What are the Top Used Modules in Python Open-Source Projects?

Luana Gribel Ito
National Institute for Telecommunications - Inatel
Santa Rita do Sapucaí, Brazil
luanagribel@gec.inatel.br

Mariana Helena Inês Moreira
National Institute for Telecommunications - Inatel
Santa Rita do Sapucaí, Brazil
marianahelena@gec.inatel.br

Sarah Brandão Souza
National Institute for Telecommunications - Inatel
Santa Rita do Sapucaí, Brazil
sarahbrandao@gec.inatel.br

Sinara Pimenta Medeiros
National Institute for Telecommunications - Inatel
Santa Rita do Sapucaí, Brazil
sinaramedeiros@gec.inatel.br

Phyllipe Lima
Federal University of Itajubá - UNIFE
Itajubá, Brazil
phyllipe@unifei.edu.br

ABSTRACT

When a team of developers are creating new software, they most likely will use libraries of code that can assist in a given required feature. One source to find these libraries can be popular question-answer websites, blogs, personal web pages and the usage of tools that can automatically suggest libraries. Popularity might be one criterion that developers can use when choosing a library. In this work, we performed an empirical evaluation through mining Python projects hosted in GitHub to identify the most popular used modules. We selected 129 projects based on specific criteria, one of them being the number of stars that reflects their popularity. To automate the data extraction process, we developed the *PySniffer*, an open-source tool that performs a static code analysis in Python scripts, checking which modules from both the standard library and external modules are used in a project. Our tool also has a front-end that can display the data more friendly with statistical information. As a result, we generated a list with the top used modules in Python projects hosted in GitHub, serving as complementary information alongside the most popular libraries informed in personal blogs and websites.

KEYWORDS

GitHub, Mining Software Repositories, Python, Static Code Analysis

1 INTRODUCTION

Python is a high-level, general-purpose, multi-paradigm programming language released in 1991. It is one of the fastest-growing languages, quickly catching up with other established languages in the market such as C, C++ and Java.

In part, this is because Python is a simple language with a fast learning curve and an emphasis on readability. In addition, Python is the preferred language used in fields such as data science and machine learning, and therefore, should keep this language among the most popular ones for the following years [1].

Python is an open-source language that attracted a large community of adept developers. In this way, the community itself can contribute to the evolution of the language by creating scripts that perform different tasks. These are called modules and together they form a package [2–4].

Searching for libraries to reuse is a common action performed by developers [5], and popularity can be one criterion for choosing a library and how to use it [6, 7]. As for the popularity, there is a lot of information on the internet about modules created by third parties and those part of Python's standard library. It is common for blogs and other websites to write about which modules are the most used, the most popular, the fastest, and so forth. However, these might lack work investigating these modules' presence more rigorously, such as mining software repositories to validate their presence.

We conducted a study to investigate the most used modules in Python open-source software system hosted in GitHub to overcome this. To reach this goal, we developed a tool named *PySniffer* that can analyze open-source repositories hosted on GitHub and extract the usage of the module. Afterwards, we can validate and compare the obtained data with the grey literature. The base chosen was GitHub because it is one of the most popular hosting platforms for source code and files. There were approximately 2,427,136 repositories with Python scripts on GitHub until writing this paper.

PySniffer can perform static analysis of Python code and verify all the imports, generating a list of external modules and another list of modules from the standard library present in the repositories with Python scripts best ranked on GitHub, using the stars as criteria [8]. Therefore, we do not only rely on data provided by package managers but instead perform a code inspection to validate the presence of the module. In addition, the tool also has a front-end that generates graphics to ease the understanding of the results and allows the user to compare his project to check if the libraries used by him are present in this generated list. Therefore, the list generated by *PySniffer* can also be seen as a reference list.

From our results, we found 1902 modules, 89% of them external. It was observed that 'os', 'sys' and 're' modules from Python standard library were the most used packages. But analyzing only the results of packages implemented by third parties, 'setuptools', 'requests' and 'pytest' were the most used in the selected projects.

The remainder of this work is organized as follows. Section 2 explains what Python Modules are and how they are imported into Python files. Section 3 discusses the concept of Abstract Syntax Tree (AST) and how the AST module operates since it is the core

of the PySniffer tool. Section 4 brings the research method adopted. Section 5 presents the PySniffer tool in details. Section 6 approaches the exploratory data analysis and discussion. Section 7 brings a selection of related works. Finally, Section 8 contains the conclusion of this study and suggestions for future work.

2 PYTHON MODULES

Modules are the base of Python language structure, but their concept can change depending on the author. Some of them use the term "Module" just for the programs that are imported to use in another file, like the standard and the third-party libraries. But according to the official documentation[1], "Module" is a file that has functions and classes definitions and Python executable instructions, whose name is the Module name plus suffix `.py`. In other words, every script with `.py` extension can be a Module. The documentation also exposes that Modules can import another Modules and usually the import declaration is made at the beginning of the program [9].

The idea of this modular design is sharing the complex tasks in small parts in order to make easier the system organization, to help in the reuse of codes and maintenance, allow the implementation of shared services and data, reduce the collision of names and optimize developing and debugging [2, 3, 9].The Python standard library brings together a considerable number of modules that offer a wide variety of resources that facilitate the code develop. Besides it, there are several other packages available for installation on Python Package Index, which can be installed using the following command:

```
python -m pip install SomePackage
```

All modules expend some memory, this way, is it recommended to install just what is needed [9]. Usually, developers use a tool that will be responsible for the management of these modules. In this context, modules are also known as dependencies. Bellow we describe three Python package/dependencies managers commonly used by developers:

- Pip[2]: A package manager that allows the installation of Python packages and uses the `Requirements` file to manage dependencies. This files uses a structure based on "key" "value" pairs, where the "key" is the dependency, and the "value" is the version. Following we list examples of this file:

  ###### Requirements without Version Specifiers
  ```
  nose
  beautifulsoup4
  ```
  ###### Requirements with Version Specifiers
  ```
  docopt == 0.6.1
  keyring >= 4.1.1
  ```
  ###### Refer to other requirements files
  ```
  -r other-requirements.txt
  ```

- Pipenv[3]: it's a production tool that includes packaging resources, allowing the creation of a virtual environment for the project and the management of dependencies through the `Pipfile` file, which uses the structure shown below:

  ```
  [packages]
  requests = "*"
  [dev-packages]
  pytest = "*"
  ```

- Poetry[4]: it's a tool to manage Python dependencies. This task is done using a file named pyproject.toml, this file follows this structure:

  ```
  [tool.poetry.dependencies]
  python = "*"
  [tool.poetry.dev-dependencies]
  pytest = "3.4"
  ```

To use a module component regarding of being a standard resource, own resource or a resource that has been installed, it must be imported. There are three ways to accomplish this [3]:

(1) Importing the module directly:
`import moduleName`

To access a module's attribute, use the structure: `moduleName.attributeName`.

(2) Import a specific attribute from a module:
`from moduleName import attributeName`

(3) Import all attributes from a module:
`from moduleName import *`

Using this strategy, all the attributes are imported, except those strating with the underscore character (_). This mode, however, is not recommended since name collisions can occur.

If we try to execute a Python file, that uses external modules, without previously installing them the code execution returns `ModuleNotFoundError` and displays the name of the first component that could not be found. However, if any resource is installed but not used, the system does not point out this information. Files created by dependencies managers, such as `Requirements`, do not allow to infer which modules are actually being used, once they have the information of installed packages. To get this information, we can perform a static code analysis on the project to verify all the imported modules. Instead of performing a pure textual analysis, an Abstract Syntax Trees can be used.

However, as explained earlier, all Python files are modules. This way, the isolate analysis of import declaration does not allow us to determine if the imported resource is a custom developer script, a module from the standard library or from a third-party library. To know which is the component origin, we need further exploration.

[1] https://docs.python.org/3/
[2] https://pip.pypa.io/en/stable/
[3] https://pipenv.pypa.io/en/latest/
[4] https://python-poetry.org/

3 ABSTRACT SYNTAX TREE

An Abstract Syntax Tree (AST) is a syntactical structure represented by a tree. It is used to interpret the source code instructions of a given programming language. Each node in this tree refers to constructions present in the source. For instance, an import instruction will be a node in the tree, just as methods, members, and so forth. The tree also presents the relationship between the nodes through the edges. For instance, a conditional block (if/else statement) has a node that connects to other nodes representing the statements inside the conditional block.

ASTs can improve in the process of syntactically comprehending a source code. Furthermore, it can be used to build software engineering tools that perform static code analysis, such as [10, 11], to track the execution flow of a program and modify the structure of a source code prior to its execution. Some previous study shows that building an AST can aid in software evolution [12] and teach data structure and algorithms [13].

```
1 tree = ast.parse(contents)
2 for node in ast.walk(tree):
3    if isinstance(node, ast.Import):
4       for subnode in node.names:
5          raw_imports.add(subnode.name)
6    elif isinstance(node, ast.ImportFrom):
7       raw_imports.add(node.module)
```

Figure 1: Python code from pipreqs package

Since this work is focused on the Python programming language, we used the `ast` module [5] from the standard library. This module has methods that help build an AST and traverse the nodes searching for information such as "imports from a .py file", as shown in Figure 1. To also obtain the used libraries in a Python project, it was used the `pipreqs` package [6], which was one of the bases for the tool PySniffer, further explained in Section 5

4 RESEARCH DESIGN

This section describes the research design to reach our main goal, i.e., investigate the top used modules in Python open-source projects. We begin presenting the steps we carried out, followed by the criteria we used to select the projects. We build our research design following the work of [14].

4.1 Research Method

- Step # 1 - Project Selection:
  The first step is to determine what projects hosted on GitHub we will use as our sample. Since we are interested in finding the most popular modules, we will also look for popular Python projects. Among other criteria that we will present in the following subsection, we used the GitHub stars [8] to rank these projects.
- Step # 2: Develop a tool to extract the data:
  We need to develop a tool that scan Python projects and extract, for every `.py` file, what are the modules that were imported. For this, we developed the *PySniffer*. This tool also has a front-end that eases displaying the most popular modules found and allows potential users to compare them with those found in their projects.
- Step # 3: Data Collection
  With the projects selected and the *PySniffer* ready, we execute the tool. It will clone all projects, scan every `.py` file, extract the modules imported, and generate a report in JSON format.
- Step # 4: Exploratory Data Analysis and Discussion
  With our data collected, we will perform an analysis of the most used modules and discuss their domain and features. Therefore, we carry both a quantitative and qualitative analysis in our exploratory analysis.

4.2 Projects Selection

To generate our data, we selected open-source projects hosted on GitHub. We were able to obtain a sample of 129 projects. This manual selection of projects was defined based on the following list of criteria:

- Uses Python 3 language;
- More than 10 thousand stars;
- READ.ME mostly written in English;
- It is not just a collection of Python language examples or teaching scripts, which includes course materials and books;
- Not archived;
- Not just a repository of links or lists or text documents;
- Not a coding font or font family or similar repository;
- Not the repository of the language itself (CPython);
- No syntax or indentation errors in .py files.

In addition, there is a restriction regarding the search for modules. If the developer uses the module name in some file or folder, the module is disregarded by the `pipreqs` tool because the name of the modules themselves are the names of the `.py` scripts. Thus, it is not a good Python programming practice to use the name of the external module or the default library as the name of files and folders. In this scenario, the result presented by *PySniffer* will have inconsistencies.

In short, from the 129 selected projects, we obtained repositories from several different domains such as desktop/web/mobile applications, tools, frameworks, libraries, packages, command-line interfaces, emulators, plugins, terminals, engines, services, platforms, research, academic, personal or commercial projects were selected.

5 PYSNIFFER

In this section we present the construction of the *PySniffer*[7] tool that we developed to scan Python software repositories hosted in GitHub and extract the modules usage data. The tool also clones the list of the selected repository. We begin presenting the tool itself, that works as a back-end, that is able to generate a report file with the modules found. Afterwards we present the front-end web application developed to display these modules, bringing these results closer to a developer interested in monitoring modules used in his own system.

[5] https://docs.python.org/3/library/ast.html
[6] https://pypi.org/project/pipreqs/
[7] https://github.com/SinaraPimenta/PySniffer

5.1 PySniffer Flow of Execution

We will describe the tool presenting both the structure and the flow of execution. Figure 2 present the overview of how *PySniffer*, going from cloning GitHub repositories to generating the results.

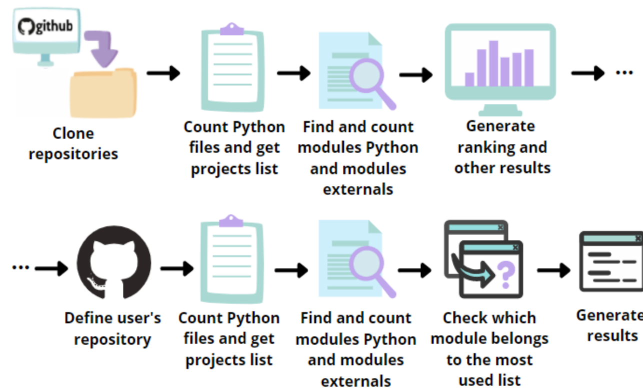

Figure 2: PySniffer General Flow

PySniffer is executed via command line and has has three operation mode, as shown in Figure 3. The `download_repos` mode is responsible for downloading repositories from GitHub. The mode `analyzing_repos` obtains which modules are the most used from the downloaded projects. And the `analyzing_my_project` extract the modules used in a specific GitHub repository and compares with the list generated from the previous mode. The goal of this mode is that a user can compare his own project with the reference list.

```
Usage: main.py [OPTIONS] COMMAND [ARGS]...

  ###########      PySniffer      ###########

Options:
  --version  Show the version and exit.
  --help     Show this message and exit.

Commands:
  analyzing_my_project  Generate statistics for my project
  analyzing_repos       Generate projects statistics
  download_repos        Download GitHub projects
```

Figure 3: PySniffer Operation Mode

The first step is to clone the repositories, using the following command:

```
python cmd/cli/main.py download_repos
```

A script will be executed to clone the projects and save them in a specific folder for future analysis. At the time of this writing, the list of projects is fixed and chosen according to criteria detailed in Subsection 4.2.

Afterwards we should execute the command:

```
python cmd/cli/main.py analyzing_repos
```

PySniffer will begin scanning the projects downloaded to obtain information about module usage. It counts the total number of Python Scripts in each project and executes the `pipreqs` library in all projects. This library is responsible for the automatic generation of the `Requirements` file for each project. This makes it possible to determine which libraries are used in the `.py` scripts in each one of these projects. Besides, the `pipreqs` has been manipulated to also obtain the most used libraries from Python itself, so the project's data analysis covers even a larger area of research. In addition, a parameter was changed to ignore errors of type `UnicodeDecodeError`.

So far, *PySniffer* has collected and prepared all the necessary data. The next step is to read the modules and count them, in order to extract useful and concrete information regarding their usage. For that, PySniffer analyzes the files that were generated previously and obtains the libraries, making a new list containing all this information.

After the previous calculations are performed, a report in JSON format is generated with the modules usage information ranked, with most used module on top. The tool also generates a graphic with the 10 most used modules from the standard library and another with the 10 most used external modules. Thus, at this point, *PySniffer* has performed the complete standard operation and generated a report, and graphics, with the top used modules from best ranked GitHub Python repositories. This report can now be used as a reference list.

As mentioned, *PySniffer* has a third operation mode that lets users compare these top used modules, with the ones found in his own project. Currently, *PySniffer* requires that this project is also hosted in GitHub. To run this command the user needs to pass, as a parameter, the URL where the project to compared is hosted. This way, the same functionalities performed in the second operation mode will be carried out. However, only this specific project will be used instead. Afterwards, the comparison process begins. This step consists in verifying if the modules found in the repository provided by the user belong to the generated list in the analysis of GitHub repositories, that is, the library is part of the group of the most ones used by Python developers. Therefore, to run this operation mode, *PySniffer* must have already generated the reference list, i.e., executed in the second operation mode.

This comparison is very interesting, because the developer can verify and validate if the used packages in his project might be the best for a given task. For instance, if a project uses the *Jasmine* test framework, but it does not appear in the most used modules list, this may indicate that the user could have used a better framework. This is assuming that the most popular modules are those that exercise their functions in a better way or are more complete. The final decision is up to the team of developers and only they know the reason for choosing a specific module to use. However, the reference list can be used as a good starting point. The generated result also allows another inferences depending on the focus of the user investigation.

Afterwards, two lists will be generated from this operation mode. One will contain the modules that also belong to the reference list, i.e., the one obtained in the `analysing-repos` mode, and a second list with the modules that were not found in that reference list.

5.2 PySniffer Front-End Web Application

To further ease the analysis process, bring the results closer to an end-user, and provide an appealing graphical user interface (GUI), *PySniffer* also has a front-end web application. This allows users to access the platform and obtain statistical information for the projects in a more friendly way when compared to numbers on tables.

The web application is responsible for reading data from the previously generated JSON files (described in Subsection 5.1) and displaying them in a more friendly and intuitive way for the user, with lists and graphics that aid in a better understanding and analysis of the results. It also allows easier comparison of the modules extracted in a given repository, with the reference lists of top used modules. In addition, the web application has a search bar and allows filtering the result by the module's origin (standard library, PyPi or any of these), as shown in Figure 4. The application is available in: https://pysnifferweb.herokuapp.com.

The web application also has details about the *PySniffer* usage, links for the source code and other relevant information of the overall project. It is also possible to download the list with all modules found and their frequency in the projects, or generate the list filtering by the module's origin. The download file is of CSV format.

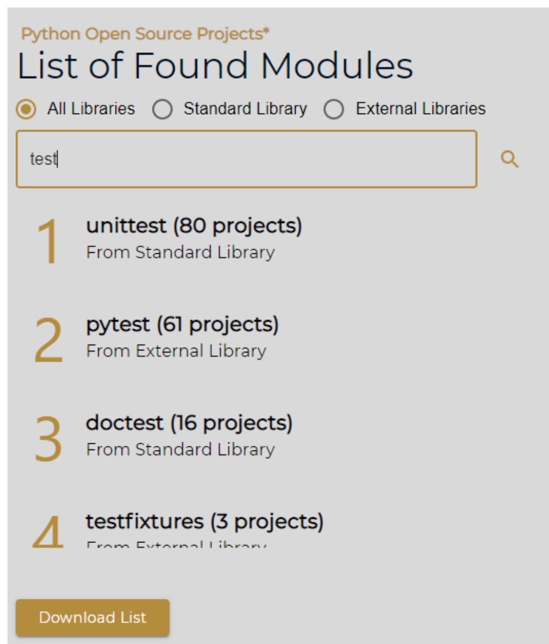

Figure 4: Web Application - Searching in List of Found Modules

Finally, another feature offered by the web application is located in the "Statistics" tab. It contains the results obtained in this study and is displayed in Figure 6. The data presented on this web page are further explained in the Section 6.

The web application source code is available in our GitHub repository[8].

[8]https://github.com/Mariana-Helena/PySniffer-Web-Application

```
######################################################################
              PySniffer - Generate My Project Statistics
######################################################################

1) COLLECTING DATA AND GENERATE RESULTS:
Lib                                  Count
Flask                                1
paho_mqtt                            1
pymongo                              1
Numero de libs = 3


Lib                                  Count
time                                 1
datetime                             1
os                                   1
random                               1
Numero de libs python = 4

Returns were generated in returns/my_project

2) ANALYZING MY PROJECT

External libraries also used by the github projects that were analyzed:
{'paho_mqtt', 'Flask', 'pymongo'}

Not exists external libraries not used by the github projects that were analyzed!

Standard libraries also used by the github projects that were analyzed:
{'time', 'os', 'datetime', 'random'}

Not exists standard libraries not used by the github projects that were analyzed!
```

Figure 5: Output of PySniffer for a custom Project Statistics

5.3 Example Usage for Custom Project

As an example usage of comparing the modules list found in a custom project with the reference list generated by *PySnnifer* we select the open-source project *Smart Windows* [9] for analysis. The main goal here is to check if the libraries used in the custom project are among those collected from popular projects from GitHub. Running *PySniffer* and passing as parameter the URL to the custom repository we have the output on Figure 5.

We observe that all modules used in the project *Smart Windows* were also used by popular Python projects. For instance, the Flask framework appears in the list of used libraries, confirming that the user has chosen a popular tool and, probably, will better meet its requirements in optimal way.

6 EXPLORATORY DATA ANALYSIS AND DISCUSSION

We divide our data analysis into two parts. First, we expose the top modules we found in GitHub repositories, separating between standard and third-party modules. Then, in our second analysis, we perform a discussion on the top modules, commenting on their responsibilities and relevant features.

The results presented below refer to the 129 repositories[10] chosen as the sample for this work.

6.1 Top Used Modules

Currently there are about 338215 external modules available in PyPI[11] and 1785 modules from the standard library[12].

From the 129 analyzed repositories, we obtained 57201 Python files. After running the *PySniffer*, 1902 modules were obtained, being 1701 external modules and 201 modules from the standard library. The number of external modules used represent approximately 89% of the used libraries, developed by third party groups to implement features that became necessary to aid developers. We present in

[9]https://github.com/SinaraPimenta/Projeto_C115_Smart_Windows
[10]Available in the repository: https://github.com/SinaraPimenta/PySniffer
[11]https://pypi.org/
[12]https://github.com/bndr/pipreqs/blob/master/pipreqs/stdlib

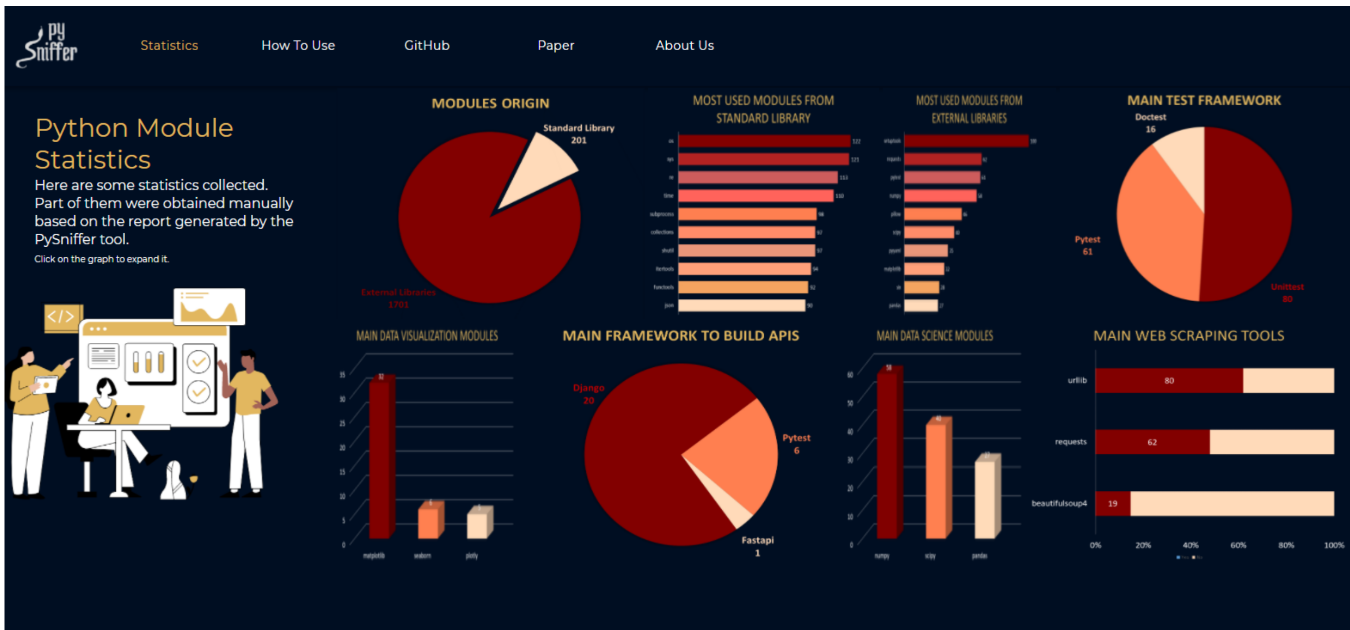

Figure 6: Web Application - Statistics Tab

Figure 7 the top 10 third party modules, and in Table 1 we present further details about them.

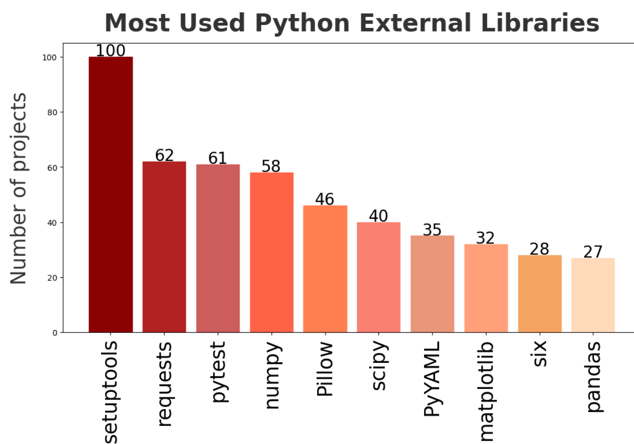

Figure 7: External Modules

For the modules that belong to the standard library, we present in Figure 8 the top 10 found. In Table 2 we further detail them.

6.2 Qualitative Analysis

Now that we have the list of the top used modules in Python open-source projects, we can discuss their responsibilities. In other words, we can discuss the top features required in these selected projects that we extracted our data. We also highlight some differences from the data found in blogs for complementary purposes. The list of used blogs is available in the *PySniffer* repository.

Table 1: External Modules

Module	Description
setuptools	Designed to make packaging Python projects easier.
requests	Requests allows you to send HTTP/1.1 requests extremely easily
pytest	Framework makes it easy to write small tests.
numpy	Is the fundamental package for scientific computing.
pillow	Provides file format support and image processing capabilities.
scipy	Is open-source software for mathematics, science and engineering.
pyyaml	Is a YAML parser and emitter for Python.
matplotlib	Create static, animated, and interactive visualizations in Python
six	Python 2 and 3 compatibility library
pandas	Open source data analysis and manipulation tool.

Table 2: Standard Library Modules

Module	Description
os	Miscellaneous operating system interfaces.
sys	System-specific parameters and functions.
re	Regular expression operations.
time	Time access and conversions.
subprocess	Subprocess management.
collections	Container datatypes.
shutil	High-level file operations.
itertools	Functions creating iterators for efficient looping.
functools	Higher-order functions and operations on callable objects.
json	JSON encoder and decoder.

We present a list with different domains and responsibilities and what modules contribute to it.

- Testing Framework: `unittest` module was the most used, being present in 80 projects, followed by `pytest`, present in 61 projects. In third place, we have the `doctest` with 16

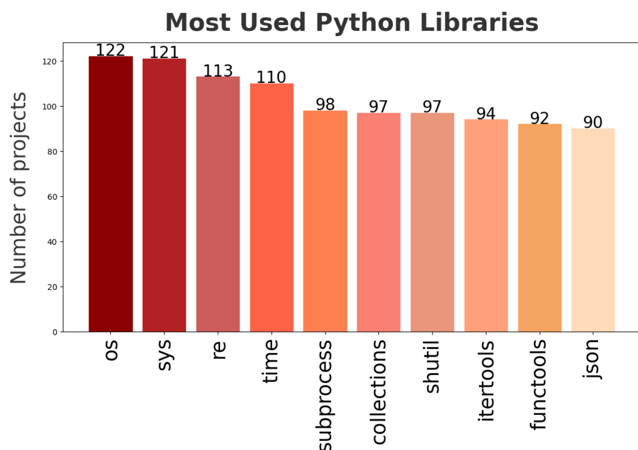

Figure 8: Standard Modules

projects. Comparing with the grey literature, the information diverges. In some blogs, the framework `robot` is cited as the most used, followed by `pytest` and, in third place, `unittest`. Other frameworks considered popular did not appear in the final result obtained by *PySniffer*. T

- Web Scraping Tools: The `urllib` module was present in 80 projects, followed by `requests` (62 projects) and `beautifulsoup4` (19 projects). This finding slightly differs from the information on some blogs and websites. In these, the `request` is considered the most popular, followed by `lxml` and, finally the `beautifulsoup4`. Although `lxml` is not present in the top 3 modules found by *PySniffer*, it was present in 14 projects, a significant number considering that the most popular projects on GitHub were selected.
- Data Science and Machine Learning: The `numpy` module was found in 58 projects, followed by `scipy` (40 projects) and `pandas` (27 projects). These are well-known libraries, and several works in the literature discuss them [1].
- Data visualization: `matplotlib` module was the most used, being present in 32 projects, followed by `seaborn` used in 6 projects, and `plotly` in 5 projects. In one of the research sources, the disclosed data corroborate the result found by the developed tool.
- APIs Frameworks to build API: The `flask module` was found in 20 projects, being the most used, followed by `django` (6 projects) and `fastapi` (1 project). These data differs slightly from the information on some blogs and websites. In these, `django` is considered the most popular, followed by `flask` and then `falcon`.
- Cloud: The `boto3` was the top used, found in 16 projects, followed by `azure_storage` (7 projects) and `pyicloud` (1 project). For this area of study, we could not find information in other sources indicating which module was considered the most popular. However, the `boto3` is a module geared towards development with Amazon Web Services (AWS), one of the most popular cloud services. This way, the result found servers as support to the popularity of AWS.
- Database: The `SQLAlchemy` module, found in 13 projects, was the most used, followed by `pymongo` (9 projects) and `sqlparse` (5 projects). These data differs from the information obtained from the literature. In these, the most popular modules are `mysql.connector`, `sqlite3` and `psycopg2`.
- MQTT: The `paho-mqtt` module was the only one found in the category. In gray literature, `hbmqtt` and `gmqtt` modules appears as options to work with mqtt, but `paho` is predominant, which confirms the obtained results.

We extracted the top used modules and presented their responsibilities, creating a discussion of the top features required and used by popular open-source Python projects hosted in GitHub. From this discussion, it is clear that unit testing is a very popular feature used nowadays to measure software quality. Furthermore, data visualization is also very popular among Python projects, given that this language is known to be strong in this field, as opposed to Java or C#.

We obtained results from an empirical study conducted through mining software repositories. The goal was to provide a reference list of top used modules in popular Python projects and complement lists found in blogs and personal websites. Developers, however, should decide which library or module they will use based on their requirements and needs. Our work can aid this process not as a final decision but as another variable to consider.

7 RELATED WORK

In this section, we present the work of other researchers that used AST to perform static code analysis and mined software repositories to study the usage of both external and standard libraries in open-source software. Even though we are interested in Python in this work, we present research carried out in other programming languages to generalize the research.

The work of [15] proposes the *APIScanner*, a tool developed to search and list deprecated APIs (Application Programming Interface) in Python libraries and, consequently, unused. They saw a need to bring this kind of technology to the Python language as it was found just for static programming languages like Java. This way, a team developing a Python project could know that the API they are using is deprecated in the beginning or middle of the project's course, not just when finished. This tool uses Abstract Syntax Tree (ASTs) and Decorators. To validate their work, the authors applied their approach to six famous Python libraries: NumPy, Matplotlib, Pandas, Scikit-learn, Scipy and Seaborn. As a result, the tool detected 838 out of 871 elements in the six analyzed libraries. The use of ApiScanner helps developers save time from analyzing documentation to find out whether or not API "X" is obsolete. Compared with our work, we do not investigate the presence of deprecated libraries, we instead focus on the usage of such libraries and the most popular ones.

The work of [16] proposes *MigrationMiner*, a tool that detects code migration that has been carried out between third-party Java libraries in projects hosted in GitHub. A simple approach is to inspect changes in the pom file and report what library was replaced. It can also search how the methods calls were migrated from the retired

library to the new library. This tool was developed since library migration is a very tedious process, and developers need to identify and understand all their particularities, which can be time-consuming. The tool searches for patterns present on library migration and can aid developers by suggesting the changes while they refactor the code to a new library.

Regarding the Java ecosystem, several frameworks and libraries use a metadata-based approach exposing their functionalities in the form of code annotations. The work of [14, 17] performed static code analysis on GitHub projects to measure how code annotations were present. It reported that the majority of external libraries found was related to testing (JUnit), object-relational mapping (JPA) and web controllers (Spring).

Finally, the work of [18] investigates an approach to identify experts in specific libraries or frameworks based on their contributions to GitHub projects. They are targeting experts in 3 JavaScript libraries: `react`, `node-mongodb` and `socket.io`. To reach their goals, they combine the areas of mining software repositories and machine learning. As a result, they were able to recommend dozens of GitHub users with robust evidence of being experts in`react`.

8 CONCLUSION

This paper conducted an empirical study to obtain the top used modules in popular Python projects hosted in GitHub. Several blogs and personal websites contain a list of popular libraries, and the list that we obtained empirically can complement this data found on the web. Empirically obtained lists might also better reflect what developers are using to build their software, and we can discuss what features are being the most requested.

The first step was to select popular projects hosted in GitHub, which yielded a list of 129 projects unfolding in 57201 ".py" files to obtain this list of used modules. Then, to automate the extraction process, we developed an open-source tool called *PySniffer*. The tool can download the selected projects, extract the top used modules, and compare them with modules obtained from another specific repository. *PySniffer* can also generate a bar chart with the ten most used modules both from the standard library and third-party libraries. Finally, we developed a web application that brings these results closer to end-users through GUI and statistics information.

From our discussion, we saw that one of the most used features was related to testing, which is a practice that has become widespread in the software engineering community, and we validated this from our empirical finding. Other popular libraries that we found are related to web scraping and data science. With our list and tools publicly available, other practitioners and researchers can use them to analyze their projects.

We have the selected projects and the base chosen as a threat to our work. We considered the top 129 projects that matched our criteria. Even though we presented these criteria in detail to ease reproducing this list, there is still room to generate a list slightly different from ours. Therefore, the used modules might differ. Furthermore, we only considered projects hosted on GitHub, and projects hosted on another base may present a different profile.

For future work, we would like to automate and customize the analysis by domain, i.e., choose projects related to web development, then projects of data analysis, then tools, and so forth. This way might better understand developers' needs depending on what type of systems they develop. With this functionality, *PySniffer* might also be able to offer tips on the most used libraries for each area. In addition, it is proposed to improve the web application to execute the tool or to request data directly from an API. Another improvement is the automation of the collection and selection of Open-Source repositories from GitHub, focusing on keeping the data up to date and allowing users to pass in custom criteria.

REFERENCES

[1] Sebastian Raschka, Joshua Patterson, and Corey Nolet. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information*, 11(4), 2020. ISSN 2078-2489. doi: 10.3390/info11040193. URL https://www.mdpi.com/2078-2489/11/4/193.

[2] Sandeep Nagar. *Introduction to Python for Engineers and Scientists: Open Source Solutions for Numerical Computation*. Apress, 1st edition, 2018. ISBN 978-1-4842-3203-3, 978-1-4842-3204-0.

[3] Maurice J. Thompson. *Python Programming For Beginners - Learn The Basics Of Python In 7 Days!* (n.p.), (n.p.) edition, 2018. ISBN 1980501114, 978-1980501114.

[4] KR Srinath. Python–the fastest growing programming language. *International Research Journal of Engineering and Technology (IRJET)*, 4(12):354–357, 2017.

[5] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, and Massimiliano Di Penta. Crossrec: Supporting software developers by recommending third-party libraries. *Journal of Systems and Software*, 161:110460, 2020. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.110460.

[6] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can i use this method? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 880–890, 2015. doi: 10.1109/ICSE.2015.98.

[7] Riccardo Rubei, Claudio Di Sipio, Phuong T. Nguyen, Juri Di Rocco, and Davide Di Ruscio. Postfinder: Mining stack overflow posts to support software developers. *Information and Software Technology*, 127:106367, 2020. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2020.106367.

[8] Hudson Borges and Marco Tulio Valente. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2018.09.016. URL https://www.sciencedirect.com/science/article/pii/S0164121218301961.

[9] Mark Lutz. *Learning Python*. O'Reilly Media, Inc., 4th edition, 2009. ISBN 978-0-596-15806-4.

[10] Phyllipe Lima, Eduardo Guerra, and Paulo Meirelles. Annotation sniffer: a tool to extract code annotations metrics. *Journal of Open Source Software*, 5(47):1960, 2020. doi: 10.21105/joss.01960. URL https://doi.org/10.21105/joss.01960.

[11] Maurício Aniche. *Java code metrics calculator (CK)*, 2015. Available in https://github.com/mauricioaniche/ck/.

[12] I. Neamtiu, J. S. Foster, and M. Hicks. Proceedings of the 2005 international workshop on mining software repositories. In *Understanding source code evolution using abstract syntax tree matching*, pages 1–5, 2005.

[13] Vartika Agrahari and Sridhar Chimalakonda. Ast[ar] – towards using augmented reality and abstract syntax trees for teaching data structures to novice programmers. In *2020 IEEE 20th International Conference on Advanced Learning Technologies (ICALT)*, pages 311–315, 2020. doi: 10.1109/ICALT49669.2020.00100.

[14] Phyllipe Lima, Eduardo Guerra, Paulo Meirelles, Lucas Kanashiro, Hélio Silva, and Fábio Silveira. A metrics suite for code annotation assessment. *Journal of Systems and Software*, 137:163 – 183, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2017.11.024. URL http://www.sciencedirect.com/science/article/pii/S016412121730273X.

[15] Aparna Vadlamani, Rishitha Kalicheti, and Sridhar Chimalakonda. Apiscanner - towards automated detection of deprecated apis in python libraries. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings(ICSE-Companion)*, pages 5–8, 2021. doi: 10.1109/ICSE-Companion52605.2021.00022.

[16] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. Migrationminer: An automated detection tool of third-party java library migration at the method level. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 414–417, 2019. doi: 10.1109/ICSME.2019.00072.

[17] Phyllipe Lima. Assessing code annotations usage in software projects, 2021-09-16 2021. URL http://urlib.net/ibi/8JMKD3MGP3W34T/45DA8DH.

[18] João Eduardo Montandon, Luciana Lourdes Silva, and Marco Tulio Valente. Identifying experts in software libraries and frameworks among github users. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 276–287, 2019. doi: 10.1109/MSR.2019.00054.