Reproducing and Improving the BugsInPy Dataset

Faustino Aguilar
Dept. of Computer Engineering
University of Panama
Panama City, Panama
orcid.org/0009-0000-1375-1143

Samuel Grayson
Dept. of Computer Science
University of Illinois Urbana-Champaign
Urbana, IL, USA
orcid.org/0000-0001-5411-356X

Darko Marinov
Dept. of Computer Science
University of Illinois Urbana-Champaign
Urbana, IL, USA
orcid.org/0000-0001-5023-3492

***Abstract*—We assess the reproducibility of the BugsInPy dataset less than three years after its original publication. The bug dataset provides some information about the software environment in which the code should be run, but this information can be incomplete or can decay into something uninstallable over time. We rectify as many of these problems as we can and redesign the original dataset to be more easily reusable and reproducible by future research projects. Based on our experience, we offer suggestions to authors of Python artifacts to improve their reproducibility.**

***Index Terms*—reproducibility, bug database, BugsInPy, Python, package managers, Pip, Conda, containers, Docker**

I. Introduction

BugsInPy [1] is a curated dataset of real-world bugs in large Python projects, intended to be used by researchers to develop and evaluate software testing and debugging tools for Python on a diverse set of real-world bugs from multiple projects. This dataset can be used to evaluate the efficacy of tools in bug detection, fixing, software reliability, and more. For example, several software engineering studies [2]–[6] already use BugsInPy.

The BugsInPy dataset contains a variety of information about each bug, and these bugs are organized by the project they come from, including:

- A buggy commit
- A fixed commit
- Python version used
- One or more test cases that indicate the bug's presence

The BugsInPy dataset includes a database abstraction layer and a test execution framework. The database abstraction layer provides a way to access the dataset in a structured way. The test execution framework allows researchers to run test cases relevant to a particular bug.

We sought to use the BugsInPy dataset to verify, for each bug, that all the test cases could be set up, that the buggy commit fails, and that the fixed commit passes. Our work is a *reproduction* (using ACM's 2020 definition [7]) because we use the scripts from the original work. (In contrast, our work would be a *replication* if we did not use any original scripts but wrote all new scripts from scratch to attempt to repeat the original results.)

Our contributions include:

- Improvements to the BugsInPy test execution framework, which make it easier to run experiments *en masse*.
- Modifications to the BugsInPy test execution framework, which install and use the correct version of Python.
- The results of which bugs were reproducible with and without our improvements and modifications.

We evaluate the following research questions:

RQ1. How many bugs in BugsInPy are reproducible with no "extra" work? For a bug to be reproducible, the software environment should install without failure, the buggy version should fail the identified test cases, and the fixed version should pass these test cases.

RQ2. How many non-reproducible bugs can we *rescue*? We rescue a bug by modifying the scripts and data such that the bug that was initially not reproducible now becomes reproducible.

This article proceeds with the methodology section, which explains how we first tried to reproduce BugsInPy and what rescue procedures we took when bugs were not reproducible. Then we summarize the results of our executions and analyze the failures. Finally, we engage in an open-ended discussion of our experiments with several pieces of advice to authors of future reproducible artifacts in Python and those seeking to reproduce such artifacts.

II. Methodology

We first tried to reproduce each bug using the original script released with the BugsInPy dataset. We run the script on an Ubuntu 18.04 distribution, because the Docker image mentioned in the original BugsInPy paper used Ubuntu 18.04. For bugs that were not reproducible using the original script, we analyzed the problems to identify the root cause and then made changes to try to *rescue* the bugs and make them reproducible.

As part of our rescue process, we made the following changes:

1. **Added Dockerfile for containers:** We use a Docker container build script, `Dockerfile`, to build a Docker image that provides a consistent starting point in which our scripts can install the correct software environment. The BugsInPy original paper and the current Git repository do not mention or include any Dockerfile, but DockerHub

does have an image for BugsInPy published by one of the authors of the BugsInPy original paper; however, we could not find a Dockerfile corresponding to that image. Moreover, we wanted our image to include other changes mentioned below (e.g., Conda). The use of container sandboxes modifications that the BugsInPy script makes to the environment (e.g., modifying `~/.bashrc`). While our image is available in the popular DockerHub registry, we suggest that users seeking robust reproducibility build the image from our Dockerfile available in our Git repository rather than depending on an external registry.

2. **Added Conda package manager:** Each bug may require a different version of Python, as specified in the dataset, but the BugsInPy script *ignores* the specified version of Python, deferring to the system default Python instead. Presumably, the BugsInPy authors manually changed their system's version of Python according to the specification of each bug, but this process is not fully automated, making it difficult for future users. We modified the main BugsInPy script to install the correct version of Python using Conda. Conda is a cross-platform package manager. Packages installed by Conda neither use nor modify the system version of those packages, so Conda can support different environments, each with its own requirements that may potentially conflict with the other environments. Conda package repositories store packages containing prebuilt binaries and metadata for each platform, so installing is much faster than compiling from source code.
3. **Replaced Pip with Conda where appropriate:** The original BugsInPy scripts install all Python packages using Pip package manager. Pip can invoke the compiler to build dependencies from source code [8] or download prebuilt binary files. The most common usage of Pip is to install packages from the Python Package Index (PyPI) using requirement specifiers. As specified in the official Python Packaging documentation, a requirement specifier typically consists of a project name followed by an optional version specifier. PEP 440 provides the specification for requirement specifiers, including a comprehensive guide to the currently supported specifiers [9]. However, some packages may require additional system libraries or dependencies that cannot be installed solely through Pip. For example, Matplotlib, a popular Python plotting library, has required system-level dependencies that Pip cannot automatically handle, such as libpng, freetype, or Tk [10]. Consequently, if a bug in a project's environment depends on Matplotlib, attempting to install and run that project on a vanilla Ubuntu or Debian system without the necessary system libraries would result in installation failures. In such cases, it becomes the responsibility of the user or system administrator to ensure that the required system libraries are installed manually before attempting to install the package with Pip. The Matplotlib documentation provides detailed instructions on how to install the necessary system-level dependencies for different platforms [11]. By following these instructions and setting up the required libraries, users can successfully install and utilize Matplotlib and any other package with similar external dependencies. Presumably, the original BugsInPy authors manually modified their system to have these system libraries; in our case, we identify packages that Pip cannot install on vanilla Ubuntu or Debian and simply install those with Conda instead.
4. **Added caching of environments:** Building the environment from source code can be costly, so we reuse environments across many bugs when their Python package requirements and Python versions are identical. This optimization helps reduce the time and resources required for environment setup, as it bypasses the costly process of building environments from source code. While it is tempting to use the same Conda environment for all bugs in each project (rather than for each bug), there are multiple occasions where different bugs of the same project require different dependencies. For example, `ansible/bugs/{1,11,14}/requirements.txt` all vary subtly.
5. **Correct installation of requirements:** The BugsInPy dataset correctly recognizes that installing the dependencies line-by-line `cat requirements.txt | filter | xargs -n 1 pip install`, rather than using `pip install -r requirements.txt`, bypasses certain restrictions imposed by Pip. Specifically, when installing all dependencies at once, Pip may ignore very old packages. However, sequentially installing the dependencies allows us to install these old packages and thus reproduce the bugs accurately. However, installing the dependencies line-by-line results in failed installations for projects that include the `-e git+https://...` syntax in their `requirements.txt` file, because they would get passed along as `pip install -e` and `pip install git+https://...`. Our revised script ensures that each line from the `requirements.txt` file is properly processed and passed as an argument to the `pip install` command. To correct this issue in the BugsInPy dataset, we have opened a pull request in the original repository [12]. This fix is crucial, because it impacts the reproducibility of bugs in several projects such as `black`, `cookiecutter`, `keras`, `luigi`, `pandas`, `sanic`, and `thefuck`. We have started from this pull request because it is the simplest of our five changes; if we receive some feedback for the original BugsInPy authors, we plan to open pull requests for the other four changes.

III. Results

This section presents and discusses our results on reproducing bugs in BugsInPy before and after our changes. Table I shows the results without our modifications, and Table II shows the

results after our modifications. The outcomes that we can get for bugs are:

- **Error (Err)**: Some step in the installation of the software environment needed to reproduce the bug failed.
- **Both-pass (B-pass)**: Both versions pass, although we would expect the buggy version to fail.
- **Both-fail (B-fail)**: Both versions fail, although we would expect the fixed version to pass.
- **Expected (Exp)**: The buggy version fails, and the fixed version passes. We consider *only* these bugs as actually "reproduced".

The tables show, for each project, the raw count and percentage of outcomes for all bugs in that project. The last, summary rows show the raw count and percentage of outcomes for all bugs in the BugsInPy dataset.

TABLE I
REPRODUCTION OF BUGS IN BUGSINPY WITHOUT OUR MODIFICATIONS

Project	Err	B-pass	B-fail	Exp	Total
PySnooper	2 (67%)	0 (0%)	0 (0%)	1 (33%)	3 (100%)
ansible	3 (17%)	0 (0%)	0 (0%)	15 (83%)	18 (100%)
black	1 (4%)	0 (0%)	0 (0%)	22 (96%)	23 (100%)
cookiecutter	2 (50%)	0 (0%)	0 (0%)	2 (50%)	4 (100%)
fastapi	0 (0%)	0 (0%)	0 (0%)	16 (100%)	16 (100%)
httpie	4 (80%)	0 (0%)	0 (0%)	1 (20%)	5 (100%)
keras	14 (31%)	0 (0%)	0 (0%)	31 (69%)	45 (100%)
luigi	33 (100%)	0 (0%)	0 (0%)	0 (0%)	33 (100%)
matplotlib	29 (97%)	0 (0%)	0 (0%)	1 (3%)	30 (100%)
pandas	47 (28%)	0 (0%)	0 (0%)	122 (72%)	169 (100%)
sanic	5 (100%)	0 (0%)	0 (0%)	0 (0%)	5 (100%)
scrapy	11 (28%)	0 (0%)	0 (0%)	29 (72%)	40 (100%)
spacy	2 (20%)	0 (0%)	0 (0%)	8 (80%)	10 (100%)
thefuck	8 (25%)	0 (0%)	0 (0%)	24 (75%)	32 (100%)
tornado	1 (6%)	0 (0%)	0 (0%)	15 (94%)	16 (100%)
tqdm	2 (22%)	0 (0%)	0 (0%)	7 (78%)	9 (100%)
youtube-dl	0 (0%)	0 (0%)	0 (0%)	43 (100%)	43 (100%)
Total	164 (33%)	0 (0%)	0 (0%)	337 (67%)	501 (100%)

RQ1. We can reproduce 67% of the expected results in the unmodified BugsInPy dataset.

TABLE II
REPRODUCTION OF BUGS IN BUGSINPY AFTER RESCUING

Project	Err	B-pass	B-fail	Exp	Total
PySnooper	1 (33%)	0 (0%)	1 (33%)	1 (33%)	3 (100%)
ansible	0 (0%)	0 (0%)	0 (0%)	18 (100%)	18 (100%)
black	0 (0%)	0 (0%)	1 (4%)	22 (96%)	23 (100%)
cookiecutter	0 (0%)	0 (0%)	0 (0%)	4 (100%)	4 (100%)
fastapi	0 (0%)	0 (0%)	0 (0%)	16 (100%)	16 (100%)
httpie	0 (0%)	0 (0%)	0 (0%)	5 (100%)	5 (100%)
keras	3 (7%)	0 (0%)	1 (2%)	41 (91%)	45 (100%)
luigi	0 (0%)	6 (18%)	0 (0%)	27 (82%)	33 (100%)
matplotlib	3 (10%)	1 (3%)	0 (0%)	26 (87%)	30 (100%)
pandas	4 (2%)	0 (0%)	0 (0%)	165 (98%)	169 (100%)
sanic	0 (0%)	0 (0%)	0 (0%)	5 (100%)	5 (100%)
scrapy	0 (0%)	2 (5%)	0 (0%)	38 (95%)	40 (100%)
spacy	1 (10%)	0 (0%)	0 (0%)	9 (90%)	10 (100%)
thefuck	0 (0%)	0 (0%)	0 (0%)	32 (100%)	32 (100%)
tornado	0 (0%)	0 (0%)	0 (0%)	16 (100%)	16 (100%)
tqdm	0 (0%)	0 (0%)	0 (0%)	9 (100%)	9 (100%)
youtube-dl	0 (0%)	0 (0%)	0 (0%)	43 (100%)	43 (100%)
Total	12 (2%)	9 (2%)	3 (1%)	477 (95%)	501 (100%)

RQ2. We were able to rescue 85% of the non-reproducible bugs in the original BugsInPy, resulting in a total reproduction rate of 95%.

With over 95% of bugs being successfully reproduced (passing the test cases in the fixed commit and failing test cases in the buggy commit), researchers have more bugs at their disposal for using BugsInPy, e.g., for evaluating fuzzing, automatic program repair, and other research techniques.

Table III presents the running time taken to run the respective containers that attempt to reproduce bugs in each project within the BugsInPy dataset. We include the time for both bugs that we could reproduce and bugs that we could not reproduce. These times are important to help researchers estimate the resources needed for running their future experiments; the original BugsInPy paper did not include these running times. The provided running times are specific to the reproduction procedure on the given VM configuration, which had 4 cores, 8GB of RAM, and 100GB of free disk space. Reproduction times can vary depending on hardware resources, system configurations, and other environmental factors. The projects are sorted based on their running time in descending order, with the project `pandas` having the highest running time of 963 minutes, followed by `luigi`, `scrapy`, and so on.

TABLE III
REPRODUCTION TIME FOR BUGS IN EACH PROJECT

Project	Running Time (minutes)
pandas	963
luigi	510
scrapy	268
keras	230
black	214
fastapi	197
thefuck	195
sanic	136
spacy	131
ansible	80
tqdm	36
youtube-dl	59
cookiecutter	40
httpie	39
matplotlib	26
tornado	14
pysnooper	5

IV. DISCUSSION

A. What makes reproduction easy?

The ease of bug reproduction in the BugsInPy dataset can be attributed to several factors:

1. **Automatation**: Our `bugsinpy-testall` script provides an automated approach to reproducing and testing all bugs in all projects included in the BugsInPy dataset. The script streamlines the overall reproduction process, minimizes manual effort, and ensures we use a consistent procedure on each project. The script also allows to select reproduction of only some of the bugs in some of the

projects. The automation script must be carefully written and maintained to handle various possible errors. For example, the original script did not have `set -e`, so some intermediate step may fail without alerting the user.
2. **Environment/package manager**: The Conda environment/package manager simplifies the management of project dependencies. The crucial insight is that Conda can install packages in a local environment without interfering with global, system-wide packages. Conda makes it possible to define project-specific versions of libraries that a platform-specific system-wide package manager would normally manage.
3. **Lack of non-deterministic bugs**: All bugs in the BugsInPy dataset are supposed to be deterministic. Our scope is limited to constructing a reproducible software environment consistent with the original bug, where the bug can manifest itself deterministically.

These factors collectively contribute to the ease of reproducing bugs in the BugsInPy dataset, providing a reliable and efficient dataset for bug analysis and investigation.

B. What makes reproduction hard?

Despite the easy-to-reproduce factors mentioned above, bug reproduction can still present challenges due to the following factors:

1. **Resource constraints during building**: The software environment can involve a computationally expensive step of building software from source code. Reproducing and testing many bugs within limited resources may result in longer reproduction times and potential resource limitations. Our script creates many Conda environments. These environments can be expensive to store, and we cannot, for example, archive our environments in GitHub due to space constraints.
2. **Missing packages in Conda**: Unfortunately, not all Pip packages and versions exist in our selected Conda repositories.

Addressing these challenges requires careful consideration of project-specific factors and may involve additional research, debugging techniques, and resources to ensure accurate and reliable bug reproduction.

C. Recommendations to Python artifact authors

For authors providing Python research artifacts, the following recommendations can enhance the reproducibility of their artifacts:

1. **Make it automatic/easy to use**: The BugsInPy dataset has Python versions, but there is no automation to switch to a specific version, so users are unlikely to do so. Our improved version uses Conda to switch to the correct Python version automatically.
2. **`requirements.txt` is not enough**: Pip cannot handle library dependencies. Researchers should provide a container, a Conda lockfile, Spack lockfile, or other detailed environment specification.
3. **Archival storage**: Ensure that the artifact repository is archived in long-term storage, such as Zenodo or FigShare, so it does not disappear. For example, some of our non-reproducible bugs are due to dependency versions that are not available, including `flake8` project that moved from GitLab to GitHub [13].

D. Threats to Validity

Some of the bugs that we find unreproducible could be actually reproduced with more effort than we expanded. Our effort may reflect an "average" user with limited resources, not a researcher with much more available time and resources.

While we show how to increase reproducibility of the BugsInPy dataset, our own work may not be reproducible for the following reasons:

1. Although we pin the exact version of our Docker base image, the image location (DockerHub) may stop hosting this base image (e.g., goes out of business, ends free tier). In this case, one would need to change the base image, but it could still work, so long as that base image has Conda. We find that Conda is still able to easily install rather old versions of Python.
2. Conda package repositories can stop existing (e.g., if Anaconda goes out of business), or they can drop the old package versions that our scripts use. However, the definition of Conda packages describes how to build the packages from source code. The package definitions are smaller than the binaries, so these package definitions may remain longer.
3. The reseachers trying to reproduce the results may need more computational resources to do the reproduction in a timely manner. We reduce the resource demands by reusing Conda environments. Furthermore, our scripts support reproducing just one project or just one bug from one project.

V. Conclusion

The study presented in this paper demonstrates the effectiveness of the BugsInPy dataset in reproducing and testing bugs in Python projects. The original BugsInPy dataset included highly useful information that aids in reproduction of these bugs, but the scripts had some issues that limited reproduction to 67% of the bugs in our experiments, before our modifications. Our modifications, embodied in the automated approach provided by our `bugsinpy-testall` script, coupled with the use of Conda for dependency management and Dockerfile for building images/containers, streamline the bug reproduction process and enhance its ease. The high success rate in reproducing bugs, with over 95% of bugs reproduced, indicates the reliability and accuracy that our modifications provided to the BugsInPy dataset. Our approach could be useful not only for BugsInPy but possibly also for other bug datasets to validate the reliability and accuracy in a more user friendly manner.

However, our experiments still depend on commercial organizations continuing to store software for free (GitHub, PyPI, Anaconda, DockerHub). Challenges still exist in creating a truly long-term reproducible software environment.

VI. Acknowledgments

We thank Mehzabin Haque and Rohit Naidu for comments on an earlier draft of this paper. We also thank Sugam Adhikari and Asif Zubayer Palak for the initial help in reproducing some bugs from BugsInPy. This work was partially supported by NSF grants CCF-1763788 and CCF-1956374. We also acknowledge support for research on flaky tests from Google and Meta.

References

[1] R. Widyasari, S. Q. Sim, C. Lok, *et al.*, "BugsInPy: A database of existing bugs in python programs to enable controlled testing and debugging studies," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, New York, NY, USA: Association for Computing Machinery, Nov. 8, 2020, pp. 1556–1560, ISBN: 978-1-4503-7043-1. DOI: 10.1145/3368089.3417943. [Online]. Available: https://doi.org/10.1145/3368089.3417943 (visited on 07/08/2023).

[2] E. N. Akimova, A. Y. Bersenev, A. A. Deikov, *et al.*, "A survey on software defect prediction using deep learning," *Mathematics*, vol. 9, no. 11, p. 1180, May 24, 2021, ISSN: 2227-7390. DOI: 10.3390/math9111180. [Online]. Available: https://www.mdpi.com/2227-7390/9/11/1180 (visited on 07/17/2023).

[3] S. Mukherjee, A. Almanza, and C. Rubio-González, "Fixing dependency errors for python build reproducibility," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Virtual Denmark: ACM, Jul. 11, 2021, pp. 439–451, ISBN: 978-1-4503-8459-9. DOI: 10.1145/3460319.3464797. [Online]. Available: https://dl.acm.org/doi/10.1145/3460319.3464797 (visited on 07/17/2023).

[4] T. Hirsch and B. Hofer, "A systematic literature review on benchmarks for evaluating debugging approaches," *Journal of Systems and Software*, vol. 192, p. 111 423, Oct. 2022, ISSN: 01641212. DOI: 10.1016/j.jss.2022.111423. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121222001303 (visited on 07/17/2023).

[5] M. Smytzek and A. Zeller, "SFLKit: A workbench for statistical fault localization," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Singapore Singapore: ACM, Nov. 7, 2022, pp. 1701–1705, ISBN: 978-1-4503-9413-0. DOI: 10.1145/3540250.3558915. [Online]. Available: https://dl.acm.org/doi/10.1145/3540250.3558915 (visited on 07/17/2023).

[6] S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of automated unit test generation for python," *Empirical Software Engineering*, vol. 28, no. 2, p. 36, Mar. 2023, ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-022-10248-w. [Online]. Available: https://link.springer.com/10.1007/s10664-022-10248-w (visited on 07/17/2023).

[7] A. I. staff. "Artifact review and badging." (Aug. 24, 2020), [Online]. Available: https://www.acm.org/publications/policies/artifact-review-and-badging-current (visited on 01/19/2023).

[8] "Cmdoption-no-binary - pip install - pip documentation v23.2." (), [Online]. Available: https://pip.pypa.io/en/stable/cli/pip_install/#cmdoption-no-binary (visited on 07/17/2023).

[9] "Installing packages — python packaging user guide." (), [Online]. Available: https://packaging.python.org/en/latest/tutorials/installing-packages/#installing-from-pypi (visited on 07/17/2023).

[10] "Installation — matplotlib 3.7.2 documentation." (), [Online]. Available: https://matplotlib.org/stable/users/installing/index.html (visited on 07/17/2023).

[11] "Contributing — matplotlib 3.7.2 documentation." (), [Online]. Available: https://matplotlib.org/stable/devel/index.html#building-matplotlib (visited on 07/17/2023).

[12] "Fixes -e option requires 1 argument. by faustinoaq · pull request #68 · soarsmu/BugsInPy," GitHub. (), [Online]. Available: https://github.com/soarsmu/BugsInPy/pull/68 (visited on 07/17/2023).

[13] "4.0.0 – 2021-10-10 — flake8 6.1.0 documentation." (), [Online]. Available: https://flake8.pycqa.org/en/latest/release-notes/4.0.0.html (visited on 08/23/2023).

Appendix
Code, Data, and Reproducing

A rolling release of all our code and data can be found at https://github.com/reproducing-research-projects/BugsInPy.

Our code includes:

- `Dockerfile` docker file setup to build projects images.
- `docker-compose.yml` orchestration to run containers.
- `framework/bin/bugsinpy-testall` script to automate execution of BugsInPy framework scripts.

To reproduce all bugs in a project, for example `httpie`, run:

```
$ rm -f projects/bugsinpy-index.csv
$ docker compose up setup httpie --build
Cleaning up temp folder ...
Reproducing bugs please wait ...
-------------------------
httpie,1,buggy,fail
...
```

After these commands, the new results will be in the file named `bugsinpy-index.csv`. See `README.md` for more detailed information.