environment.

- An anaconda or miniconda installation, with the conda command available on your path.

**Note:** Anaconda or miniconda is preferred if the dependencies of your project involve 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, these dependencies will 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 py.test. 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:

```
python setup.py test
```
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. Importantly, the result data stored alongside the benchmark suite may grow quite large, which is a good reason to not include it in the main project repository.

The user interacts 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
Edit asv.conf.json to get started.
```

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 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/spacetelescope/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 for more information.

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.

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:

```python
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]
```

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 master 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 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.

conda is a recommended method if it contains the dependencies your project needs, because it is faster and in many cases will not have to compile the dependencies from scratch.

2.2. Running benchmarks

7
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", "3.3"]
```

then it will use the executables named `python2.7` and `python3.3` 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`.

### 2.2.3 Benchmarking

Finally, the benchmarks are run:

```
$ asv run
  · Cloning project.
  · Fetching recent changes..
  · Creating environments
    · Creating conda environment for py2.7
    · Creating conda environment for py3.4
  · Installing dependencies..
  · Discovering benchmarks
    · Creating conda environment for py2.7
    · Uninstalling project from py2.7
    · Installing project into py2.7.
    [ 0.00%] · For project commit hash ac71c70d:
    [ 0.00%]   · Building for py2.7
    [ 0.00%]   · Uninstalling project from py2.7
    [ 0.00%]   · Installing project into py2.7.
    [ 0.00%]   · Benchmarking py2.7
    [ 10.00%]   · Running benchmarks.MemSuite.mem_list 2.4k
    [ 20.00%]   · Running benchmarks.TimeSuite.time_iterkeys 9.27μs
    [ 30.00%]   · Running benchmarks.TimeSuite.time_keys 10.74μs
    [ 40.00%]   · Running benchmarks.TimeSuite.time_range 42.20μs
    [ 50.00%]   · Running benchmarks.TimeSuite.time_xrange 32.94μs
    [ 50.00%]   · Building for py3.4
    [ 50.00%]   · Uninstalling project from py3.4
    [ 50.00%]   · Installing project into py3.4.
    [ 50.00%]   · Benchmarking py3.4
    [ 60.00%]   · Running benchmarks.MemSuite.mem_list 2.4k
    [ 70.00%]   · Running benchmarks.TimeSuite.time_iterkeys failed
    [ 80.00%]   · Running benchmarks.TimeSuite.time_keys 7.29μs
    [ 90.00%]   · Running benchmarks.TimeSuite.time_range 30.41μs
    [100.00%]   · Running benchmarks.TimeSuite.time_xrange failed
```

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

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 the branch was created:

```
asv run mybranch@{u}..mybranch
```

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

```
asv run v0.1..master
```
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
```

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

```
asv run MISSING
```

Note: There is a special version of `asv run` that is useful when developing benchmarks, called `asv dev`. See `Writing benchmarks` for more information.

The results are stored as a tree of 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, the normal workflow is to commit these results to a source code repository 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

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 these files on any webserver that can serve static content. Github Pages works quite well, for example. If using Github Pages, asv includes the convenience command `asv gh-pages` to automatically publish the results to the `gh-pages` branch.

### 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
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. Note that generally the results will be stored in a source code repository, so it should be possible to undo any of the changes using the DVCS directly as well.

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
```

### 2.5 Finding a commit that produces a large regression

Since benchmarking can be rather time consuming, it’s likely that you are 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 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 command line 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 --------------<>-------------------------------------
- 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 preview 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      0.104    0.035    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 {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}
300000  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)
```
(continues on next page)
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`.

## 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 7810d6d7 19aa5743
· Fetching recent changes.

All benchmarks:

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th>after</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7810d6d7]</td>
<td>[19aa5743]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
+ 1.75ms  | 152.84ms | 87.28  | time_quantity.time_quantity_array_conversion |
+ 933.71μs | 108.22ms | 115.90 | time_quantity.time_quantity_init_array |
83.65μs  | 55.38μs  | 0.66   | time_quantity.time_quantity_init_scalar |
281.71μs | 146.88μs | 0.52   | time_quantity.time_quantity_scalar_conversion |
+ 1.31ms  | 7.75ms   | 5.91   | time_quantity.time_quantity_ufunc_sin |
5.73ms   | 5.73ms   | 1.00   | time_units.mem_unit |
```

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 2 (that is, benchmarks that improve by more than a factor of 2 or worsen by a factor of 2 are color coded). The threshold can be set with the `--threshold=value` option. Finally, the benchmarks can be split into ones that have improved, stayed the same, and worsened, using the same threshold.
Benchmarks are stored in a collection of .py files in the benchmark suite’s benchmark directory (as defined by benchmark_dir in the asv.conf.json file). They may be arbitrarily nested in subdirectories, and all .py files will be used, regardless of their file name.

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:

```python
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:

```python
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, 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
```

## 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 set an attribute called `setup` to a free function. For example:

```python
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 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 = 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. Generally, however, it is not required: each benchmark runs in its own process, so any tearing down of in-memory state happens automatically.

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

```python
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
```

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.

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:

```python
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:

```python
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.

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}[f]
    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”.

3.5 Benchmark types

3.5.1 Timing

Timing benchmarks have the prefix time.

The timing itself is based on the Python standard library’s timeit module, with some extensions for automatic heuristics shamelessly stolen from IPython’s %timeit magic function. This means that in most cases the benchmark function itself will be run many times to achieve accurate timing.

The default timing function is the POSIX CLOCK_PROCESS_CPUTIME, which measures the CPU time used only by the current process. This is available as time.process_time in Python 3.3 and later, but a backport is included with asv for earlier versions of Python.

Note: One consequence of using CLOCK_PROCESS_CPUTIME is that the time spent in child processes of the benchmark is not included. If your benchmark spawns other processes, you may get more accurate results by setting the timer attribute on the benchmark to timeit.default_timer.

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()

Attributes:

- goal_time: asv will automatically select the number of iterations to run the benchmark so that it takes between goal_time / 10 and goal_time seconds each time. If not specified, goal_time defaults to 2 seconds.
- number: Manually choose the number of iterations. If number is specified, goal_time is ignored.
• **repeat**: The number of times to repeat the benchmark, with each repetition running the benchmark number of times. The minimum time from all of these repetitions is used as the final result. When not provided, defaults to `timeit.default_repeat(3)`.

• **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 `time.process_time` (or a backported version of it for versions of Python prior to 3.3), but other useful values are `timeit.default_timer` to use the default `timeit` behavior on your version of Python.

On Windows, `time.clock` has microsecond granularity, but `time.time`'s granularity is 1/60th of a second. On Unix, `time.clock` has 1/100th of a second granularity, and `time.time` is much more precise. On either platform, `timeit.default_timer` measures wall clock time, not the CPU time. This means that other processes running on the same computer may interfere with the timing. That’s why the default of `time.process_time`, which only measures the time used by the current process, is often the best choice.

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

### 3.5.2 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:

```python
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.

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

### 3.5.3 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:

```python
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 spawn a separate subprocess for executing memory-intensive setup tasks.
3.5.4 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:

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

Attributes:

- `unit`: The unit of the values returned by the benchmark. Used for display in the web interface.
4.1 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.

4.1.1 project

The name of the project being benchmarked.

4.1.2 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.

4.1.3 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:spacetelescope/asv.git
- https://github.com/spacetelescope/asv.git
- ssh://hg@bitbucket.org/yt_analysis/yt
- hg+https://bitbucket.org/yt_analysis/yt
The repository may be readonly.

Note: Currently, mercurial works only on Python 2, although the interface to Mercurial used in asv (python-hglib) is being ported to Python 3. At the present time, Mercurial support will only function on Python 2.

4.1.4 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, “master” (Git) or “tip” (Mercurial) is chosen.

4.1.5 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/

4.1.6 pythons

The versions of Python to run the benchmarks in. If not provided, it will default to the version of Python that the asv command (master) 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. "2.7", in which case:
  – if conda is found, conda will be used to create an environment for that version of Python
  – 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.

4.1.7 matrix

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

If provided, it must be a dictionary, where the keys are the names of dependencies and the values are lists of versions (as strings) of that dependency. If the list is empty, use the “latest” version of that dependency available on PyPI.

For example, the following will test with two different versions of Numpy, and the latest version of Cython:
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 is rather limited, as it only supports dependencies that are installable from PyPI using `pip`, and there is no functionality for limiting the matrix to specific combinations.

### 4.1.8 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".

### 4.1.9 environment_type

Specifies the tool to use to create environments. May be “conda”, “virtualenv” 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.

### 4.1.10 env_dir

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

### 4.1.11 results_dir

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

### 4.1.12 html_dir

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

### 4.1.13 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.

### 4.1.14 plugins

A list of modules to import containing asv plugins.

### 4.1.14 wheel_cache_size

The number of wheels (builds) to cache for each environment.
4.2 Commands

4.2.1 asv help

usage: asv help [-h]

optional arguments:
  -h, --help  show this help message and exit

4.2.2 asv quickstart

usage: asv quickstart [-h] [--dest DEST]

Creates a new benchmarking suite

optional arguments:
  -h, --help  show this help message and exit
  --dest DEST, -d DEST  The destination directory for the new benchmarking suite

4.2.3 asv machine


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

optional arguments:
  -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)'.
  --ram RAM  The amount of physical RAM on this machine. For example, '4GB'.

4.2.4 asv setup

usage: asv setup [-h] [--parallel [PARALLEL]]
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.

optional arguments:
- `-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.

4.2.5 asv run


Run a benchmark suite.

positional arguments:
- `range` Range of commits to benchmark. 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. Also accepts the special values 'NEW', 'ALL', and 'EXISTING'. '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. By default, will benchmark the head of the current master branch.

optional arguments:
- `-h, --help` show this help message and exit
- `--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.
- `--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.
- `--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.
--python PYTHON Specify a Python interpreter in which to run the benchmarks. By default, uses the same Python
interpreter that asv is using. It may be an executable
to be searched for on the $PATH, an absolute path, or
the special value "same" which will use the same
Python interpreter that asv is using. This "same"
interpreter must have the benchmarked project already
installed, including its dependencies. A specific
revision may not be provided when --python is
provided. It may also be any string accepted by any of
the environment plugins. For example, the conda plugin
accepts "2.7" to mean create a new Conda environment
with Python version 2.7.

--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

#### 4.2.6 asv dev

usage: asv dev [-h] [--bench BENCH] [--machine MACHINE] [--python PYTHON]

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

optional arguments:
  -h, --help    show this help message and exit
  --bench BENCH, -b BENCH
    Regular expression(s) for benchmark to run. When not
    provided, all benchmarks are run.

  --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.

  --python PYTHON
    Specify a Python interpreter in which to run the benchmarks. By default, uses the same Python
    interpreter that asv is using. It may be an executable
to be searched for on the $PATH, an absolute path, or
    the special value "same" which will use the same
    Python interpreter that asv is using. This "same"
    interpreter must have the benchmarked project already
    installed, including its dependencies. A specific
    revision may not be provided when --python is
    provided. It may also be any string accepted by any of
    the environment plugins. For example, the conda plugin
    accepts "2.7" to mean create a new Conda environment
    with Python version 2.7.
installed, including its dependencies. A specific revision may not be provided when --python is provided. It may also be any string accepted by any of the environment plugins. For example, the conda plugin accepts "2.7" to mean create a new Conda environment with Python version 2.7.

### 4.2.7 asv continuous

**usage:** asv continuous [-h] [--factor FACTOR] [--show-stderr] [--bench BENCH] [--machine MACHINE] [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 master branch.

**optional arguments:**
- **-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 2, if a benchmark gets twice as slow or twice as fast, it will be displayed in the results list.
- **--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.
- **--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.

### 4.2.8 asv find

**usage:** asv find [-h] [--invert] [--show-stderr] [--machine MACHINE] 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.

optional arguments:
- `--invert`, `-i`  Search for a decrease in the benchmark value, rather than an increase.
- `--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.

### 4.2.9 asv rm

usage: asv rm [-h] [-y] 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.

optional arguments:
- `--help`, `-h`  show this help message and exit
- `--y`, `-y`  Don't prompt for confirmation.

### 4.2.10 asv publish

usage: asv publish [-h]

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.

optional arguments:
- `--help`, `-h`  show this help message and exit

### 4.2.11 asv preview

usage: asv preview [-h] [--port PORT] [--browser]

Preview the results using a local web server.

optional arguments:
- `--help`, `-h`  show this help message and exit
- `--port PORT`, `-p PORT`  Port to run webserver on. [8080]
- `--browser`, `-b`  Open in webbrowser
4.2.12 asv profile

                  [--environment ENVIRONMENT] [--python PYTHON]
                  benchmark [revision]

Profile a benchmark

positional arguments:
  benchmark               The benchmark to profile. Must be a fully-specified
                          benchmark name.
  revision                The revision of the project to profile. May be a
                          commit hash, or a tag or branch name.

optional arguments:
  -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.
  --environment ENVIRONMENT, -e ENVIRONMENT
                          Which environment to use. Your benchmarking project
                          may have multiple environments if it has a dependency
                          matrix or multiple versions of Python specified. This
                          should the name of an environment directory as already
                          created by the run command. If `None` is specified,
                          one will be chosen at random.
  --python PYTHON         Specify a Python interpreter in which to run the
                          benchmarks. By default, uses the same Python
                          interpreter that asv is using. It may be an executable
                          to be searched for on the $PATH, an absolute path, or
                          the special value "same" which will use the same
                          Python interpreter that asv is using. This "same"
                          interpreter must have the benchmarked project already
                          installed, including its dependencies. A specific
                          revision may not be provided when --python is
                          provided. It may also be any string accepted by any of
                          the environmentplugins. For example, the conda plugin
                          accepts "2.7" to mean create a new Conda environment
                          with Python version 2.7.

4.2.13 asv update

usage: asv update [-h]

Update the results and config files to the current version

optional arguments:
  -h, --help              show this help message and exit
4.2.14 asv compare

usage: asv compare [-h] [--factor FACTOR] [--split] [--machine MACHINE]
                  revision1 revision2

Compare two sets of results

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

optional arguments:
  -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 2, if a
               benchmark gets twice as slow or twice as fast, it will
               be displayed in the results list.
  --split, -s
               Split the output into a table of benchmarks that have
               improved, stayed the same, and gotten worse.
  --machine MACHINE, -m MACHINE
               The machine to compare the revisions for.

4.2.15 asv gh-pages

usage: asv gh-pages [-h]

Publish the results to github pages

optional arguments:
  -h, --help  show this help message and exit
This section describes some things that may be of interest to developers of `asv`.

## 5.1 Benchmark suite layout

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.
- `$benchmark_dir`: Contains the benchmark code, created by the user. Each subdirectory needs an `__init__.py`.
- `$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 and dependencies 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 `distutils python setup.py build` command. This means repeated builds happen in the same place and `cache` is able to cache and reuse many of the build products.

* wheels/: If `wheel_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.

  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. Useful keys include:

    · commit_hash: The project commit that the benchmarks were run on.
    · 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.
    · results: A dictionary from benchmark names to benchmark results.

      If non-parameterized benchmark, the result is a single value.

      For parameterized benchmarks, the result is a dictionary with keys `params` and `result`. The `params` value 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.

      In the results, `null` indicates a failed benchmark, including failures in installing the project version. `NaN` indicates a benchmark explicitly skipped by the benchmark suite.

* `$html_dir/`: The output of `asv publish`, that turns the raw results in `$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.
  * **date_to_hash**: A dictionary mapping Javascript date stamps to commit hashes. 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 dates, 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.

## 5.2 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:

```
python setup.py test -a "--webdriver=PhantomJS"
p y.test --webdriver=PhantomJS
```

The allowed values include PhantomJS (default) and Chrome, corresponding to:

- **PhantomJS**: Headless web browser. Runs without requiring a display. On Ubuntu, install via `apt-get install phantomjs`.
- **ChromeDriver**: Chrome-based controllable browser. Cannot run without a display, and will pop up a window when running. On Ubuntu, install via `apt-get install chromium-chromedriver`.

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

5.2. Full-stack testing
6.1 0.1.1 (2015-05-05)

6.2 0.1rc3 (2015-05-01)

6.2.1 Bug Fixes

Include pip_requirements.txt.
Display version correctly in docs.

6.3 0.1rc2 (2015-05-01)

6.4 0.1rc1 (2015-05-01)
CHAPTER 7

Credits

Michael Droettboom would like to thank the following contributors to asv:

• Pauli Virtanen
• Thomas Robitaille
• Richard Hattersley
• Matthew Turk
• Antoine Pitrou
• Christoph Deil
• Chris Beaumont
• Kacper Kowalik
• Erik M. Bray
• Nathan Goldbaum
• MinRK
• Christopher Whelan
a

asv.commands, 24
A

asv.commands (module), 24