Reproducing BugsInPy

Faustino Aguilar
dept. name of organization (of Aff.)
University of Panama
Panama City, Panama
email address or ORCID

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

Darko Marinov
Dept. of Computer Science
University of Illinois at Urbana Champaign
Urbana, IL
https://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.

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

- A buggy commit
- A fixed commit
- Test cases that indicate the bugs presence
- A `requirements.txt` file
- A script saying how to setup the environment

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*~**acm**˙**artifact**˙**2020** 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 use a provide 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, we suggest users build the image themselves rather than depending on this registry to remain available.

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 systme'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.

The original BugsInPy scripts install all Python packages using Pip. Pip is the default package builder and installer for Python. Pip can invoke the compiler to build dependencies from source or download prebuilt binary files. Projects do this for their own C source, however they don't do this for shared libraries, because the build system can be quite complex, and system

administrators would rather have one version of that shared library on the system managed by the system package manager. Therefore, some packages may require external setup that Pip is unable to do; for example, Matplotlib requires certain system libraries which Pip cannot install, so any bug whose environment depends on Matplotlibn will not be installable on a blank machine. 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 Debian and simply install those with Conda instead.

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.

The following is a pseudo-code representing the approach used to reproduce the BugsInPy dataset:

```
for each project in BugsInPy dataset:
  For each bug for that project:
    For version in {buggy, fixed}:
      Checkout the buggy version
      Env ID := hash(requirements.txt + python version)
      If no env with Env ID already exists:
        Use Conda to set up the env
      Activate env with Env ID
      Report environment errors
      Run tests
      Report test results
      Deactivate env
  done
done
```

A. Pip and PyPI work

III. Results

A. Why the percentage of success and failure for the reproducibility of bugs?

- The percentage of success and failure indicates the reproducibility of bugs in the given dataset.
- Success indicates that the bug was successfully reproduced and the test passed, verifying the correctness of the fix.
- Failure indicates that the bug could not be reproduced or the test failed, indicating that further investigation or fixes may be required.
- The percentage of success and failure provides insights into the overall quality and reliability of the codebase and the effectiveness of the bug fixing process.
- It helps identify projects with higher rates of successful bug reproduction, indicating better code quality and easier bug fixing processes.

Analyzing the success and failure percentages can inform the project maintainers about the areas that require further attention and improvement. It helps prioritize bug fixes, identify patterns in the types of bugs encountered, and allocate resources for improving the overall reliability and stability of the software.

The following is a table with the percentage of bug reproducibility for each project:

Project	Failed	OK	Total	% Failed	% OK
PySnooper	2	1	3	66.7%	33.3%
ansible	4	13	17	23.5%	76.5%
black	1	21	22	4.5%	95.5%
cookiecutter	2	2	4	50.0%	50.0%
fastapi	0	12	12	0.0%	100.0%
httpie	4	1	5	80.0%	20.0%
keras	25	35	60	41.7%	58.3%
luigi	32	0	32	100.0%	0.0%
matplotlib	29	0	29	100.0%	0.0%
pandas	59	86	145	40.7%	59.3%
spacy	2	8	10	20.0%	80.0%
thefuck	11	12	23	47.8%	52.2%
tqdm	4	4	8	50.0%	50.0%
youtube-dl	0	23	23	0.0%	100.0%
sanic	5	0	5	100.0%	0.0%
scrapy	17	21	38	44.7%	55.3%
spacy	2	8	10	20.0%	80.0%
tornado	1	14	15	6.7%	93.3%
tqdm	4	4	8	50.0%	50.0%
Total	**285**	**216**	**501**	**56.9%**	**43.1%**

The table provides an overview of the test results for each project. It includes the number of failed and OK tests, the total number of tests performed, and the percentage of failed and OK tests.

The total line aggregates the numbers from all projects and calculates the overall percentages of success and failure for the reproducibility of bugs.

The inclusion of Python version information in the bug dataset is crucial for ensuring the reproducibility of tests executed for bug detection and fixing. When developers encounter a bug, having access to the specific Python version used during its occurrence enables them to reproduce the bug in a controlled environment.

python_version	count	percentage
3.6.9	31	6.19%
3.7.0	58	11.58%
3.7.3	50	9.98%
3.7.4	33	6.59%
3.7.7	9	1.80%
3.8.1	33	6.59%
3.8.3	287	57.28%
Total	501	100%

The table provides a summary of Python versions and their corresponding counts in the bug dataset mentioned in the "bugsinpy" paper. The dataset contains bug information for various Python repositories, with each bug having a corresponding "bug.info" file specifying the Python version.

The table includes the Python version, the count of bugs associated with each version, and the percentage of bugs represented by each version out of the total count. The

percentages give us insights into the distribution of bugs across different Python versions in the dataset.

Analyzing the table, we observe that the majority of bugs (57.28%) are reported in Python version 3.8.3, followed by version 3.7.0 (11.58%). Versions 3.7.3, 3.6.9, and 3.8.1 account for approximately 9.98%, 6.19%, and 6.59% of the bugs, respectively. Additionally, versions 3.7.4 and 3.7.7 contribute to 6.59% and 1.80% of the bugs, respectively.

These percentages provide valuable insights into the distribution of bugs among different Python versions in the dataset. By referring to the "bugsinpy" paper and the associated bug.info files, further analysis can be performed to understand any patterns or trends related to specific Python versions and their corresponding bug occurrences.

By using the same Python version specified in the bug, developers can accurately replicate the conditions under which the bug manifested. This consistency in the Python environment helps in understanding the root cause of the bug and facilitates the debugging process. It allows developers to examine the code, libraries, and dependencies associated with that particular Python version, increasing the likelihood of identifying and resolving the issue effectively.

Reproducibility plays a significant role in software development, particularly in bug fixing and testing. Having access to the exact Python version used during the bug occurrence enhances the accuracy and reliability of the testing process. Developers can execute the same code with the same environment, ensuring that any fixes or improvements applied can be tested and validated consistently.

In summary, the inclusion of Python version information in the bug dataset not only provides insights into the distribution of bugs across different versions but also enables developers to reproduce and investigate the bugs more effectively. This contributes to improved bug detection, diagnosis, and ultimately, the development of more reliable and stable software systems.

IV. Discussion

A. What makes our reproduction easy?

- The `bugsinpy-testall` script provides a standardized and automated approach to reproduce and test bugs in Python projects. It automates the process of setting up the environment, checking out specific versions of projects, compiling code (if required), and running tests.
- The use of Conda allows for easy management of dependencies and ensures consistent environments across different systems.
- The script maintains logs of the test results and outputs them to `~/projects/output.csv`, providing a centralized record for analysis and tracking the progress of bug reproduction.

B. What makes our reproduction hard?

- Reproducing bugs can be challenging due to various factors, including complex codebases, interdependencies, and compatibility issues between different versions of libraries and tools.
- In some cases, bugs may be intermittent or require specific conditions to manifest, making it difficult to consistently reproduce them.
- Projects with a large number of bugs or a high failure rate pose additional challenges in terms of time and resources required for reproduction.

C. Recommendations to BugsInPy users

D. Recommendations to artifact authors

E. Threats to validity

Is it possible that our reproduction is not working when it should be?

Is it possible that our reproduction won't be reproducible by others?

V. Conclusion

Does our study support the conclusion?

Future directions for research?

- Improve reproducibility of BugsInPy
- Improve description of software environments
- Make reproducibility tools easier to use

References

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).

Appendix
Code, Data, and Reproducing

A snapshot of the latest state of this code can be found at: ...(ZenodoDOI).

A rolling release of the code can be found at: ...(GitHub).

In the rolling release or snapshot:

- `data` holds a machine-readable view of the data, split across several files.
- `spack/spack.lock` contains the Spack environment in which this experiment was run.

To reproduce this paper, run:

```
#␣command
output
```

After which, the results will be here:

- `reports/main.pdf` This is the actual paper.
- `raw_data` This is the raw data.