Reproduction and Improvement on 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 at Urbana Champaign
Urbana, IL
orcid.org/0000-0001-5411-356X

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

***Abstract*—We present our experience on replicating a bug dataset for the Python programming language. We assess the reproducibility of the original dataset less than three years after its original publication. The bug dataset provides some information about the software environment, but this information can be incomplete or it can decay into something uninstallable. We rectify as many of these problems as we can and redesign the original dataset to be more easily reusable and reproducible by future authors. Based on our experience, we offer suggestions to Python artifact authors to improve their reproducibility.**

***Index Terms*—reproducibility, bug database, python**

I. Introduction

BugsInPy [1] is a curated dataset of real-world bugs in large Python projects. BugsInPy is 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 can help to ensure that the tools are effective in detecting real-world bugs.

BugsInPy is used in many software engineering studies [2]–[6]. Researchers and practitioners have leveraged the dataset to investigate different aspects of bug detection, fixing, and software reliability. BugsInPy has become an great resource for advancing the field of software engineering and fostering innovation in bug detection and fixing techniques.

The BugsInPy dataset contains a variety of information about each bug, including:

- A buggy commit
- A fixed commit
- Python version used
- Test cases that indicate the bugs presence

BugsInPy 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 the test cases that reveal the bugs.

BugsInPy is a valuable resource for researchers who are developing and evaluating software defect detection tools. The dataset provides a large and diverse set of bugs that can be used to test the effectiveness of these tools. BugsInPy also includes a variety of tools and resources that can help researchers to use the dataset effectively.

We sought to use the BugsInPy framework to verify that the test cases could be set up, that the buggy commit fails, and that the fixed commit passes. This is a *reproduction* [7] of the original work, since we are using the original framework.

Our contributions are:

- A new script which can invoke the BugsInPy framework *en masse*.
- Modifications to the BugsInPy framework to install and use the correct version of Python.
- The results of which bugs were reproducible (software environment installs, buggy version fails exactly that one test, fixed version passes all tests).

This article proceeds with the methodology section, which explains what tools we used to run these codes reproducible. Then we summarize the results of our executions and analyze the failures. Finally, we engage in an open-ended discussion of our experiment with advice to authors of reproducible artifacts and those seeking to reproduce artifacts.

II. Methodology

We would like to answer the following questions:

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

RQ2. How many of those which are not reproducible can we rescue? We rescue a bug by modifying the scripts and data such that the bug is reproducible.

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

1. We use a container build script to provide a consitent starting point in which our scripts will install the correct software environment. Running in a container also sandboxes modifications that the BugsInPy script wants to make (e.g., modifying `~/.bashrc`). While this image

is available in this registry [8], we suggest users build the image themselves rather than depending on this registry to remain available.

2. For each bug, 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 is not an automated process, and that makes it difficult for future users. We modified this 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 their own possibly conflicting requirements. Conda package repositories store packages containing pre-compiled binaries and metadata for each platform, so installing is rather quick.
3. The original BugsInPy scripts install all Python packages using Pip package manager. Pip can invoke the compiler to build dependencies from source [9] or download prebuilt binary files. 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 certain system-level dependencies that pip cannot automatically handle, such as libpng, freetype, or Tk, which are required for Matplotlib to function properly [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. Presumably the original 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. Building the environment from source can be quite costly, so we reuse environments when the Python package requirements and Python versions are exactly the same. This optimization helps reduce the time and resources required for environment setup, as the costly process of building environments from source can be bypassed. While it is tempting to use the same Conda environment for each project, 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. The BugsInPy framework 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, installing the dependencies sequentially allows us to reproduce the bugs accurately. However, this results in failed installations for projects that include the `-e git+https://...` syntax in their `requirements.txt` file, since they would get passed along as `pip install -e` and `pip install git+https://...`. This revised code ensures that each line from the requirements.txt file is properly processed and passed as an argument to the `pip install` command.

By utilizing the modified approach, we were able to resolve the installation issues for specific repository dependencies. For instance, the installation of the `luigi` library from a specific GitHub repository now proceeds as expected, as evidenced by the successful cloning and checkout of the specified commit.

To address this issue, we have opened a pull request in the original repository [11]. This fix is crucial, as it impacts the reproducibility of bugs in projects such as black, cookiecutter, keras, luigi, pandas, sanic, and thefuck.

III. Results

TABLE I
Reproduction of bugs in BugsInPy without modification

Project	Err	Bth-pass	Bth-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. With the original BugsInPy framework, we achieved the following bug reproduction rates Table I. In that table, the results we can get are:

- **Error**: Some part of the installation of the software environment needed to test the bug failed.
- **Both-pass**: Both versions pass; we would expect the buggy version to fail.
- **Both-fail**: Both versions fail; we would expect the fixed version to pass.
- **Expected**: The buggy version fails, and the fixed version passes. We consider *only* these bugs, "reproduced".

In that table, for each project, we show the raw count as well as percentage of outcomes for all bugs in that project. The last row shows the raw count as well as percentage of outcomes for all bugs in the dataset.

TABLE II
REPRODUCTION OF BUGS IN BUGSINPY, AFTER RESCUING

Project	Err	Bth-pass	Bth-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 many of the broken test cases, as shown in Table I.

Analyzing the specific project results, we can observe variations in the success rates. Some projects, such as Ansible, Cookiecutter, FastAPI, Httpie, Sanic, Thefuck, Tornado, and Tqdm, show a 100% success rate in reproducing and passing the bug tests for both the buggy and fixed versions. On the other hand, projects like Pandas, Keras, Scrapy, and Matplotlib have a lower success rate in reproducing and passing the bug tests. Note that our bugs are *repeatable*, which means that we can run our code twice and get the same result [7].

By achieving a high success rate in reproducing bugs and passing tests, the BugsInPy dataset demonstrates the effectiveness of the bug fixing process and highlights the overall quality of the codebase. With over 95% of bugs being successfully reproduced and passing the tests, researchers have more bugs at their disposal for evaluating fuzzing, automatic program repair repair, and other research techniques.

The success rate also indicates that the projects in the BugsInPy dataset have undergone rigorous testing and debugging processes. This level of thoroughness contributes to improved software quality and can inspire trust among users and stakeholders.

Table III presents the running time required to reproduce bugs in each project within the BugsInPy dataset, along with the respective containers. The provided running times are specific to the reproduction process on the given VM configuration, which had 8GB of RAM, 4 cores, 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.

Additionally, it is noteworthy that the "pandas" project has the most bugs registered in the dataset, with a total of 169 bugs out of the 501 bugs analyzed. The reproduction of bugs in the "pandas" project required a relatively longer running time, taking approximately 963 minutes. The extended time may be attributed to factors such as the complexity of the codebase, the number of bugs present in the project, and the specific characteristics of the bugs encountered during the reproduction process.

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

The inclusion of Python version information in the bug dataset is crucial for ensuring the reproducibility of tests executed for bug detection and fixing. Often, packages required by the software environment are only installable in a specific version of Python.

TABLE IV
PYTHON VERSIONS IN BUGSINPY DATASET

Python	Bugs	
3.6.9	31	6%
3.7.0	58	12%
3.7.3	50	10%
3.7.4	33	7%
3.7.7	9	2%
3.8.1	33	7%
3.8.3	287	57%
Total	501	100%

Table IV provides a summary of Python versions and their corresponding counts in the BugsInPy dataset. The percentages give us insights into the distribution of bugs across different Python versions in the dataset. The dataset certainly shows its age; Python 3.6 and 3.7 reached their official end-of-life and are not supported by CPython developers [12].

IV. DISCUSSION

A. What makes our reproduction easy?

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

1. **Automatation**: The `bugsinpy-testall` script provides an automated approach to reproducing and testing bugs in Python projects. The script streamlines the reproduction process, minimizes manual effort, and ensures we use a consistent procedure on each project. Note that these automation scripts must be carefully written and maintained. In particular, the original scripts did not have `set -e`, so some intermediate step might fail without alerting the user.
2. **Environment/Package manager**: Conda environment/package manager contributes to easy management of project dependencies. The key insight is that Conda can install packages to a local environment without interferring with global, system-wide packages. This makes it possible to define project-specific versions of libraries which would be normally managed by a platform-specific system-wide package manager.
3. **Lack of non-deterministic bugs**: None of the bugs in our dataset are race-conditions. Our scope is limited to constructing a reproducible software environment consistent with the original bug, and then the bug will show itself deterministically.

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

B. What makes our reproduction hard?

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

1. **Resource constraints during building**: Projects with a large number of bugs can pose challenges in terms of the time and resources required for bug reproduction. Sometimes, the software environment will involve a computationally-expensive step of building software from source. Reproducing and testing a significant number of bugs within limited resources may result in longer reproduction times and potential resource limitations.
2. **Lack of storage**: Our script ends up with quite a few Conda environments. These can be quite expensive to store, and we can't, for example, archive our environments in GitHub, due to space constraints.
3. **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 artifact authors

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

1. **`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.
2. **Archival storage**: Ensure that the artifact repository is archived in long-term storage such as Zenodo or FigShare, so that it does not become bit rot.
3. **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 automatically switch to the right Python version.

D. Threats to Validity

It may be possible that some of these bugs are reproducible, but the error is on our side. In particular, the authors do release data on the Python version, but none of their scripts make use of that information. Furthermore, we believe our efforts reflect an "average" user effort with limited resources, not a researcher with infinite time to reproduce every bit of the BugsInPy dataset.

It may be possible that our work is not reproducible for the following reasons:

1. Although we pin our Docker base images exact version, it is possible that the source location (DockerHub) stops hosting our image (e.g., goes out of business, ends free tier). In this case, one would need to change the base image, but it should still work, so long as that base image has Conda.
2. Conda package repositories can stop existing (e.g., if Anaconda goes out of business) or drop the old package versions we refer to. However, the definition of Conda packages describes how to build the packages from source. The package definitions are smaller than the package binaries, so they may remain longer.
3. The reproducers might not have enough computational resources to do the reproduction in a timely manner. However, we reuse Conda environments to minimize the computational cost. 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 standardized and automated approach provided by the `bugsinpy-testall` script, coupled with the use of Conda for dependency management, streamlines the bug reproduction process and enhances its ease.

The high success rate in reproducing bugs, with over 95% of bugs passing the tests, indicates the reliability and accuracy of the bug fixes in the dataset.

However, challenges still exist in reproducing certain bugs due to complex codebases, interdependencies, intermittent

nature, and resource constraints. Addressing these challenges requires further research, debugging techniques, and allocation of appropriate resources.

To improve the reproducibility of BugsInPy, it is recommended to enhance the documentation and description of software environments, making reproducibility tools more accessible and user-friendly. Collaboration within the Python community can also foster a supportive and collaborative environment for addressing challenges and enhancing the reproducibility process.

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] 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).

[3] 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).

[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] 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).

[6] 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).

[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] "Faustinoaq/bugsinpy-testall - docker image — docker hub." (), [Online]. Available: https://hub.docker.com/r/faustinoaq/bugsinpy-testall (visited on 07/17/2023).

[9] "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).

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

[11] "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).

[12] C. developers. "Status of python versions," Python Developer's Guide. (), [Online]. Available: https://devguide.python.org/versions/ (visited on 07/17/2023).

Appendix
Code, Data, and Reproducing

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

In the code:

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

To reproduce all bugs in httpie for example, run:

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

After which, the results will be in `bugsinpy-index.csv` This is the new result index.

See the `README.md` for more information.