OXFORD

Systems biology

Microbench: automated metadata management for systems biology benchmarking and reproducibility in Python

Alexander L. R. Lubbock [1,2,†] and Carlos F. Lopez [1,2,3,*,‡]

[1]Department of Biochemistry, Vanderbilt University, Nashville, TN 37232, USA, [2]Vanderbilt-Ingram Cancer Center, Vanderbilt University, Nashville, TN 37232, USA and [3]Department of Biomedical Informatics, Vanderbilt University Medical Center, Nashville, TN 37203, USA

*To whom correspondence should be addressed.

[†]Present address: The Rosalind Franklin Institute, Harwell Science & Innovation Campus, Didcot OX11 0FA, UK

[‡]Present address: Altos Labs, 2000 Island Dr., Redwood City, CA 64065, USA

Associate Editor: Janet Kelso

Abstract

Motivation: Computational systems biology analyses typically make use of multiple software and their dependencies, which are often run across heterogeneous compute environments. This can introduce differences in performance and reproducibility. Capturing metadata (e.g. package versions, GPU model) currently requires repetitious code and is difficult to store centrally for analysis. Even where virtual environments and containers are used, updates over time mean that versioning metadata should still be captured within analysis pipelines to guarantee reproducibility.

Results: *Microbench* is a simple and extensible Python package to automate metadata capture to a file or Redis database. Captured metadata can include execution time, software package versions, environment variables, hardware information, Python version and more, with plugins. We present three case studies demonstrating *Microbench* usage to benchmark code execution and examine environment metadata for reproducibility purposes.

Availability and implementation: Install from the Python Package Index using `pip install microbench`. Source code is available from https://github.com/alubbock/microbench.

Contact: cflopezw@gmail.com

Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

Analysis pipelines in computational biology frequently make use of multiple software dependencies and are often run across heterogeneous compute environments (e.g. laptop, compute cluster, cloud). However, differences in software, hardware and configuration can affect both reproducibility and performance. Standardized virtual environments using virtualenv (virtualenv.pypa.io), pipenv (pipenv.pypa.io) or Anaconda (anaconda.com) are an important first step toward reproducibility. Containerization frameworks like Docker (Boettiger, 2015) further help by standardizing system libraries and adding portability, but require time to setup and compile, and so are not always used during development. Even with virtual environments and containers, it remains important to track software versions over time. In addition, Python *requirements.txt* files often only specify the direct dependencies of the code, and not downstream dependencies which may change if an environment or container is later re-built. Continuous integration (CI) (Meyer, 2014) can help to identify these issues, but reproducibility issues can also occur due to

differences in hardware (e.g. GPU model) and environment (e.g. environment variables). In addition, comprehensive CI is not always feasible with large -omics datasets. Capturing the full environment at runtime would help to identify the cause of reproducibility issues, should they be discovered later.

Here, we introduce *Microbench*, an open-source Python package which automates metadata capture and execution timing of annotated functions for performance benchmarking and to improve reproducibility. For example, *Microbench* can capture Python package versions, environment variables, host hardware specifications, Python version and function arguments. It can be applied to both pure Python code and in scenarios where Python wraps external scripts and non-Python libraries. *Microbench* can capture line-by-line execution times and telemetry—CPU and RAM utilization recorded periodically during function execution. Results are logged to a file or a central Redis database (redis.io) and can be analyzed using *pandas* DataFrames (pandas.pydata.org). *Microbench* can be extended with plugins to capture additional metadata types. *Microbench* lowers the administrative burden of capturing metadata

for benchmarking and reproducibility. We believe it will significantly help computational biologists working in Python who faces these challenges.

2 Results

Implementation
Microbench is written in pure Python for cross-platform compatibility (tested on Windows 10, Mac, Linux) and has no default runtime dependencies outside of the Python standard library. The *line_profiler* package (pypi.org/project/line-profiler) is required for line-by-line benchmarking, and *pandas* is recommended to examine and analyze results. *Microbench* can be easily extended with new metadata capture capabilities.

Installation and usage
Microbench is installed using `pip install microbench` (Supplementary Text S1). Source code is available on GitHub (https://github.com/alubbock/microbench) under the MIT license. A minimal usage example is given in Supplementary Text S2; an extended usage example is given in Supplementary Text S3. Briefly, a benchmark suite is first specified, which determines what metadata are captured. A Python decorator is then used to mark functions for benchmarking/metadata capture. Results are saved to a JSON file or Redis instance for later analysis (Fig. 1). The results format is described in Supplementary Text S4.

Metadata capture
Microbench can capture a wide variety of metadata. By default, start timestamp, end timestamp and function name are captured. Additional metadata are determined by the user in the form of mixins—small Python classes, which are included when the benchmark suite is constructed. A list of available mixins is shown in Supplementary Table S1. *Microbench* can be extended with custom mixins to capture additional or bespoke metadata (Supplementary Text S5).

Redis support
Microbench typically appends metadata to a file, but includes support for Redis—an unstructured, in-memory database. If Redis is run on an internet-accessible server, metadata can be deposited from any internet-connected device. *Microbench* Redis support requires the redis-py package (pypi.org/project/redis) (Supplementary Text S6).

NVIDIA GPU support
Attributes relevant to NVIDIA GPUs can be captured using a metadata collector for NVIDIA's nvidia-smi utility (developer.nvidia.com/nvidia-system-management-interface), including GPU model number and GPU memory capacity (Supplementary Text S7).

Line profiler support
Microbench can capture line-by-line execution times using the line_profiler package. This is particularly useful in algorithm development and for bottleneck identification (Supplementary Text S8).

Telemetry support
Microbench can capture function telemetry—metrics such as CPU and RAM utilization—at specified intervals using a background thread. This can help identify resource intensive code blocks (Supplementary Text S9).

Case study: biochemical model simulations
We include an application of *Microbench* to a biochemical model simulation using PySB (Lopez *et al.*, 2013) within a Jupyter Notebook (Perkel, 2018), where we capture package versions from an Anaconda environment to check for reproducibility (Supplementary File S1, Example 1). This example also uses telemetry analysis to examine CPU and RAM utilization over time during function execution. An example using Tellurium (Choi *et al.*, 2016) is also provided (Supplementary File S1, Example 4).

Case study: NumPy return value differs by version
Software library updates can result in differences or errors, which may or may not be documented in those libraries' documentation or spotted by the end user. We present a simple example case study where a computation gives a different result using NumPy 1.15 compared to NumPy 1.20 (https://numpy.org/doc/stable/release/1.20.0-notes.html#np-linspace-on-integers-now-uses-floor). Capturing version metadata can help to identify such issues retrospectively (Supplementary File S1, Example 2).

Case study: benchmarking with SLURM
SLURM (Yoo *et al.*, 2003) is a cluster management system which can be used to execute computational jobs in parallel. Large clusters are often subject to rolling or phased hardware refreshes, leading to a heterogeneous compute environment. *Microbench* can capture hardware specifications, and subsequently, the runtime of an algorithm can be compared by feature (e.g. max. CPU frequency, total RAM) for benchmarking or specifying requirements (Supplementary File S1, example 3).

3 Discussion

Microbench is designed to provide a simple and unobtrusive way to benchmark Python functions and capture key metadata for reproducibility. It is particularly useful in heterogeneous compute environments and ephemeral cloud computing environments, where one cannot easily go back and examine the exact instance after analysis completion. *Microbench* has no runtime dependencies outside the Python standard library, making it easy to deploy across
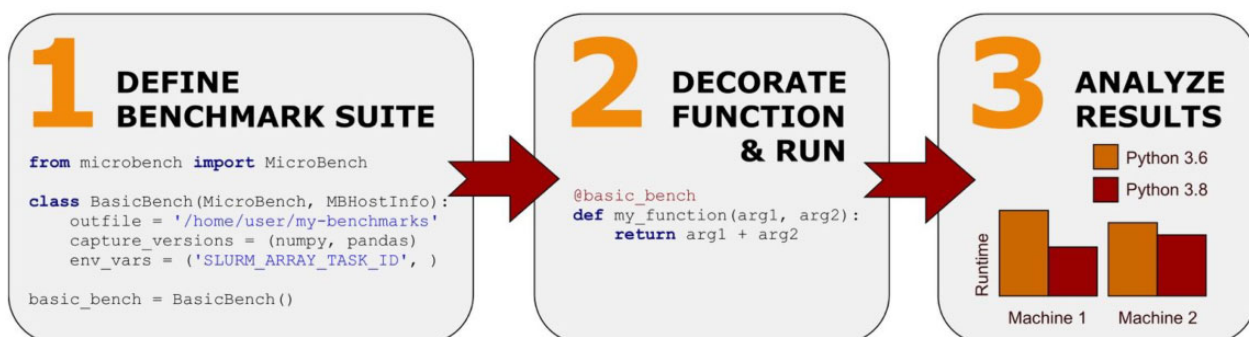

Fig. 1. *Microbench* usage. The user defines the set of metadata to capture, then decorates the target function and runs it, then analyzes the results. Captured metadata can be used to check issues in case of a reproducibility anomaly, or for performance benchmarking

heterogeneous compute environments. For containers (e.g. Docker), we recommend deploying *Microbench* inside the container alongside the target code. By making metadata capture simple to add to existing Python code, *Microbench* can be used routinely, allowing for retrospective debugging and benchmarking should an issue arise later. We believe the research community should build on existing metadata standards (Leipzig *et al.*, 2021) to establish minimum requirements and best practices toward greater reproducibility.

Funding

Conflict of Interest: none declared.

References

Boettiger,C. (2015) An introduction to Docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, **49**, 71–79.

Choi,K. *et al.* (2016) Tellurium: a Python based modeling and reproducibility platform for systems biology. *BioRxiv* 054601. https://doi.org/10.1101/054601.

Leipzig,J. *et al.* (2021) The role of metadata in reproducible computational research. *Patterns*, **2**, 100322.

Lopez,C.F. *et al.* (2013) Programming biological models in Python using PySB. *Mol. Syst. Biol.*, **9**, 646. https://doi.org/10.1038/msb.2013.1.

Meyer,M. (2014) Continuous integration and its tools. *IEEE Softw.*, **31**, 14–16.

Perkel,J.M. (2018) Why Jupyter is data scientists' computational notebook of choice. *Nature*, **563**, 145–146.

Yoo,A.B. *et al.* (2003). SLURM: simple linux utility for resource management. In: Feitelson, D. *et al.* (eds.) *Job Scheduling Strategies for Parallel Processing*. Springer, Berlin, Heidelberg, pp. 44–60.