Analyzing Python Project Requirements: Accuracy, Updates, and Third-Party Generators

Bithy Das
Master of Computer Science
Lakehead University
Thunder Bay, ON
bdas@lakeheadu.ca

Khyati Patel
Master of Computer Science
Lakehead University
Thunder Bay, ON
kpate101@lakeheadu.ca

***Abstract*—Effective management of project dependencies is crucial for ensuring the replicability and success of software development initiatives. In the realm of Python programming, the 'requirements.txt' file serves as a central tool for specifying project dependencies. However, maintaining this file's accurate and up-to-date dependency information remains challenging, leading to compatibility issues and hindered project replication.**

This paper investigates the complexities surrounding Python project requirements, particularly in the context of replicating projects sourced from GitHub repositories. The primary concern lies in the inconsistency and potential inaccuracy of dependencies listed in requirements.txt, exacerbated by outdated files as projects evolve. To address these challenges, third-party tools like pipreqs and pipfreeze have emerged as potential solutions to automate the generation of requirements.txt files. Through a comprehensive evaluation of pipreqs and pipfreeze, we found how pipreqs perform compared to pipfreeze, capturing required project dependencies. Even though pipreqs appears to be more promising, it is far from flawless.

***Index Terms*—Project dependencies, requirements.txt, pipreqs**

I. Introduction

In the dynamic landscape of software development, Python has come to prominence as the foundational programming language, developing a prominent open-source community that contributes to a wide variety of projects hosted on platforms like GitHub [2]. Replicating and expanding on these initiatives has become a regular occurrence, allowing other disciplines to progress. However, identifying and maintaining project dependencies appropriately via the requirements.txt file is a considerable struggle.

This paper analyzes the critical issue of Python project requirements, focusing specifically on the difficulties encountered by developers when replicating projects sourced from GitHub repositories [3]. The primary concern lies in the inconsistency and possibility of the inaccuracy of the listed dependencies in the requirements.txt file, often leading to compatibility issues and hindering successful project replication. This problem is compounded by the fact that such files have a tendency to go outdated as the project continues to develop, which further exacerbates the difficulties faced by individuals.

To address these challenges, various third-party tools have become available to be potential solutions. Among these, we examine two prominent options: pipreqs and pipfreeze [4]. These tools are designed to automate the generation of 'requirements.txt' files by analyzing project code and extracting its dependencies. In order to maintain consistency across various development environments, it is imperative that we streamline the process of accurately capturing dependencies.

Our research is motivated by the desire to enhance the reliability and efficiency of replicating Python projects by bettering the accuracy of their dependency management. We have thoroughly examined both pipreqs and pipfreeze, evaluating their effectiveness in generating reliable requirements.txt files [5] [6]. We determined that pipreqs is not providing a satisfying solution through a series of experiments and comparisons, suggesting higher quality accuracy and reliability in gathering required project dependencies.

As we proceed, subsequent sections of this paper are structured to provide an in-depth exploration of our findings. The "Related Work" part puts into perspective our research within the larger context of software dependency management, whereas the "Dataset" section explains the procedure we used for collecting a representative sample of Python GitHub projects. The "Methodology" part describes our approaches to answering each research question, and the "Results" section reveals the results of our studies. Our discussion and conclusion integrate the findings and consequences of our research, requiring an enhanced understanding of Python project specifications and supervision.

Research Questions:

To investigate these issues, we determine the following research questions:

RQ1: What is the frequency of updates of the requirements.txt files given in Python GitHub projects, and how do these aspects affect project usability?

RQ2: Are there any third-party libraries that can effectively generate requirements.txt files that precisely capture the needed dependencies of Python GitHub projects?

RQ3: How do the accuracy and utility of pipreqs and pipfreeze differ in the context of generating requirements.txt files for Python projects?

RQ4: Why do pipreqs fail to generate requirements.txt files for some Python projects, and what are the most prevalent reasons or restrictions for these errors?

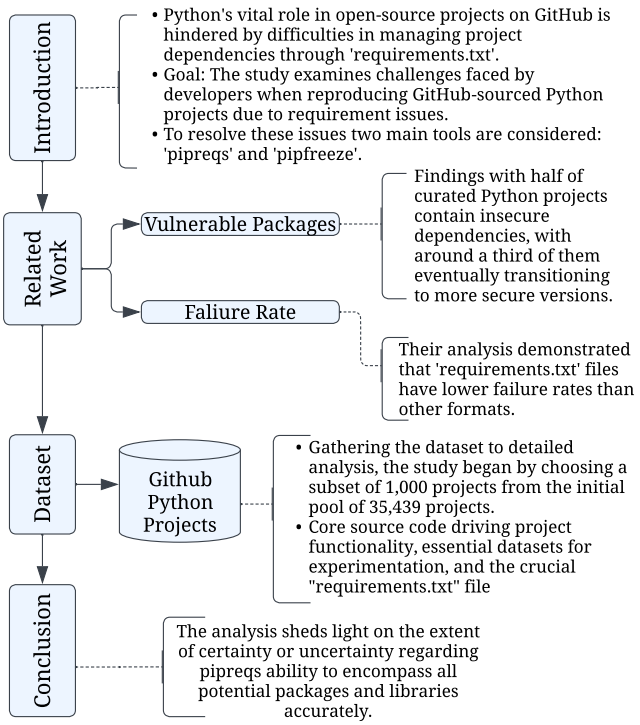

Fig. 1. A schematic flow of this experiment paper

II. Related Work

We conducted a comprehensive quantitative analysis to identify the existence of susceptible packages. We observed how software adjusts in response to identified vulnerabilities and the time frame during which vulnerabilities are present, as referenced by our benchmark study, they found The diversity of modern software necessitates using packages for efficient functionality rather than custom coding. However, many packages lack robust security, leading to vulnerabilities inherited by projects. While security patches are released, integrating newer versions isn't common, resulting in insecure software. Their study examines vulnerable packages in open-source Python software and their response to discovered flaws. Among curated Python projects, over half have insecure dependencies and roughly a third transition to safer versions [9]. The speed at which developers detect and rectify flaws, along with the promptness of app updates incorporating these fixes, stand as critical factors in addressing package issues. Our analysis revealed that requirements.txt files exhibit lower failure rates compared to alternative formats. Additionally, a significant number of errors stem from inadequate module dependencies. Hence, we recommend explicitly stating dependencies and specifying version pins. A wide-ranging study conducted by [10] on Jupyter Notebooks reveals their widespread adoption across various communities in both scientific and industrial domains. These notebooks offer a platform for creating literate programming documents, seamlessly integrating code, text, and executed outputs alongside visualizations and other multimedia elements.

Notably, Jupyter Notebooks are celebrated for their inherent self-documentation capabilities, and their contribution to result reproducibility, both recognized as significant merits. However, it's worth noting that criticisms have surfaced, suggesting that certain usage patterns of notebooks can lead to unexpected outcomes, foster less-than-optimal coding practices, and present challenges in reproducing results. In each project category (ML and traditional), they computed the update ratio. This ratio indicates the fraction of library updates within that category compared to all updates across the entire project, encompassing libraries from all categories. This observation served as a catalyst for our comprehensive study [14].

III. Dataset

Before beginning our research, we started by gathering the fundamental information from a renowned study titled "Curating GitHub for engineered software projects" [1]. This initial integration of insights set the stage for our subsequent endeavors. Guided by our research objectives, we strategically focused on Python projects, aligning our investigation with the programming language of interest. Leveraging the capabilities of the GitHub API, we executed a meticulous validation process to identify projects equipped with the essential "requirements.txt" file—a key element that outlines the prerequisites necessary for the project's optimal functionality.

The culmination of these strategic steps resulted in the creation of our ideal dataset. This thoroughly controlled repository only featured projects that included the must-have "requirements.txt" file. In the vast scope of Python, the data we collected had a notable count of 35439 projects. This comprehensive dataset served as a rich tapestry, interweaving insights from diverse projects and shedding light on the intricate nuances of the Python landscape.

Transitioning from dataset acquisition to in-depth analysis, we commenced our research by selecting a small dataset of only 1000 projects from those 35439 projects, serving as the bedrock upon which our upcoming studies were achieved. These project identifiers, presented as "username/project_name," provided distinct markers amidst the expansive GitHub repository landscape. We created and executed a comprehensive codebase to ensure a complete and thorough examination of each of these projects. This code proficiently selected one project at a time from the aforementioned dataset and proceeded to extract the entire contents of the selected project from the GitHub repository. These contents included an overall combination of essential components that, when combined, permitted a thorough understanding and eventual replication of the project.

The detailed information gathered for each project was comprehensive. They included but were not limited to, the primary source code underlying the project's functionality, the associated datasets necessary for experimentation, the essential "requirements.txt" file required for seamless setup, and a

comprehensive readme documentation that provided detailed information on the project's features and workings.

When taken as a whole, the dataset produced as a consequence of this thorough effort demonstrated to be a potent knowledge base. It was a collection of 1000 different projects totaling 8.60 gigabytes in size. This massive collection was made up of 191,150 separate files, each of which contributed to the numerous levels of information contained therein. These files were precisely categorized throughout 49,403 folders, confirming the sophisticated and expansive nature of our cultivated dataset. This dataset stood as a testament to the comprehensive nature of our research initiative and the invaluable insights it harbored for the broader academic and practitioner community.

IV. Methodology

Our research focuses on automating the creation and maintenance of the requirements.txt file for Python projects. The purpose of this project is to investigate the challenges faced by open-source Python projects and their implications for the community [1].To facilitate this study, we gather data from GitHub repositories. However, the selection process for these repositories is guided by a crucial requirement that the repositories must contain a requirements.txt file detailing all the Python dependencies used within the project [9]. This comprehensive approach allows us to gain valuable insights into the landscape of open-source Python projects and their impact on the community. This study follows a systematic approach to finding out these issues, and our analysis will be shown further ahead in the paper.

A. Project Selection and Renaming:

To ensure a representative sample, we employed a rigorous selection process wherein a diverse set of 1000 Python projects was randomly chosen for comprehensive analysis. Each of these projects was initially assigned arbitrary alphanumeric titles as Fig. 3. However, for the sake of clarity and facilitating precise project tracking throughout the study, we subsequently adopted a sequential renaming convention, assigning the projects unique names ranging from 0 to 999 as Fig. 4.

By implementing this naming convention, the iterative and processing phases became notably streamlined and more manageable. Furthermore, to exemplify the logic employed in editing 1000 folders within a Windows system, each project was correspondingly organized into individual folders bearing sequential numerical names, ensuring a systematic and organized structure for the analysis.

B. Preservation of Original Requirements Files:

The original requirements.txt files, as provided by the developers, formed a crucial component of our analysis framework Fig. 5. To ensure the integrity of these files, a systematic approach was adopted wherein they were methodically renamed to 'OldRequirements.txt' for each respective project as Fig. 6. This strategic procedure serves to preserve the initial state of project dependencies, allowing for unaltered reference and facilitating meaningful comparisons.

The Python code we made uses the 'os' library to navigate through a specified base directory containing project folders. For each project folder, it first checks if the path corresponds to a directory. If it does, the code constructs the paths for both the original 'requirements.txt' file and the desired 'OldRequirements.txt' file. It then attempts to rename the original file to the new name. This process is carried out for every project folder in the base directory, ensuring that each 'requirements.txt' file is methodically renamed to 'OldRequirements.txt' to maintain the authenticity of the initial project dependency records. If any errors occur during this renaming process, the code handles them and continues with the rest of the folders.

C. Utilizing Pipreqs for Requirement Generation:

Next, we utilized the pipreqs tool to generate new requirements.txt files for each project. The pipreqs package scanned the source code of each project, identified imported packages, and compiled requirements files containing the names and versions of these packages. This ensured the maintenance of an updated list of packages required for each project.

A script was executed to determine the count of projects that were lacking a "requirement.txt" file, and this task was accomplished using the "pipreqs" library. This library, commonly employed in Python development, automatically generates a list of a project's dependencies based on the imported modules within the codebase. To ensure that the requirements.txt file accurately reflects the project's dependencies, the methodology leverages the pipreqs tool. This external command-line utility examines the source code of the given Python project folder to automatically determine the required external libraries and packages. The –force flag is used to ensure that the generated requirements.txt file overwrites any existing version. By running the script, the number of projects without this crucial requirement file could be identified, playing a significant role in managing and reproducing the necessary dependencies for various Python projects.

1) Motivation: Pipreqs streamlines the creation of requirements files by analyzing source code and intelligently discerning essential packages while excluding unnecessary ones, thus simplifying the requirement-building process [8]. This library takes on the role of automatically generating the Requirements file for each project [7], facilitating the identification of libraries employed within the .py scripts of these projects. Furthermore, pipreqs has been extended to capture widely utilized Python libraries, broadening the scope of the project's data analysis to encompass a more comprehensive realm of research.

D. Identification of Missing Requirement Files:

A custom script was developed to methodically evaluate the efficacy of the pipreqs library in generating 'requirement.txt' files. This pivotal step unearthed a notable vulnerability within the pipreqs library, wherein it failed to construct requirement files for 447 out of the 1000 sampled

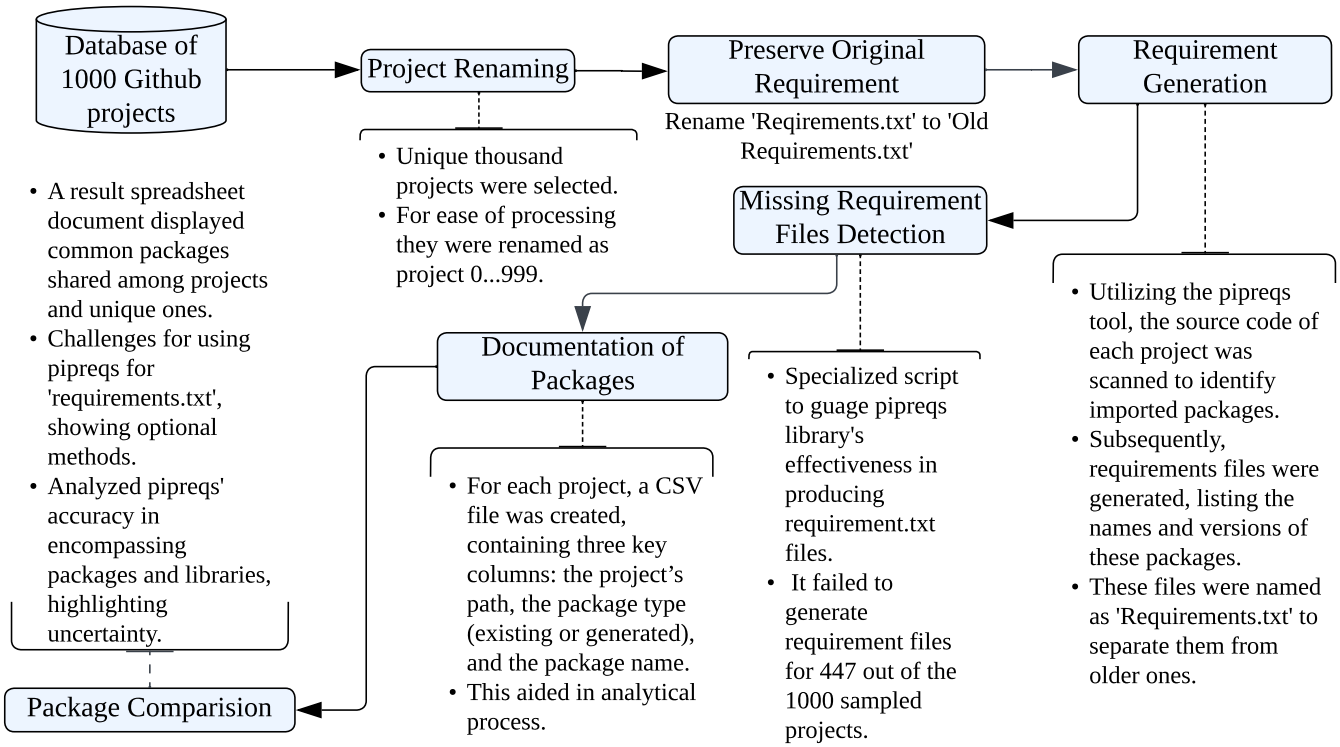

Fig. 2. Methodology flow process steps

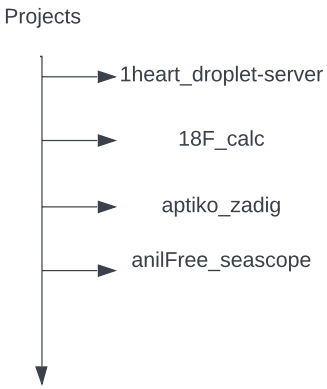

Fig. 3. Folder Structure Before Renaming

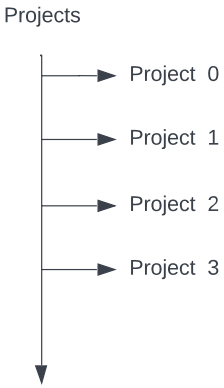

Fig. 4. Folder Structure After Renaming

projects. This initial revelation lays a pivotal cornerstone for the subsequent analysis, indicating an inherent inaccuracy in the dependency lists within those files. The implications of this issue are multifaceted: these projects might encounter deficiencies in required packages, leading to potential errors during execution. Moreover, the oversight could extend to security concerns if vulnerable packages are not appropriately cataloged. In essence, our findings underscore the potential need for enhancements within the pipreqs library, aimed at fostering the creation of dependable 'requirement.txt' files. Such enhancements are vital for ensuring the steadfast and secure development of projects.

After that, we did a broad research about the error that we found while running pipreqs to get requirement.txt files, and it was also analyzed generally from other people's findings that were reported in Stackoverflow or other Python community sites. On the other hand, pipreqs solely captures local modules within third-party packages and needs to request the latest version from PyPI, which leads to an increase in its time-related overhead [8]. However, differing from pipreqs, which promptly halts its operation upon encountering a SyntaxError stemming from incompatible Python versions during program

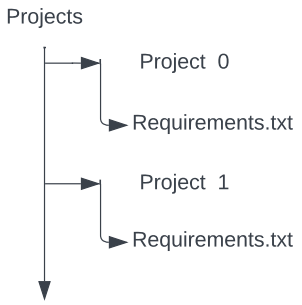

Fig. 5. Requirements.txt Prior to Rename

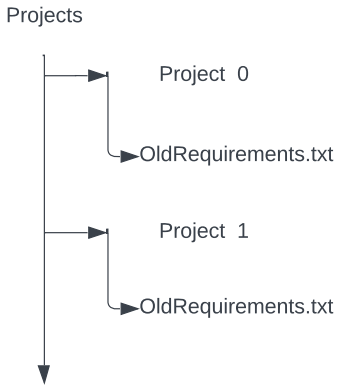

Fig. 6. Requirements.txt now renamed to 'OldRequirements.txt'

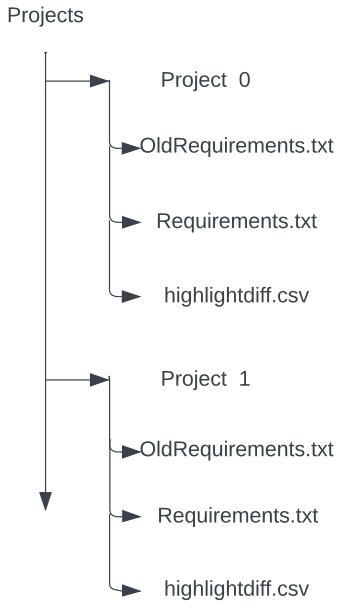

Fig. 7. Folder Structure after generating new 'requirements.txt' using PipReqs

analysis.

E. Dataset Documentation of Packages:

A comprehensive documentation strategy was devised to capture and analyze the dependencies. For each project, a CSV file was created containing three key columns: the project's path, the package type (existing or generated), and the package name. This meticulous documentation enabled a granular evaluation of the packages, enabling a nuanced understanding of both pre-existing and newly generated dependencies.

We developed a Python script that defined a function called "compare text files", which aimed to compare the contents of two text files, namely 'OldRequirements.txt' and requirements.txt, located within a specified project folder. The difflib library was employed to analyze the differences between these files on a line-by-line basis. The dissimilarities were then structured into rows for a CSV file named 'highlightdiff.csv' as Fig.7. Each CSV row included the textual content of the line, a type indicator ('Existing package' or 'Generated package'), and the corresponding package name. Our script ensured the existence of the required files, read their content, performed the comparison, processed the differences, and wrote the findings into the CSV file. This script proved to be valuable for identifying discrepancies between package lists or dependencies in diverse contexts.

F. Compared Package Names:

We created a CSV document to display common packages shared among projects and those unique to specific ones. Critical choices about the adoption or standardization of specific packages were facilitated by this study of comparisons.

We created code by importing essential libraries such as os, re, difflib, and csv, which are required for various functionalities like differentiating text between two files, The "re" module of the Python standard library is for regular expressions which helped us process these larger text which is mixed of special characters such as '=', '.' and. Its main purpose is to provide a search for which a string and a regular expression are required.

We proceeded to define several utility functions that serve specific purposes. The function normalizes _package _name(package _name) was designed to take a package name and transform it by retaining only alphabetical characters using regular expressions. The function read _package _(filename) reads a file containing package names, converts them to lowercase, and normalizes them, resulting in a set of normalized package names.

The function compare_package _lists(OldRequirements.txt, requirements) was responsible for comparing two sets of normalized package names. It calculated the unique packages present in each list and identified the common packages shared between them. Furthermore, the function finds _differences(text1, text2) utilized the difflib library to generate a diff-like output indicating the distinctions between two sets of normalized package names.

The main operational logic was encapsulated within the main() function. We initially specified the project directory and

output filename. As the script iterated through each project within the designated directory, it verified the existence of both 'OldRequirements.txt' and requirements.txt files. Upon confirming their presence, the script extracted and normalized package names from both files. It then employed the utility functions to compute various statistics, such as package counts and differences. These outcomes were subsequently compiled and written into a CSV file. The underlying objective of this script was to facilitate the management of software dependencies across multiple projects by highlighting differences in package lists.

1) Common Packages: When a package appears in both the 'OldRequirements.txt' and 'requirements.txt' files, it indicates that the package is recognized and can be reliably installed by pip. The fact that the package is consistent in both lists signifies a level of certainty that it is widely available and compatible.

2) Differences: When there are packages in 'OldRequirements.txt' that are missing from 'requirements.txt' or vice versa, it suggests uncertainty or inconsistency in package availability. That indicates situations where certain packages might not be easily discoverable by pip or might have varying names, versions, or compatibility constraints. These discrepancies highlight a potential limitation in pip's ability to gather all possible packages accurately.

3) Highlighting Uncertainty: The presence of differences in the comparison signaled that pip might not cover the entire universe of available packages or could encounter difficulties in resolving package names consistently. This could be due to variations in package naming, different package sources, or potential version incompatibilities. The larger the number of differences, the greater the uncertainty about pip capturing the full spectrum of available packages.

Overall we compared 'OldRequirements.txt' and 'requirements.txt' and observed the differences and commonalities. The analysis sheds light on the extent of certainty or uncertainty regarding pipreqs ability to encompass all potential packages and libraries accurately as table I. It underscored the challenges that might arise when relying solely on pipreqs for generating 'requirements.txt' files, and it suggested that supplementary methods or tools might be necessary to ensure comprehensive package coverage in certain cases.

TABLE I
PACKAGE COMPARISON OF 1000 PROJECTS, SAMPLE SIZE OF 40. C1-PROJECT PATH, C2-EXISTING PACKAGES, C3-GENERATED PACKAGES, C4-UNIQUE TO EXISTING PACKAGES, C5-UNIQUE TO GENERATED PACKAGES, C6-COMMON PACKAGES

C1	C2	C3	C4	C5	C6
G:/ReqProject/gitprojects/project/001	1	3	1	3	0
G:/ReqProject/gitprojects/project/002	6	4	2	0	4
G:/ReqProject/gitprojects/project/003	2	5	0	3	2
G:/ReqProject/gitprojects/project/005	12	9	5	2	7
G:/ReqProject/gitprojects/project/006	2	2	1	1	1
G:/ReqProject/gitprojects/project/008	1	3	0	2	1
G:/ReqProject/gitprojects/project/009	3	3	2	2	1
G:/ReqProject/gitprojects/project/010	1	11	1	11	0
G:/ReqProject/gitprojects/project/011	8	3	7	2	1
G:/ReqProject/gitprojects/project/012	3	1	3	1	0
G:/ReqProject/gitprojects/project/014	8	3	7	2	1
G:/ReqProject/gitprojects/project/016	7	4	5	2	2
G:/ReqProject/gitprojects/project/017	4	2	2	0	2
G:/ReqProject/gitprojects/project/020	7	2	6	1	1
G:/ReqProject/gitprojects/project/021	1	2	0	1	1
G:/ReqProject/gitprojects/project/023	2	3	1	2	1
G:/ReqProject/gitprojects/project/024	5	4	1	0	4
G:/ReqProject/gitprojects/project/025	6	5	1	0	5
G:/ReqProject/gitprojects/project/026	14	2	12	0	2
G:/ReqProject/gitprojects/project/027	7	4	4	1	3
G:/ReqProject/gitprojects/project/029	1	1	0	0	1
G:/ReqProject/gitprojects/project/031	0	1	0	1	0
G:/ReqProject/gitprojects/project/032	4	7	1	4	3
G:/ReqProject/gitprojects/project/033	1	7	0	6	1
G:/ReqProject/gitprojects/project/036	22	9	14	1	8
G:/ReqProject/gitprojects/project/038	6	8	3	5	3
G:/ReqProject/gitprojects/project/039	3	1	3	1	0
G:/ReqProject/gitprojects/project/040	4	1	3	0	1
G:/ReqProject/gitprojects/project/041	9	4	7	2	2
G:/ReqProject/gitprojects/project/042	3	3	0	0	3
G:/ReqProject/gitprojects/project/043	4	6	1	3	3
G:/ReqProject/gitprojects/project/044	4	3	1	0	3
G:/ReqProject/gitprojects/project/046	5	3	2	0	3
G:/ReqProject/gitprojects/project/049	10	1	9	0	1
G:/ReqProject/gitprojects/project/052	17	1	16	0	1
G:/ReqProject/gitprojects/project/054	7	6	5	4	2
G:/ReqProject/gitprojects/project/056	2	3	0	1	2
G:/ReqProject/gitprojects/project/057	13	2	11	0	2
G:/ReqProject/gitprojects/project/058	3	2	2	1	1
G:/ReqProject/gitprojects/project/060	2	5	0	3	2
G:/ReqProject/gitprojects/project/063	3	5	0	2	3

V. RESULTS

RQ1: What is the frequency of updates of the requirements.txt files given in Python GitHub projects, and how do these aspects affect project usability?

The practical value of Python GitHub projects may be greatly improved by being aware of how frequently requirements.txt files are updated [3]. The frequency of changes has a direct influence on the experience of developers and contributors, impacting their ability to replicate development environments and successfully collaborate. Recognizing the relationship between update frequency and project usability may shed light on best practices for maintaining accurate dependencies.

We investigated a variety of Python projects that were stored on GitHub by methodically examining them. Our dataset included a diverse range of fields of study and project sizes, representing a cross-section of the Python open-source community. We looked at the commit history for every project and tracked down the modifications made to the requirements.txt file. We documented the total number of commits made to the repository as well as the occasions where the requirements.txt file was modified.

Our research uncovers surprising insights regarding the frequency of updates to requirements.txt files and their influence on project usability. While many Python projects populate the requirements.txt file throughout the initial setup and then make no additional modifications, such an approach may result in diminished project usefulness as dependencies develop. On the other hand, projects that are dedicated to keeping their requirements.txt file up to date show a beneficial commitment

```
Total commits  7  --  zemlanin/Libr
Changed req  0
Total commits  1766  --  maraujop/django-crispy-forms
Changed req  25
Total commits  6  --  aisk/leancloud-python-test-app
Changed req  0
Total commits  385  --  rkp8000/hypothesize
Changed req  6
Total commits  127  --  shradhag/OPENLearn
Changed req  3
Total commits  345  --  tdfischer/spacewiki
Changed req  22
Total commits  379  --  flacjacket/pywayland
Changed req  1
Total commits  84  --  arruda/estatistica-github
Changed req  0
Total commits  22212  --  LuminateWireless/grpc
Changed req  23
Total commits  14  --  ahmetkucuk/personilized-education
Changed req  0
Total commits  8  --  wware/dumb-epydoc-project
Changed req  1
Total commits  37  --  victorevector/Genetics-and-Drugs
Changed req  1
Total commits  8  --  coxley/interface_monitor
Changed req  0
Total commits  49  --  digidotcom/python-wvalib
Changed req  4
Total commits  31  --  joakimalgroy/ip-location-map
Changed req  0
Total commits  2  --  0xANDREW/niagara-wishlist-exporter
Changed req  0
Total commits  10  --  lovedaybrooke/diceware
Changed req  0
```

Fig. 8. Changes in Requirements.txt file versus Total Commits

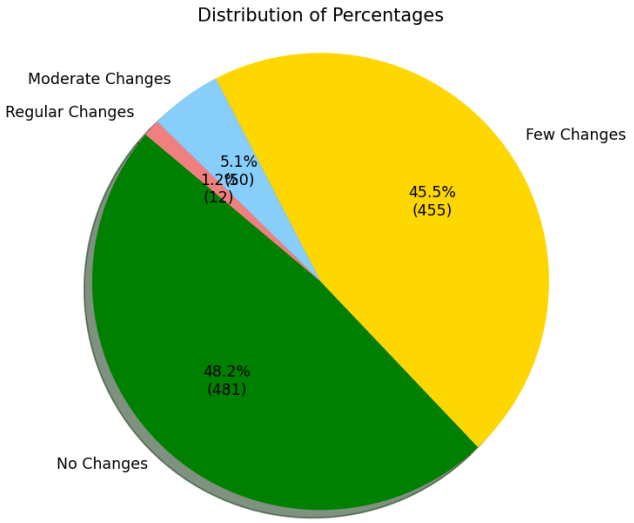

Fig. 9. Changed in requirements.txt file percentage distribution for projects

to improving usability.

For example, ”maraujop/django-crispy-forms” shows a devoted approach with regular revisions to the requirements.txt file, totaling 25 changes over 1766 contributions. Similarly, ”tdfischer/spacewiki” demonstrates a proactive technique with 22 modifications across 345 contributions. These projects encourage compatibility and deliver a user-friendly experience [13].

In contrast, ”arruda/estatistica-github” with 84 commits and absolutely no changes to requirements.txt, and ”zemlanin/Libr” with 7 commits and no modifications take a different approach. Similarly, ”LuminateWireless/grpc” demonstrates a reactive method with 23 modifications throughout 22212 commits. While their initial arrangement may have been handy, irregular upgrades might make it difficult to replicate the original setting as projects expand. 8 illustrates a list of projects with changes in the requirements.txt file compared to total commits.

We classify instances based on modifications noticed after carefully examining the changes to the requirements.txt file and the project’s total commits. Using the computed percentages, we divided the occurrences into four groups: ”No Changes,” ”Few Changes,” ”Moderate Changes,” and ”Regular Changes.” The distribution showed that the dataset comprised 482 instances (48.2%) with no changes, 455 instances (45.5%) with few changes, 51 instances (5.1%) with moderate changes, and 12 instances (1.2%) with frequent changes. A pie chart was created, as shown in 9, to highlight the proportional distribution of each group and provide a visual representation of these findings. This research sheds light on the frequency of various degrees of requirements.txt file changes made in the projects used as the dataset. Our findings suggest that more projects tend towards less frequent updates to their requirements.txt files, possibly due to a lack of awareness or perceived necessity [12].

RQ2: Are there any third-party libraries that can effectively generate requirements.txt files that precisely capture the needed dependencies of Python GitHub projects?

As an outcome of the knowledge obtained through Research Question 1, during which we considered the frequency that requirements.txt files are updated, our research now focuses on the problem of appropriately gathering project dependencies. We focused on finding third-party libraries or commands that are capable of assisting with the generation of requirements.txt files, consequently offering an efficient approach for developers to maintain accurate dependency information.

We checked a comprehensive assessment of some of the most commonly utilized libraries and commands currently available in the Python environment with the objective of gaining insight into the landscape of third-party solutions. We concentrate on establishing means that could precisely and automatically generate requirements.txt files and determine project dependencies. We have taken into consideration both libraries that have been acknowledged for their effectiveness in this particular scenario and those that receive considerable attention in the developer community.

Our research revealed a variety of third-party libraries and tools with a variety of ways of effectiveness when generating requirements.txt files for applications developed with Python.

In addition to the possibilities that we considered, the following distinguished out as noteworthy candidates:

- **pipreqs:** A third-party library that automates the creation of requirements.txt files by analyzing import statements in the project's source code. It offers simplicity and ease of use, which makes it an appealing alternative for projects with a clear import structure [16].
- **pip-tools (pip-compile):** This library provides commands like pip-compile to generate a locked requirements.txt file with pinned versions for dependencies. It emphasizes reproducibility, making it suitable for projects that require strict version control [17].
- **pipdeptree:** While its primary focus is on visualizing and managing dependency trees, it also offers the capability to generate requirements.txt files. It's particularly useful for understanding complex dependency relationships [18].
- **pipfreeze:** A built-in command in the pip package that generates a requirements.txt - like output listing all installed packages and their versions. It provides a quick snapshot of the environment but may include unnecessary dependencies [19].

After studying these alternatives, we observed that both pipreqs and pipfreeze had been frequently employed by developers for generating requirements.txt files, and the following are the primary explanations that led us to choose these two to do future work:

pipreqs:

- Simplicity: Its process involves analyzing import statements, resulting in the automated and streamlined creation of the requirements.txt file.
- Selectivity: It generates a list of only those packages that are required by the project, reducing unnecessary dependencies.
- Ease of Use: Its user-friendly command-line interface makes it accessible to developers of varying expertise levels.
- Customization: It offers options to exclude specific directories and files from scanning, allowing fine-tuning as per project needs.

pipfreeze:

- Built-In: As a part of the standard pip package, it eliminates the need for additional library installation.
- Quick Output: With a single command, it provides a snapshot of all currently installed packages and their versions in a requirements.txt - compatible format.
- Compatibility: Its output effectively captures the exact package versions in the current environment, making it suitable for sharing project-specific dependencies.

Ultimately, our research found a number of third-party commands and libraries for creating requirements.txt files in Python applications. However, developers encouraged pipreqs and pip freeze partly because of their ease of application and capacity to easily preserve project-specific requirements.

RQ3: How do the accuracy and utility of pipreqs and pipfreeze differ in the context of generating requirements.txt files for Python projects?

After carefully analyzing the packages pipfreeze and pipreqs for generating requirements.txt files, we've emphasized differences in accuracy and usability.

Differences in Accuracy and Usability:

Accuracy: pipfreeze retrieves version information from installed packages, ensuring a real-time representation of the environment. In contrast, pipreqs derives version details through static analysis of import statements, potentially leading to discrepancies if version constraints are not specified clearly [20] [23].

Usability: pipfreeze is readily available as part of the standard pip package, requiring no additional installations. However, its comprehensiveness can make it less user-friendly when pinpointing project-specific dependencies. pipreqs, on the other hand, is an external library that provides focused dependency generation but relies heavily on the organization of import statements [21] [25].

We comprehended many things information regarding the capabilities of pipreqs and pipfreeze when generating requirements.txt files after performing a comprehensive examination of 1000 projects. In particular, pipreqs experienced difficulties with generating requirements.txt file for 447 projects, but pipfreeze without difficulty succeeded across the entire spectrum. Additional investigation of the package counts uncovered many different components of each tool's result.

Our investigation demonstrated that when determining the projects, pipreqs generated 1366 packages, which corresponded with 3200 already existing packages in the environment. Pip freeze, on the other hand, demonstrated its capacity by producing a phenomenal 102380 packages, much outnumbering pipreqs and aligning with 7899 existent packages. These package count comparisons provided a meaningful glimpse into the two techniques' contrasting outcomes and covering.

Consider the "1heart_droplet-server" project as an example of a real-world application. The requirements.txt file the developer gave initially had 142 requirements, as shown in 10. However, pipreqs' focused approach reduced this list to a singly externally imported dependency in the code, notably the "Flask==2.3.2" requirement (11). Pipfreeze, on the other hand, demonstrated its thoroughness by generating 146 requirements (as seen in 12). Diagnostic and Statistical Manual code examination suggested that only the contents of the Flask package were externally imported into the project's functioning. These practical examples complemented the larger findings, illustrating the differences in the techniques' approaches and outputs and an attempt of pipreqs to generate a requirements.txt file but failed due to its limitation to generate direct dependencies only, which missed indirect dependencies that were necessary for the project.

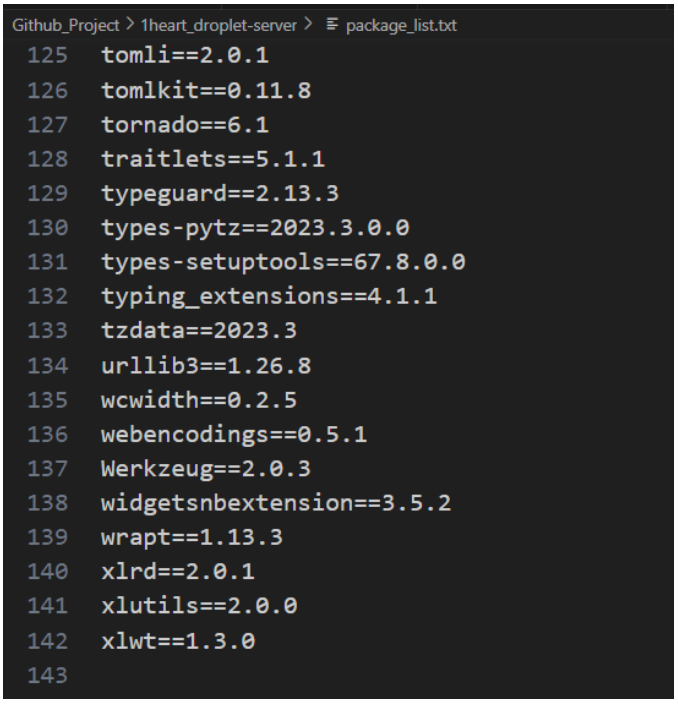

Fig. 10. Requirements.txt file from developer

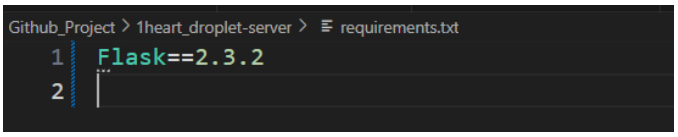

Fig. 11. Requirements.txt file generated by pipreqs

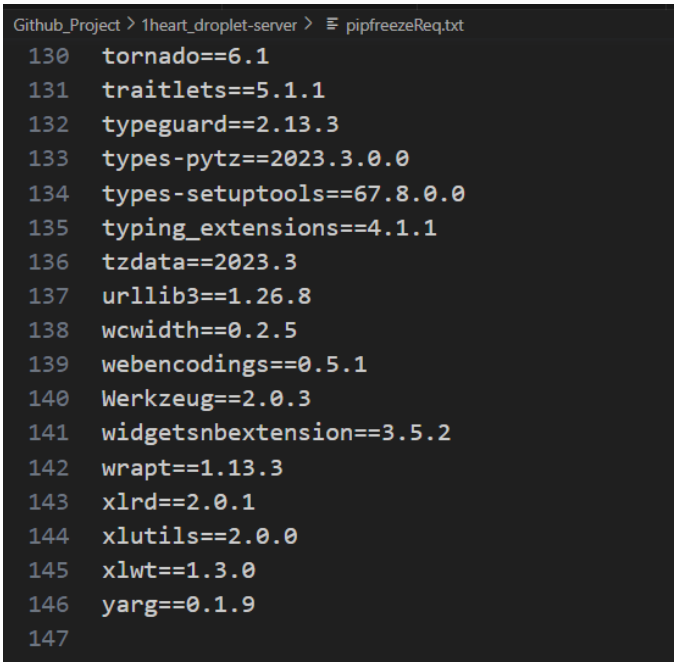

Fig. 12. Requirements.txt file generated by pipfreeze

A. *RQ4: Why do pipreqs fail to generate requirements.txt files for some Python projects, and what are the most prevalent reasons or restrictions for these errors?*

Although, pipreqs is an advantageous library, it is not without drawbacks when it has to do with generating requirements.txt files that correspond to particular Python applications. An in-depth investigation of the error messages and characteristics showed a number of prevalent explanations and restrictions associated with these failures. After evaluating the data, we can conclude that the errors arise for many different reasons, as listed below. The below table illustrates several Error Messages that can be encountered while utilizing pipreqs for generating the requirements.txt file for a project.

Error Name	Count
SyntaxError: Missing parentheses in call to 'print'. Did you mean print(...)?	286
SyntaxError: multiple exception types must be parenthesized	29
SyntaxError: invalid syntax	28
UnicodeDecodeError: 'charmap' codec can't decode byte 0x8*in position x: character maps to undefined	40
UnicodeDecodeError: 'charmap' codec can't decode byte 0x9* in position x: character maps to undefined	32
TabError: inconsistent use of tabs and spaces in indentation	8
IndentationError: unexpected indent	1
except TypeError, exc:	1
except pyinotify.NotifierError, err:	1
except IOError, yaml.error.YAMLError:	1
except ClientHttpError, e:	1
SyntaxError: expected ':'	1
except OSError, e:	2
print "KeyError", ke	1
except urllib2.HTTPError, e:	1
print "HTTP Error", response.status_code	1
print "Validation Error"	1
except TemplateSyntaxError, e:	1
except request.URLError, ex:	1
SyntaxError: invalid hexadecimal literal	1

Error Patterns and Causes:

1) **SyntaxError:**
One of the most recurring errors encountered is the "SyntaxError: Missing parentheses in call to 'print.'" This error, observed 286 times, arises from a compatibility mismatch between the version of Python used in the project and the code within the pipreqs package [31]. Specifically, the error message "Did you mean print(...)?," indicates that the package's code, designed for Python 2.x, encounters issues when used with Python 3.x as stated in a discussion in the pipreqs' GitHub Issues [35]. The discrepancy between print statements,

where Python 2.x used print as a statement without parentheses while Python 3.x uses it as a function, triggers this syntax error [31] [34].

2) **UnicodeDecodeError:**
Instances of UnicodeDecodeError point to challenges in decoding bytes into Unicode characters. These errors arise while processing files with diverse character encodings, underscoring potential issues with the compatibility of text encodings in the project's codebase. Some recommendations have been made to fix this issue, such as enforcing utf-8 as the default encoding used in gbk. However, efforts are still ongoing, and the problem hasn't been solved [37].
3) **IndentationError and TabError:**
The occurrence of IndentationError and multiple instances of TabError points to indentation inconsistencies within the Python code. These errors typically surface when there's a misalignment between tabs and spaces in the codebase, underscoring the significance of consistent code formatting practices. These are syntactic issues in Python 3, not a flaw or feature of pipreqs. Pipreqs should operate without issue if the faults are corrected [36].
4) **TypeError and Other Errors:**
Multiple occurrences of TypeError and various other error types, such as KeyError, ValueError, NameError, IndexError, AttributeError, and ImportError, point to diverse issues in the codebase, ranging from mismatched data types to incorrect usage of functions and operators. Some of the methods for resolving errors have been found to work for some people but not for others. [38].

Limitations and Dependencies:

Pipreqs, as a static code analysis tool, relies heavily on the structure and organization of the project's code. Some of the notable limitations and dependencies include:

- **Python Version Compatibility:** Pipreqs exhibits sensitivity to the version of Python employed in the project. While pipreqs code aligns with Python 2.x, employing it in a Python 3.x environment results in SyntaxErrors due to differing print statement syntax [31].
- **Encoding Issues:** UnicodeDecodeError instances highlight potential discrepancies in character encoding practices across files within the project.
- Code Formatting: IndentationError and TabError underscore the importance of consistent code formatting, particularly when mixing tabs and spaces for indentation.
- Static Analysis: Pipreqs' reliance on static code analysis implies that its efficacy hinges on the clarity of import statements and overall code structure within the project [34].

In essence, the difficulties experienced by pipreqs highlight the necessity of maintaining code compatibility, adhering to consistent formatting, and understanding its dependencies on Python versions and coding guidelines within the projects under consideration.

VI. Discussion

Our study's insights into generating accurate requirements.txt files through tools like pipreqs and pipfreeze are indeed valuable. However, it's essential to consider certain limitations that deserve careful attention. While our project selection aimed at diversity, it's worth acknowledging that inherent variations across projects could influence our findings. We addressed this potential bias by deliberately including projects from various domains and sizes.

External generalizability can be impacted by the diversity of code structures and project requirements. To enhance the robustness of our findings, we intentionally selected projects spanning a wide range of domains and sizes, thereby bolstering external validity. While construct validity was upheld through the utilization of multiple analysis methods, it's important to recognize that tool limitations and nuances in codebases might introduce some level of bias. To mitigate this concern, we conducted meticulous manual validation and employed a range of diverse analysis techniques.

After carefully analyzing the packages pip freeze and pipreqs for generating requirements.txt files, we've identified different advantages and disadvantages associated with each application, emphasizing differences in accuracy and usability.

Advantages and Disadvantages:

pipfreeze:

Advantages:

- Comprehensive List: pipfreeze generates an exhaustive list of all installed packages in the Python environment, regardless of their hierarchy within the dependency tree [20].
- Complete Snapshot: This approach provides a snapshot of the exact environment, ensuring that no dependencies are overlooked [21].
- Real-Time Version Information: Version details are retrieved from the currently installed packages, providing real-time accuracy [22].

Disadvantages:

- Inclusion of Unnecessary Packages: The comprehensive nature of the output can lead to the inclusion of redundant or unused packages, potentially bloating the requirements.txt file [20].
- Difficulty in Discerning Project Dependencies: As it captures all installed packages, distinguishing between direct project dependencies and packages used internally by the environment can be challenging [21].

pipreqs:

Advantages:

- Targeted Dependency List: pipreqs focuses on identifying direct project dependencies by analyzing import statements in the project's source code [23].
- Streamlined and Simplified: The generated requirements.txt file is concise and includes only packages explicitly required by the project, reducing potential clutter [24].

- Potential for Smaller Footprint: The narrower scope of dependencies can lead to a more manageable and focused development environment [25].

Disadvantages:

- Limited to Direct Dependencies: The tool may miss indirect dependencies, leading to an incomplete requirements.txt file if import statements are not comprehensive [23].
- Potential Omission of Dependencies: pipreqs might not include specific required packages if their import statements are not structured in a recognized manner [24].
- Dependency on Source Code Structure: The accuracy of the generated list heavily relies on the organization and clarity of import statements in the source code [25].

Looking ahead, several promising avenues for future research beckon. There is a potential to refine tool accuracy, explore automated approaches to conflict resolution in dependencies, and integrate to enhance requirement generation. Furthermore, investigating the frequency and underlying dependency updates and compatibility analysis across different package managers could offer valuable insights.

In summation, our study serves as a foundational stepping stone for refining tools and methodologies in the realm of dependency management. At the same time, we acknowledge the limitations; the path forward lies in addressing these concerns through further research, ultimately paving the way for enhanced accuracy, usability, and comprehensiveness in the domain.

VII. Conclusion

In conclusion, our study delved into the accuracy and usability of requirements.txt files generated by pipreqs and pip freeze for Python projects. Our findings shed light on the challenges and opportunities within dependency management. Analyzing the frequency of requirements.txt updates, we revealed the crucial role of regular updates in enhancing project usability. Projects that maintained up-to-date requirements.txt files experienced smoother onboarding for contributors and reduced compatibility issues.

Our exploration of third-party tools demonstrated that pipreqs and pip freeze offer viable options for generating accurate requirements.txt files. While both tools play vital roles, pipreqs stand out for their precision in capturing project-specific dependencies.

By evaluating the advantages and disadvantages of these tools, we highlighted the distinctive characteristics that set them apart. Pipfreeze provides a comprehensive package snapshot, whereas pipreqs excels in generating a compact yet accurate dependency list. Nonetheless, our analysis of pipreqs errors also exposed compatibility and code construct limitations.

Though our study offers valuable insights, limitations exist due to varying code structures and project requirements. Future research should focus on refining tool accuracy and automating conflict resolution to advance the field. Investigating dependency update patterns, cross-package manager compatibility, and machine-learning applications can further enhance our understanding of dependency management.

In essence, our study contributes to the ongoing evolution of Python dependency management practices. We aspire to foster accurate, reproducible, and user-friendly project environments by addressing the limitations and paving the way for future investigations.

Acknowledgment

We would like to express our sincere gratitude to Prof. Dr. Muhammad Asaduzzama of Lakehead University's MS in the Computer Science department for supporting and directing us through his talks and for giving important inputs and feedback that helped us finish this project, as well as the time and effort he put into assisting us in completing the paper successfully.

References

[1] Munaiah, N., Kroh, S., Cabrey, C., & Nagappan, M. (2017, April 18). Curating github for engineered software projects - empirical software engineering. SpringerLink. https://link.springer.com/article/10.1007/s10664-017-9512-6

[2] Jones, E., Oliphant, T., & Peterson, P. (2014). SciPy: Open Source Scientific Tools for Python. Computing in Science & Engineering, 16(2), 13-21.

[3] Brown, N., & Wilson, G. (2018). Ten simple rules for making research software more robust. PLoS Computational Biology, 14(10), e1005996.

[4] Pimentel, J. S., Sousa, J. P., & Pereira, J. A. (2019). Empirical study on the accuracy and performance of Python package managers. In Proceedings of the 20th ACM/IEEE International Conference on Mobile Software Engineering and Systems (pp. 94-104)

[5] Nat. (2023, January 3). Better Python dependency while Packaging Your Project. Medium. https://medium.com/python-pandemonium/better-python-dependency-and-package-management-b5d8ea29dff1

[6] Van Rossum, G. (2020). PEP 508: Dependency specification for Python Software Packages. Python Enhancement Proposals, 508.

[7] Ito, Luana Gribel, et al. "What are the Top Used Modules in Python Open-Source Projects?." Anais do Computer on the Beach 13 (2022): 037-044.

[8] Ye, Hongjie, et al. "Knowledge-based environment dependency inference for Python programs." Proceedings of the 44th International Conference on Software Engineering. 2022.

[9] Abbas, Mashal, Shahpar Khan, and Abdul Monum. "Fix Your Requirements (. txt)! A Study of Vulnerable Python Packages in Open-Source Software."

[10] Pimentel, João Felipe, et al. "A large-scale study about quality and reproducibility of jupyter notebooks." 2019 IEEE/ACM 16th international conference on mining software repositories (MSR). IEEE, 2019.

[11] Apostolidis, George David. "Evaluation of Python code quality using multiple source code analyzers." (2023).

[12] Van Rossum, G. (2020). PEP 508: Dependency specification for Python Software Packages. Python Enhancement Proposals, 508.

[13] Gousios, G., & Spinellis, D. (2012). A git-based infrastructure for capturing software evolution. Science of Computer Programming, 77(7-8), 779-797.

[14] B. M. Randles, I. V. Pasquetto, M. S. Golshan and C. L. Borgman, "Using the Jupyter Notebook as a Tool for Open Science: An Empirical Study," 2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL), Toronto, ON, Canada, 2017, pp. 1-2, doi: 10.1109/JCDL.2017.7991618.

[15] M. Alfadel, D. E. Costa and E. Shihab, "Empirical Analysis of Security Vulnerabilities in Python Packages," 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 2021, pp. 446-457, doi: 10.1109/SANER50967.2021.00048.

[16] Doug, H. (2016). pipreqs. GitHub Repository. https://github.com/bndr/pipreqs

[17] Vinta, J. (2020). pip-tools. GitHub Repository. https://github.com/jazzband/pip-tools

[18] Jazzband. (2021). pipdeptree. GitHub Repository. https://github.com/jazzband/pipdeptree

[19] Python Packaging Authority. (2021). pip documentation. https://pip.pypa.io/en/stable/
[20] Python Packaging Authority. (2021). pip documentation. https://pip.pypa.io/en/stable/
[21] Full Stack Python. (n.d.). Application Dependencies. https://www.fullstackpython.com/application-dependencies.html
[22] Stack Overflow. (2017). How to find all packages and its version used in my project. https://stackoverflow.com/questions/43718867/how-to-find-all-packages-and-its-version-used-in-my-project
[23] Pipreqs Documentation. (n.d.). GitHub Repository. https://github.com/bndr/pipreqs
[24] Doug, H. (2016). pipreqs. GitHub Repository. https://github.com/bndr/pipreqs
[25] Vinta, J. (2020). pip-tools. GitHub Repository. https://github.com/jazzband/pip-tools
[26] Yadav, A. (2023, February 27). Python requirements.txt: What it is and how to use it?. AskPython. https://www.askpython.com/python/python-requirements-txt
[27] Introduction to python:getting started. Introduction to Python: Getting Started. (n.d.). https://rowannicholls.github.io/python/intro/getting_started.html
[28] Application dependencies. Full Stack Python. (n.d.). https://www.fullstackpython.com/application-dependencies.html
[29] Pipreqs. PyPI. (n.d.). https://pypi.org/project/pipreqs/
[30] Python Pipreqs - how to create requirements.txt file like a sane person. Better Data Science. (n.d.). https://betterdatascience.com/python-pipreqs/: :text=Pipreqs%20works%20by%20scanning%20all,txt%20.
[31] Pipreqs Issue Tracker. (2021). SyntaxError with print statement 145. https://github.com/bndr/pipreqs/issues/145
[32] Pipreqs Pull Request. (2021). Add support for Python 3 204. https://github.com/bndr/pipreqs/pull/204
[33] Pipreqs Issue Tracker. (2015). Python 3 print without parenthesis. https://github.com/bndr/pipreqs/issues/60
[34] Stack Overflow. (2015). Python 3 print without parenthesis. https://stackoverflow.com/questions/32122868/python-3-print-without-parenthesis
[35] Bndr. (n.d.). Python 3.6 support? · issue 68 · BNDR/pipreqs. GitHub. https://github.com/bndr/pipreqs/issues/68
[36] Bndr. (n.d.-b). Taberror and syntaxerror when running pipreqs in downloaded repos · issue 60 · BNDR/pipreqs. GitHub. https://github.com/bndr/pipreqs/issues/60
[37] Bndr. (n.d.-c). UNICODEDECODEERROR: “GBK” codec can’t decode byte 0x8c in position 358: Illegal Multibyte Sequence · issue 241 · BNDR/pipreqs. GitHub. https://github.com/bndr/pipreqs/issues/241
[38] Bndr. (n.d.-a). Error executing newly installed pipreqs · issue 4 · BNDR/pipreqs. GitHub. https://github.com/bndr/pipreqs/issues/4