Caliper Documentation

Release 0.0.18

Vanessa Sochat

Jan 22, 2021

GETTING STARTED

Caliper is a tool for measuring and assessing change in packages. To see the code, head over to the repository

CHAPTER

ONE

GETTING STARTED WITH CALIPER

Caliper can be installed from pypi or directly from the repository. See *Installation* for installation, and then the *Getting Started* section for using caliper.

CHAPTER

TWO

SUPPORT

- For **bugs and feature requests**, please use the issue tracker.
- For **contributions**, visit Caliper on Github.

CHAPTER

THREE

RESOURCES

Caliper Analysis An example analysis of tensorflow using Caliper.

Caliper Metrics A small flat file (json) database of metrics extracted for pypi and GitHub packages

3.1 Getting Started

Caliper is a tool for measuring and assessing change in packages. This means that we can extract metrics across releases of packages, and then test any number of scripts against those releases. Caliper can help you to answer questions like:

- **How informative is a semantic version change?** For example, we could measure changes in code (lines, imports, etc.) between different releases, and we would expect major version changes to be larger than minor.
- **When will it break?** If we have a scientific script without a requirements.txt file (for Python) we can test running it across versions of a major dependency to assess which will work.
- **How correct are the versions we specify?** If we are creating entire solvers around trying to resolve a list of dependency versions, it better be the case that our specifications are accurate! By comparing changes between versions and then measuring when a script breaks, we can better understand how flexible these versions really are.

If you have any questions or issues, please let us know

3.1.1 Installation

Caliper can be installed from pypi, or from source. For either, it's recommended that you create a virtual environment, if you have not already done so.

Virtual Environment

First, clone the repository code.

```
$ git clone git@github.com:vsoch/caliper.git
$ cd caliper
```

Then you'll want to create a new virtual environment, and install dependencies.

```
$ python -m venv env
$ source env/bin/activate
$ pip install -r requirements.txt
```

And install Caliper (from the repository directly)

```
$ pip install -e .
```

Install via pip

Caliper can also be installed with pip.

```
$ pip install caliper
```

Once it's installed, you should be able to inspect the client!

```
$ caliper --help
usage: caliper [-h] [--version] [--quiet] [--verbose] [--log-disable-color] [--log-
↪use-threads]
               {version,metrics,analyze,extract,view} ...

Caliper is a tool for measuring and assessing changes in packages.

optional arguments:
  -h, --help            show this help message and exit
  --version             suppress additional output.

actions:
  actions

  {version,metrics,analyze,extract,view}
                        actions
    version             show software version
    metrics             see metrics available
    analyze             analyze functionality of a package.
    extract             extract one or more metrics for a software package.
    view                extract a metric and view a plot.

LOGGING:
  --quiet               suppress logging.
  --verbose             verbose output for logging.
  --log-disable-color   Disable color for caliper logging.
  --log-use-threads     Force threads rather than processes.
```

3.1.2 User Guide

Caliper is a tool for measuring and assessing change in packages. It's currently of interest if you want to study how developers create software, or some kind of change that might predict something breaking. It might eventually be useful as a tool to, given a script with dependencies speciied, know the kind of flexibility you have to honor those dependencies (or not). This could be hugely useful for package managers and solvers that need to try more flexible constraints given a conflict.

Concepts

- **Manager**: a handle to interact with a package manager
- **Extractor**: a controller to use a manager to extract metrics of interest
- **Analysis**: A caliper analysis means attempting to build containers across versions of a library, and run against scripts for tests to assess functionality.
- **Version repository**: a repository created by an extractor that tagged commits for package releases
- **Metrics**: are a type of classes that can extract a single timepoint, or a change over time (e.g., lines changed). You can see example metrics that have been extracted under in the vsoch/caliper-metrics

Managers

A manager is a handle to interact with a package.

Pypi

The first kind of package we are interested in is one from pypi. We might quickly extract all metrics to an output folder from the command line for a Pypi package:

```
caliper extract --outdir caliper-metrics/ pypi:sregistry
```

Pypi Details

or we can instantiate a manager from Python, and walk through the steps that the client takes. First we create the manager.

```
from caliper.managers import PypiManager
manager = PypiManager("sregistry")
```

The manager specs include the source archive, version, and hash for each version of the package. The schema of the spec is a subset of the spack package schema. Every manager exposes this metadata.

```
manager.specs[0]
Found 82 versions for sregistry
{'name': 'sregistry',
 'version': '0.0.1',
 'source': {'filename': 'https://files.pythonhosted.org/packages/ef/2f/
↪ccc36e816dc081abbe0932c422586eda868719025ec07ac206ed254d6a3c/sregistry-0.0.1.tar.gz
↪',
  'type': 'source'},
 'hash': 'd4ee6933321b5a3da13e0b1657ca74f90477f670e59096a6a0a4dbb30a0b1f07'}

manager.specs[-1]
{'name': 'sregistry',
 'version': '0.2.36',
 'source': {'filename': 'https://files.pythonhosted.org/packages/75/6c/
↪2b5bcf0191c0ddc9b95dd156d827c8d80fa8fe86f01f7a053fdd97eaea41/sregistry-0.2.36.tar.gz
↪',
  'type': 'source'},
 'hash': '238ebd3ca0e0408e0be6780d45deca79583ce99aed05ac6981da7a2b375ae79e'}
```

If you just interact with *manager.specs*, you'll get a random architecture for each one. This can be okay if you want to do static file analysis, but if you want to choose a specific python version, your best bet is to call the get package metadata function directly and provide your preferences. For example, here we want Tensorflow for Python 3.5 and a specific linux architecture:

```
manager.get_package_metadata(python_version="35", arch="manylinux1_x86_64")
```

To derive these search strings, you can look at examples of wheels provided. This isn't the default because not all packages provide such rich choices. Here is an example from an early version of tensorflow.

```
tensorflow-0.12.0-cp27-cp27m-macosx_10_11_x86_64.whl
tensorflow-0.12.0-cp27-cp27mu-manylinux1_x86_64.whl
tensorflow-0.12.0-cp34-cp34m-manylinux1_x86_64.whl
tensorflow-0.12.0-cp35-cp35m-macosx_10_11_x86_64.whl
tensorflow-0.12.0-cp35-cp35m-manylinux1_x86_64.whl
tensorflow-0.12.0-cp35-cp35m-win_amd64.whl
```

For more recent versions you would see Python 3.8 and 3.9, and definitely not 2.x. The above function still selects one release based on your preferences. You can also choose to return a subset of _all_ versions with the filter function. For example, here let's narrow down the set to include those that can be installed on Linux.

```
releases = manager.filter_releases('manylinux1_x86_64')
```

You can also get a set of unique Python versions across packages:

```
python_versions = manager.get_python_versions()
# {'cp27', 'cp33', 'cp34', 'cp35', 'cp36', 'cp37', 'cp38'}
```

Not all package versions are guaranteed to have these Python versions, but that's something interesting to consider. And you can always interact with the raw package metadata at *manager.metadata*.

GitHub

We might also be interested in releases from GitHub. Extracting metrics from the command line would look like this:

```
caliper extract --outdir caliper-metrics/ github:vsoch/pull-request-action
```

GitHub Details

And we could do the same steps as above (as we did with the [pypi manager](#pypi-manager) to create an interactive manager client.

```
from caliper.managers import GitHubManager
manager = GitHubManager("vsoch/pull-request-action")
```

GitManager

A GitManager is a special kind of manager that exists to interact with a git repository. It will be possible to use it as a manager proper (not yet developed) but it can also serve to create and interact with local git repositories. For example, let's create a temporary directory, add stuff to it, commit and then tag it.

```
from caliper.managers import GitManager
import tempfile
git = GitManager(tempfile.mkdtemp())
git.init()

# write some content (file.txt)

git.add("file.txt")
git.commit("Adding new content!")
git.tag("tag")
```

Note that when you run `git.init()` a dummy username and email will be added to the `.git/config` file so we can continue interactions without needing a global setting. This is done intentionally based on the idea that the user likely won't keep the version repository, however if you do want to keep it, feel free to change or remote these settings in favor of global ones.

You can imagine how this might be used - we can have a class that can take a manager, and then iterate over versions/releases and create a tagged commit for each. We can then easily extract metrics about files changed between versions. This is the metrics extractor discussed later.

Caliper Analyze

Caliper supports analyzing package functionality, which means that we take a configuration file, a `caliper.yaml` with a package name, manager, Dockerfile template to build, and a list of tests. We do this with the Caliper *analyze* command:

```
$ caliper analyze --help
usage: caliper analyze [-h] [--config CONFIG] [--no-progress] [--serial] [--force] [--
↪nprocs NPROCS]

optional arguments:
  -h, --help       show this help message and exit
  --config CONFIG  A caliper.yaml file to use for the analysis (required)
  --no-progress    Do not show a progress bar (defaults to unset, showing progress)
  --serial         Run in serial instead of parallel
  --force          If an output file exists, force re-write (default will not␣
↪overwrite)
  --nprocs NPROCS  Number of processes. Defaults to cpu count.
```

For example, we might use the example and do:

```
$ caliper analyze --config examples/analyze/caliper.yaml
```

to do a `docker system prune --all` after each build (recommended) add `--cleanup`

```
$ caliper analyze --config examples/analyze/caliper.yaml --cleanup
```

And if your caliper.yaml is in the same folder as you are running caliper from, you don't need to supply it (it will be auto-detected):

```
caliper analyze --cleanup
```

and run the builds in serial. A parallel argument is supported, but in practice it doesn't work well building multiple containers at once.

Analyze Output

The output of analyze will be to write results to a `.caliper` folder, specifically to `.caliper/data`. The result files can then be parsed to generate an interactive interface to explore them. A script is provided in the examples/plot_analyze folder of the repository, an example shown at vsoch/caliper-analysis, and a screensht shown below.

caliper.yaml

The caliper.yaml file is a small configuration file to run caliper. Currently, it's fairly simply and we need to define the dependency to run tests over (e.g., tensorflow), the Dockerfile template, a name, and then a list of runs:

```
analysis:
  name: Testing tensorflow
  packagemanager: pypi
  dockerfile: Dockerfile
  dependency: tensorflow
  versions:
    - 0.0.11
  python_versions:
    - cp27
  tests:
    - tensorflow_v0.11/5_MultiGPU/multigpu_basics.py
    - tensorflow_v0.11/1_Introduction/basic_operations.py
    - tensorflow_v0.11/1_Introduction/helloworld.py
    - tensorflow_v0.11/4_Utils/tensorboard_advanced.py
```

If you don't define a list of `python_versions` all will be used by default. If you don't define a list of `versions` (e.g., versions of tensorflow) all versions of the library will be tested. If you want to add custom arguments for your template (beyond a base image that is derived for your Python software, and the dependency name to derive wheels to install) you can do this with args:

```
analysis:
  name: Testing tensorflow
  packagemanager: pypi
  dockerfile: Dockerfile
  args:
     additionaldeps:
       - scikit-learn
```

The functionality of your arguments is up to you. In the example above, `additionaldeps` would be a list, so likely you would loop over it in your Dockerfile template (which uses jinja2).

Dockerfile

The `Dockerfile` template (specified in the caliper.yaml) should expect the following arguments from the caliper analysis script:

- **base**: The base python image, derived from the wheel we need to install
- **filename**: the url filename of the wheel to download with wget
- **basename**: the basename of that to install with pip

Additional arguments under args will be handed to the template, and are up to you to define and render appropriately.

Metrics Extractor

Finally, a metrics extractor provides an easy interface to iterate over versions of a package, and extract some kind of metric. There are two ways to go about it - starting with a repository that already has tags of interest, or starting with a manager that will be used to create it. For each, you have three options for how to save data:

- **json**: is a folder with a json file for each version. This is recommended for large repositories (e.g., tensorflow)
- **json-single**: is a single json file of results, recommended for smaller repositories (e.g., sregistry)
- **zip**: is an intermediate, a compression json file, recommended for large but not massive repositories.

You can specify the format with `--fmt (json|json-single|zip)`. The default is json, which is the most conservative to ensure small file sizes for GitHub. It's recommended that you test extractions and choose the size that is right for you. Whatever you choose, an `index.json` file is generated in the output metric folder that will make it possible to detect what is available programatically with a request. We currently only support one extraction type at once, however if you think it necessary, we can add support for multiple.

Metrics Available

All caliper metrics available can be found in the metrics/collection folder in the caliper repository. Adding a new metric corresponds to creating a new folder here, and then adding the metric to this list.

- **functiondb**: is the function database metric. This means that for each version of a library we extract a lookup of all module functions, classes, and arguments. You might, for example, use this lookup to compare changes in the module over time.
- **changedlines**: is exactly what it sounds like - we look _between_ versions and count the number of changed lined. Thus, this uses a `ChangeMetricBase` instead of a standard `MetricBase`.
- **totalcounts**: is also exactly what it sounds like! We look at each version and create a lookup that shows the total number of files for the metric. If other totals are wanted, we could add them here.

You can look at the caliper-metrics repository to get a sense of the data. The changedlines metric is great for creating plots to show change in the module over time, while the functiondb is great for assessing overall change to the module structure.

Extraction Using Client

When installed, caliper comes with an executable, `caliper` that can make it easy to extract a version repository.

```
$ caliper --help
usage: caliper [-h] [--version] {version,metrics,analyze,extract,view} ...

Caliper is a tool for measuring and assessing changes in packages.

optional arguments:
  -h, --help            show this help message and exit
  --version             suppress additional output.

actions:
  actions

  {version,metrics,analyze,extract,view}
                        actions
    version             show software version
    metrics             see metrics available
    analyze             analyze functionality of a package.
    extract             extract one or more metrics for a software package.
    view                extract a metric and view a plot.

LOGGING:
  --quiet               suppress logging.
  --verbose             verbose output for logging.
  --log-disable-color   Disable color for caliper logging.
  --log-use-threads     Force threads rather than processes.
```

The `extract` command allows to extract metrics for a package:

```
$ caliper extract --help
usage: caliper extract [-h] [--metric METRIC] [-f {json,zip,json-single}] [--no-
↪cleanup] [--outdir OUTDIR] [--force]
                       [packages [packages ...]]

positional arguments:
```

(continues on next page)

(continued from previous page)

```
  packages              package to extract, e.g., pypi:, github:

optional arguments:
  -h, --help            show this help message and exit
  --metric METRIC       one or more metrics to extract (comma separated), defaults to␣
↪all metrics
  -f {json,zip,json-single}, --fmt {json,zip,json-single}, --format {json,zip,json-
↪single}
                        the format to extract. Defaults to json (multiple files).
  --no-cleanup          do not cleanup temporary extraction repositories.
  --outdir OUTDIR       output directory to write files (defaults to temporary␣
↪directory)
  --force               if a file exists, do not overwrite.
```

But first we might want to see metrics available:

```
$ caliper metrics
     totalcounts: caliper.metrics.collection.totalcounts.metric.Totalcounts
      functiondb: caliper.metrics.collection.functiondb.metric.Functiondb
    changedlines: caliper.metrics.collection.changedlines.metric.Changedlines
```

Let's say we want to extract the changedlines metric for a pypi repository, sif, which will return insertions, deletions, and overall change for each tagged release. That would look like this:

```
$ caliper extract --metric changedlines pypi:sif
Found 2 versions for sif
Downloading and tagging 0.0.1, 1 of 2
Downloading and tagging 0.0.11, 2 of 2
Repository for [manager:pypi] is created at /tmp/sif-94zn1b6b
Results will be written to /home/vanessa/Desktop/Code/caliper-metrics/pypi/sif
```

You can change the format by specifying `--fmt`

```
$ caliper extract --metric changedlines --fmt zip pypi:sif
$ caliper extract --metric changedlines --fmt json-single pypi:sif
```

By default, if you don't specify an output directory, the metrics will be saved to the present working directory. The organizaion is by package type, name, and then results files Here we can see results in all three formats: `zip`, `json` (multiple files), and `json-single`:

```
$ tree
└── pypi
    └── sif
        └── changedlines
            ├── changedlines-0.0.1..0.0.11.json
            ├── changedlines-EMPTY..0.0.1.json
            ├── changedlines-results.json-single
            ├── changedlines-results.zip
            └── index.json
```

And the index file makes it possible to see the contents of the folder (e.g., from a programmatic standpoint):

```
{
    "data": {
        "zip": {
            "url": "totalcounts-results.zip"
```

(continues on next page)

(continued from previous page)

```
        },
        "json": {
            "urls": [
                "totalcounts-0.0.1.json",
                "totalcounts-0.0.11.json"
            ]
        },
        "json-single": {
            "url": "totalcounts-results.json-single"
        }
    }
}
```

As an alternative to saving in the present working directory, you can instead save to an output folder of your choosing (with the same structure).

```
$ mkdir -p examples/metrics/
$ caliper extract --metric changedlines --outdir examples/metrics/ pypi:sif
```

For a change metric (a type that looks at change across tagged commits) you'll see a range of version like *EMPTY..0.0.1*. For a metric specific to a commit you will see just the tag (e.g., *0.0.1*). To extract just one specific version (or a list of comma separated versions with no spaces) you can define `--versions`:

```
$ caliper extract --metric functiondb --versions 0.12.1 pypi:tensorflow
```

Extraction Using Manager

The manager knows all the files for a release of some particular software, so we can use it to start an extraction. For example, let's say we have the Pypi manager above:

```
from caliper.managers import PypiManager
manager = PypiManager("sregistry")

manager
# [manager:sregistry]
```

We can then hand it off to the extractor:

```
from caliper.metrics import MetricsExtractor
extractor = MetricsExtractor(manager)

# This repository will have each release version represented as a tagged commit
repo = extractor.prepare_repository()

...
[master b45263b] 0.2.34
 8 files changed, 60 insertions(+), 65 deletions(-)
[master 555962b] 0.2.35
 4 files changed, 4 insertions(+), 4 deletions(-)
[master ead9302] 0.2.36
 117 files changed, 141 insertions(+), 141 deletions(-)
Repository for [manager:sregistry] is created at /tmp/sregistry-j63wuvei
```

At this point you'll see the extractor iterating through each repository version, and commiting changes based on the version. It's fun to open the repository folder (in /tmp named based on the package) and watch the changes happening

in real time. At this point we would have our **version repository** that we can calculate metrics over. For example, we can see commits that correspond to versions:

```
$ git log
commit ead9302cec47e62f8eabc5aefc0e55eeb3b8d717 (HEAD -> master, tag: 0.2.36)
Author: vsoch <vsochat@stanford.edu>
Date:   Fri Dec 18 14:51:34 2020 -0700

    0.2.36

commit 555962bad5f9e6d0d8ea4c4ea6bb0bdcb92d12f3 (tag: 0.2.35)
Author: vsoch <vsochat@stanford.edu>
Date:   Fri Dec 18 14:51:34 2020 -0700

    0.2.35

commit b45263b9c4da6aef096d49cc222bb9c64d2f6997 (tag: 0.2.34)
Author: vsoch <vsochat@stanford.edu>
Date:   Fri Dec 18 14:51:34 2020 -0700

    0.2.34

commit 113bc796acbffc593d400a19471c56c36289d764 (tag: 0.2.33)
Author: vsoch <vsochat@stanford.edu>
Date:   Fri Dec 18 14:51:33 2020 -0700
...
```

We can see the tags:

```
$ git tag
0.0.1
0.0.2
0.0.3
...
0.2.34
0.2.35
0.2.36
```

This is really neat! Next we can use the extractor to calculate metrics.

Extraction from Existing

As an alternative, if you create a repository via a manager (or have another repository you want to use that doesn't require one) you can simply provide the working directory to the metrics extractor:

```
from caliper.metrics import MetricsExtractor
extractor = MetricsExtractor(working_dir="/tmp/sregistry-j63wuvei")
```

You can see that we've created a git manager at this root:

```
extractor.git
<caliper.managers.git.GitManager at 0x7ff92a66ca60>
```

And we then might want to see what metrics are available for extraction.

```
extractor.metrics
{'totalcounts': 'caliper.metrics.collection.totalcounts.metric.Totalcounts',
 'changedlines': 'caliper.metrics.collection.changedlines.metric.Changedlines'}
```

Without going into detail, there are different base classes of metrics - a `MetricBase` expects to extract some metric for one timepoint (a tag/commit) and a `ChangeMetricBase` expects to extract metrics that compare two of these timepoints. The metric `changedlines` above is a change metric, and `totalcounts` is a base metric (for one commit timepoint). We can then run the extraction:

```
extractor.extract_metric("changedlines")
```

Note that you can also extract all metrics known to the extractor.

```
extractor.extract_all()
```

The recommended way to save is then to use the `save_all` function, which loops through the known metrics that have been run:

```
extractor.save_all(outdir, force=False, fmt="json")
```

For formats you can again choose between `json`, `json-single`, and `zip`. As stated earlier, you'd want to use `json` for the largest repos and metrics (e.g., a function database, functiondb is very large, and this scales with the number of releases), a `json-single` for smaller metric/release combinations, and `zip` if it's somewhere in betwee. Caliper can load all three, so you don't need to worry.

Extraction From Repository

It can be useful for a later analysis to put the contents of a metrics extraction into a repository, such as what is present at vsoch/caliper-metrics on GitHub. We can easily create a MetricsExtractor class and then read content there as follows:

```
from caliper.metrics import MetricsExtractor
extractor = MetricsExtractor("pypi:tensorflow")
result = extractor.load_metric("functiondb")
```

For loading the metric, you can also provide a different `repository` (defaults to vsoch/caliper-metrics), `metric` name (required), `subfolder` (defaults to empty string), and `branch` (defaults to main). If the metric exists in the repository, it will download and load the data for you into result. If not, None will be returned. You can also load a zip metric from a repository:

```
result = extractor.load_metric("functiondb", extension="zip")
```

And finally, you can also load the metric directly from a filename, which might be appropriate if the file is too big for version control:

```
result = extractor.load_metric("functiondb", filename="functiondb-results.zip")
```

Either zip or json files are supported. Once you load the result, the extracted data should be available, with the top level a key for a version or a difference between two versions.

```
result.keys()
# dict_keys(['0.0.1'])
```

You can then continue to use the result as needed. For the example above, since we have function signatures for every version of tensorflow, we might generate a comparison or similiarity matric depending on those signatures.

Parsing Results

For each extractor, you can currently loop through them and extract results for the metric. The results are organized by version (e.g., `0.0.1`), or difference between versions (e.g., `0.0.1..0.0.11`), depending on the metric.

```
for name, metric in extractor:
    # Changedlines <caliper.metrics.collection.changedlines.metric.Changedlines at␣
↪0x7f7cd24f4940>

    # A lookup with versions
    metric.get_results()
```

For example, an entry in the changedlines group results might look like this:

```
{'0.2.34..0.2.35': {'size': 0, 'insertions': 4, 'deletions': 4, 'lines': 8}}
```

To say that between versions 0.2.34 and 0.2.35 there were 4 insertions, 4 deletions, and 8 lines changed total, and there was no change in overall size. We will eventually have more examples for how to parse and use this data.

Checking and Updating Metrics

Let's say you did an extraction, and have an output folder of current results.

```
pypi/sif/
├── changedlines
│   ├── changedlines-results.json
│   └── index.json
├── functiondb
│   ├── functiondb-results.json
│   └── index.json
└── totalcounts
    ├── index.json
    └── totalcounts-results.json
```

For a given package, you can check the status of all metrics with `caliper update --check`

```
$ caliper update --check pypi:sregistry
Found 82 versions for sregistry
[✓  ] pypi:sregistry|totalcounts is up to date.
[✓  ] pypi:sregistry|functiondb is up to date.
[✓  ] pypi:sregistry|changedlines is up to date.
```

or a specific metric:

```
$ caliper update --check pypi:sregistry --metric functiondb
Found 82 versions for sregistry
[✓  ] pypi:sregistry|functiondb is up to date.
```

Or if you have many results in a respository, you might want to run a nightly (or weekly) job to check for new releases, and if any new releases are found, to update your data. To support much larger numbersof checks, you can use a `caliper.yaml` file to list the metric modules that you want to update. The simplest version just has a name for each:

```
metrics:
  - name: pypi:tensorflow
```

(continues on next page)

(continued from previous page)

```
  - name: pypi:sif
  - name: pypi:sregistry
  - name: pypi:singularity-cli
```

But you can add additional arguments such as the metrics to check:

Notice that each package requires a prefix of the manager (pypi). You can then target this file with `caliper update`, or just specify a list of packages with the command:

```
$ caliper update pypi:tensorflow
```

Either way. first you might want to check to look for new versions (the following command detects the caliper.yaml in the present working directory:

```
$ caliper update --check
Found 110 versions for tensorflow
Found 2 versions for sif
Found 82 versions for sregistry
[✓  ] pypi:tensorflow|totalcounts is up to date.
[✓  ] pypi:tensorflow|functiondb is up to date.
[  ] pypi:tensorflow|changedlines has 1 new versions.
[✓  ] pypi:sif|totalcounts is up to date.
[✓  ] pypi:sif|functiondb is up to date.
[  ] pypi:sif|changedlines has 1 new versions.
[  ] pypi:singularity-cli|changedlines is not found.
[✓  ] pypi:sregistry|totalcounts is up to date.
[✓  ] pypi:sregistry|functiondb is up to date.
[  ] pypi:sregistry|changedlines has 1 new versions.
```

And then perform the update.

```
$ caliper update --check
```

Metrics View

To extract and view metrics, you can use *caliper view*

```
usage: caliper view [-h] [--metric METRIC] [--title TITLE] [--outdir OUTDIR] [--
↪force] input

positional arguments:
  input            input data file to visualize.

optional arguments:
  -h, --help       show this help message and exit
  --metric METRIC  a metric to extract
  --title TITLE    the title for the graph (defaults to one set by metric)
  --outdir OUTDIR  output directory to write files (defaults to temporary directory)
  --force          if a file exists, do not overwrite.
```

For example, let's say we want to view an already extracted metric. We would provide the file as input:

```
$ caliper view ../caliper-metrics/github/spack/spack/changedlines/changedlines-
↪results.json
```

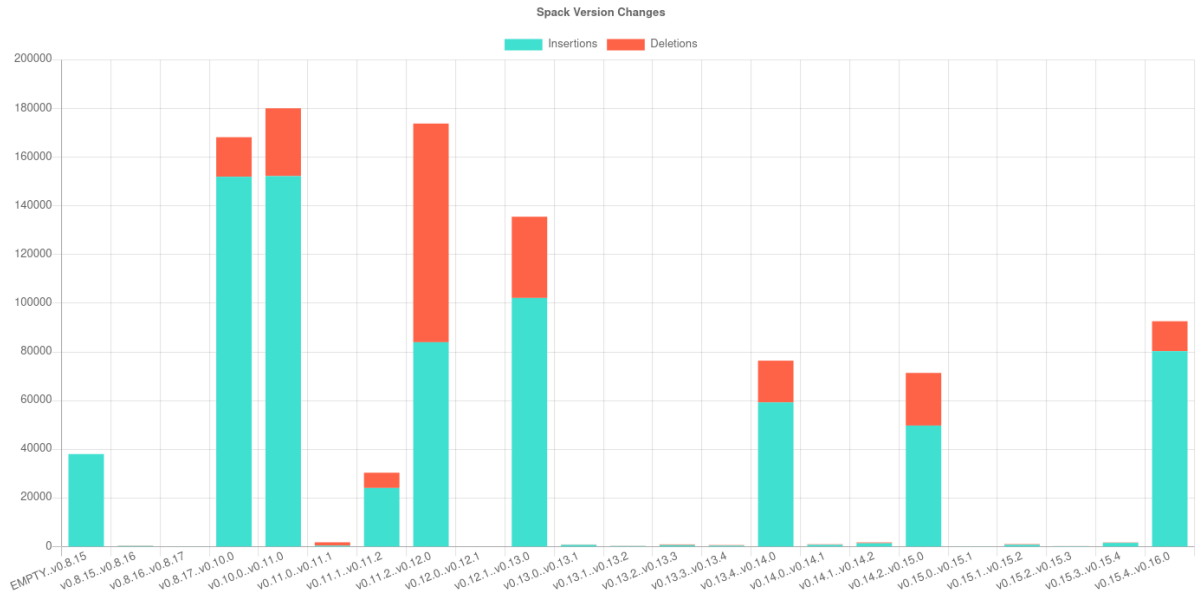
We might also add a custom title:

```
$ caliper view ../caliper-metrics/github/spack/spack/changedlines/changedlines-
↪results.json --title "Spack Version Changes"
```

Note that caliper will attempt to derive the metric name from the file. If you've renamed the file, then you'll need to provide it directly:

```
$ caliper view --metric changedlines mystery-file.json
```

Note from the usage that you can also select an output directory. Caliper tries to derive the name of the metric from the filename (e.g., `<metric>-results.json` however if you rename the file, you can specify the metric directly with `--metric`. An example output is shows here:



3.1.3 Use Cases

Assess Version Changes

Using the MetricsExtractor, we can start with a package and then calculate metrics for each version change, and ask questions like:

- What is the degree of change between minor/major versions?
- How much do dependencies change over time?
- How quickly does the package grow?

We might then be able to say that one package is more volatile than another, and use the metrics in other kinds of analyses.

Break a Workflow

An interesting use case for caliper is to use metrics to figure out if we can predict breaking. For example, we might have:

1. A Dockerfile with an entrypoint and command that generates some output
2. A list of requirements defined in a requirements.txt file (or similar)

And then we might derive a baseline container to run the workflow in question, and then vary package versions to determine if the container is still able to run and produce the same result, or if the dependency cannot be resolved all together. We can then assess, based on ranges of package versions that work vs. not and the degree of changes for each:

1. The degree to which some version increment is likely to break a build or run and
2. How close the developer was to representing a "correct" set of versions.

"Correct" is in quotes because we cannot easily assess dependency interaction (but perhaps can make some kind of proxy for it eventually).

Note this is all still being developed, and likely to change!

3.2 The Caliper API

These sections detail the internal functions for Caliper.

3.3 Internal API

These pages document the entire internal API of Caliper.

3.3.1 caliper package

Submodules

caliper.client module

caliper.analysis module

caliper.managers module

caliper.logger module

class caliper.logger.**Logger**(*MESSAGELEVEL=None*)

Bases: object

abort(*message*)

addColor(*level*, *text*)

addColor to the prompt (usually prefix) if terminal supports, and specified to do so

critical(*message*)

custom(*prefix*, *message=''*, *color='\x1b[95m'*)

debug(*message*)

emit(*level*, *message*, *prefix=None*, *color=None*)
emit is the main function to print the message optionally with a prefix :param - level: the level of the message :type - level: int :param - message: the message to print :type - message: str :param - prefix: a prefix for the message :type - prefix: str

emitError(*level*)
determine if a level should print to stderr, includes all levels but INFO and QUIET

emitOutput(*level*)
determine if a level should print to stdout only includes INFO

error(*message*)

exit(*message*, *return_code=1*)

exit_info(*message*, *return_code=0*)

failure(*message*)
Given a message string, print as a failure in red. :param - message: the message to print in red (indicating failure).

get_logs(*join_newline=True*)
'get_logs will return the complete history, joined by newline (default) or as is.

info(*message*)

isEnabledFor(*messageLevel*)
check if a messageLevel is enabled to emit a level

is_quiet()
is_quiet returns true if the level is under 1

log(*message*)

newline()

show_progress(*iteration*, *total*, *length=40*, *min_level=0*, *prefix=None*, *carriage_return=True*, *suffix=None*, *symbol=None*)
create a terminal progress bar, default bar shows for verbose+

Parameters

- **iteration**(*current iteration (Int)*)–
- **total**(*total iterations (Int)*)–
- **length**(*character length of bar (Int)*)–

success(*message*)
Given a message string, print as a success in green. :param - message: the message to print in green (indicating success).

table(*rows*, *col_width=2*)
table will print a table of entries. If the rows is a dictionary, the keys are interpreted as column names. if not, a numbered list is used.

useColor()
useColor will determine if color should be added to a print. Will check if being run in a terminal, and if has support for asci

verbose(*message*)

verbose1(*message*)

verbose2(*message*)

verbose3(*message*)

warning(*message*)

write(*stream*, *message*)
write will write a message to a stream, first checking the encoding

caliper.logger.**convert2boolean**(*arg*)
convert2boolean is used for environmental variables that must be returned as boolean

caliper.logger.**setup_logger**(*quiet=False*, *nocolor=False*, *stdout=False*, *debug=False*, *use_threads=False*, *wms_monitor=None*)

PYTHON MODULE INDEX

C

INDEX