Reproducing BugsInPy

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

4th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

5th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

6th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

***Abstract*—We present our experience on replicating a bug dataset for the Python programming language. The bug dataset provides some information about the software environment, but this environment decays quickly into something uninstallable. We assess the reproducibility over time and improve the reproducibility of the dataset.**

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

I. Introduction

BugsInPy is a curated dataset of real-world bugs in large Python projects. It provides a database of known bugs in Python code, along with the test cases that reveal the bugs. BugsInPy is intended to be used by researchers to develop and evaluate software defect detection tools.

BugsInPy exists to provide a high-quality benchmark for testing and debugging tools targeting Python programs. By providing a large dataset of known bugs, BugsInPy helps researchers to test their tools on a variety of different bugs. 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:

- The bug's location in the code
- The bug's type
- The bug's expected behavior
- The test cases that reveal the bug
- The tools that have been used to detect the bug

BugsInPy also 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.

The provided text follows a logical and comprehensive reading, covering various aspects related to the BugsInPy dataset, the methodology used, Docker-based reproducibility, the bugsinpy-testall script, Conda package management, and the results obtained from the bug reproduction process. The text explains the purpose of the dataset, its components, and how it can be used for testing and debugging tools targeting Python programs.

The Dockerfile and its breakdown are described in a clear and step-by-step manner, highlighting how it improves reproducibility by providing a controlled execution environment for Python projects. The bugsinpy-testall script is provided, which outlines the automated process of reproducing and testing the BUgsInPy dataset. The inclusion of success and failure percentages in the results section helps to analyze the reproducibility of bugs across different projects and Python versions.

The text also discusses the significance of Python version information in the bug dataset and how it aids in reproducing and investigating bugs accurately. It emphasizes the importance of using the same Python version to ensure reproducibility and facilitate the identification and resolution of issues. The advantages of reproducibility and the impact it has on the testing and debugging process are highlighted.

The discussion section covers both the ease and challenges of bug reproduction. It acknowledges that reproducing bugs can be complex and highlights factors such as code complexity, interdependencies, and intermittent nature of bugs that can make the process challenging. It also mentions the potential difficulties posed by projects with a large number of bugs or a high failure rate.

II. Methodology

By using Docker, we can ensure that your Python project runs consistently across different environments, making it more

reproducible. It eliminates potential conflicts with system dependencies and provides a self-contained execution environment.

```
FROM continuumio/miniconda3:23.3.1-0
MAINTAINER faustinoaq <faustino.aguilar@up.ac.pa>

RUN useradd -ms /bin/bash user
RUN apt-get update
RUN apt-get -y install git nano build-essential

WORKDIR /home/user

COPY . /home/user/BugsInPy

RUN chown -R user:user /home/user/BugsInPy

USER user

RUN echo "export PATH=$PATH:/home/user/BugsInPy/framework/bin" >> /home/user/.bashrc

CMD ["/bin/bash"]
```

This Dockerfile sets up a Docker image for running a Python project using Miniconda and improves reproducibility by providing a controlled environment for project execution. Let's break it down step by step:

1. `FROM continuumio/miniconda3:23.3.1-0`: This line specifies the base image for the Dockerfile, which is `continuumio/miniconda3` with the version tag `23.3.1-0`. Miniconda is a minimal version of Anaconda, which provides a Python environment and package management capabilities.
2. `MAINTAINER faustinoaq <faustino.aguilar@up.ac.pa>`: This line specifies the author/maintainer of the Dockerfile.
3. `RUN useradd -ms /bin/bash user`: This command creates a non-root user named "user" with `/bin/bash` as the default shell.
4. `RUN apt-get update`: This command updates the package lists on the container to ensure the latest versions are available for installation.
5. `RUN apt-get -y install git nano build-essential`: This line installs additional packages (`git`, `nano`, and `build-essential`) using `apt-get`. These packages are commonly used for version control, text editing, and building software.
6. `WORKDIR /home/user`: This line sets the working directory inside the container to `/home/user`.
7. `COPY . /home/user/BugsInPy`: This command copies the entire current directory (where the Dockerfile is located) into the container at `/home/user/BugsInPy`.
8. `RUN chown -R user:user /home/user/BugsInPy`: This line changes the ownership of the `/home/user/BugsInPy` directory to the non-root user "user" created earlier. This ensures that the user has the necessary permissions to access and modify the files within the directory.
9. `USER user`: This instruction switches the container to run as the non-root user "user". This is a security best practice to limit potential vulnerabilities.
10. `RUN echo "export PATH=$PATH:/home/user/BugsInPy/framework/bin" >> /home/user/.bashrc`: This line appends the path `/home/user/BugsInPy/framework/bin` to the `PATH` environment variable in the user's `.bashrc` file. This allows the user to execute scripts or binaries located in that directory directly from the command line.
11. `CMD ["/bin/bash"]`: This is the default command that will be executed when the container starts. It launches the Bash shell, providing an interactive terminal for the user.

By using this Dockerfile, you can build a Docker image that encapsulates your Python project and its dependencies. This approach improves reproducibility by ensuring that the project runs in a controlled and isolated environment. The specified Miniconda version ensures consistency, and the installed packages and their versions are locked in the image. The user configuration helps avoid running the application as the root user, enhancing security and preventing accidental modifications to the host system.

To use this Dockerfile for your Python project, follow these steps:

1. Place the Dockerfile in the root directory of your project.
2. Build the Docker image by running the following command in the same directory as the Dockerfile:

   ```
   docker build -t myproject .
   ```

   This command builds the Docker image using the Dockerfile and tags it with the name `myproject`. Make sure you have Docker installed on your system.
3. Once the image is built, you can run a container based on it using the following command:

   ```
   docker run -it myproject
   ```

   This command starts an interactive container based on the `myproject` image. You will be dropped into a Bash shell inside the container, where you can execute commands and run your Python project.

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

```
echo "test 'ok' means reproduced successfully, buggy version fa
echo "test 'fail' means unable to reproduce, error happened or
echo "See full tests logs in ~/logs.txt"
echo "See results in ~/projects/output.csv"
# Iterate over the projects
for project in $projects; do
  # Get the number of bugs in the project
  # More project handling...

  # For each bug, execute the test
  for bug in $(seq $start $finish); do
    # Check if bug is already tested
    grep "$project,$bug," ~/projects/output.csv
    # More checks...

    # Test buggy (0) version
    # Checkout the buggy version
    # Set up the Python environment using conda
    # Compile and test the code
```

```
    # Handle failures and update output.csv

    # Test fixed (1) version
    # Checkout the fixed version
    # Compile and test the code
    # Handle failures and update output.csv

    # Deactivate the Python environment
  done
done
```

The `bugsinpy-testall` script automates the execution of the BugsInPy dataset, which contains bugs in various Python projects. The script reproduces the bugs, executes tests, and records the results. It enhances the reproducibility of Python projects by providing a standardized process for reproducing and testing bugs in different projects.

Here's a summary of how the script works:

1. The script takes command-line arguments to control its behavior. It provides options to display help, perform cleanup, and specify projects or ranges of bugs to reproduce and test.
2. It creates a `~/projects` directory to store the output and logs.
3. The script iterates over the specified projects or all projects in the `~/BugsInPy/projects` directory.
4. For each project, it determines the range of bugs to reproduce and test.
5. It executes the tests for each bug by performing the following steps:
   a. Checks if the bug has already been tested and skips it if so.
   b. Sets up the environment for testing the buggy (0) version:
      - Uses `bugsinpy-checkout` to checkout the buggy version.
      - Activates the proper Python environment using Miniconda (specified in the `bugsinpy_bug.info` file).
      - Compiles the project (if required) and runs the tests using `bugsinpy-test`.
   c. Checks if the buggy version fails the test. If it does, it proceeds to test the fixed (1) version:
      - Uses `bugsinpy-checkout` to checkout the fixed version.
      - Compiles the project (if required) and runs the tests.
   d. Records the test results (`ok` if the fixed version passes the test, `fail` otherwise) in `~/projects/output.csv`.
6. The script deactivates the Conda environment and repeats the process for the next bug in the project.

The `bugsinpy-testall` script improves reproducibility by providing a standardized and automated approach to reproduce and test bugs in Python projects. It ensures that bugs are tested consistently across different projects, enabling easier verification of bug fixes and facilitating the replication of test results. By logging the output and test results, it helps track the status of bug reproduction and provides a central record for analysis and further investigation.

Now, let's answer your additional questions:

A. *`bugsinpy-testall` script*

The `bugsinpy-testall` script automates the process of reproducing and testing bugs in Python projects. It iterates over specified projects or all projects in the `~/BugsInPy/projects` directory. For each project, it determines the range of bugs to reproduce and test. The script checks if a bug has already been tested and skips it if so. It sets up the environment for testing the buggy (0) version, runs the tests, and checks if the bug fails. If it fails, it proceeds to test the fixed (1) version and checks if it passes the test. The results are recorded in `~/projects/output.csv`.

B. *Conda package manager*

Conda is a cross-platform package management system and environment management system. It allows users to create and manage isolated environments with specific versions of software packages. Conda packaging works by creating packages containing pre-compiled binaries and metadata. These packages can include Python libraries, dependencies, and other software tools. Conda allows for the creation of custom environments with specific package versions, enabling reproducibility by ensuring consistent software dependencies across different systems.

C. *Pip and PyPI work*

Pip is the default package installer for Python, which allows users to install and manage Python packages from the Python Package Index (PyPI) and other package repositories. PyPI is the official software repository for Python packages. When using Pip, it searches for packages on PyPI based on package names and versions specified in the requirements file or command-line arguments. Pip then downloads the package and its dependencies from PyPI and installs them into the Python environment. PyPI serves as a central repository for sharing and distributing Python packages, making it easier for developers to package and distribute their software for others to use.

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

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.