Python Library Dependency Analysis

By Steve Han

Author Bio

Steve Han is a motivated student in Toronto who is highly interested in computer science and research. He is engaged in many volunteer and internship opportunities to enhance his coding skills. He hopes to improve people's daily lives by researching various issues to support his community.

Abstract

Python is a high-level programming language used by software engineers, computer scientists, and mathematicians. The purpose of this research is to help with the future development of the Python language, such as the optimization of the PIP manager and choosing built-in packages for Python. The main goal of this research is to analyze the pattern in the dependency network and how it changed after 2016. Using the official API provided by PyPI, a network of all related packages was made. After filtering any incomplete or irrelevant nodes, the entire graph is 10x larger than the research done by Kevin Gullikson in 2016. By ranking the number of edges, connectivity, as well as degree distribution of the nodes, a shift into machine learning and data analysis can be seen in the Python community. Some required packages shifted and reflected the current technological trend in recent years, such as Numpy for machine learning and Django for web development. Future studies are recommended to further investigate any particular changes in the usage of Python.

Keywords: Python; Graph analysis; PIP; NetworkX; Web-scraping; Graphing; PyPI; Data Parsing.

Introduction

Python has developed a huge package dependency tree throughout the years. To install a Python package, one only has to type the package's name and press enter. It was often thought to be a simple download from the library archive. However, when a package is installed, it does not just install one package in most cases. Every package has a requirements list that includes packages and versions of the packages that it requires to run.

While using the Python language, it would be very inefficient to code everything from scratch. For example, a data analyst uses mathematical libraries to do complex arithmetic on a large scale of data and graph drawing packages to present their results in a readable format. While possible, it would waste valuable time to reimplement these pre-existing codes for every occasion the user needs them. Users can package the codes they wrote and publish them on the Internet and then simply use Python's package manager to reuse the code so that people can focus on their analysis and research.

Python uses PIP to manage packages. PIP is a recursive acronym for "Pip Installs Packages." Recursive acronyms are a common feature in many computer programs, where the acronym is found in the full name. PIP simplifies many of the problems that might occur during the installation of packages. For example, import loops (where package A requires B, but B requires A, creating a loop) and dependency resolution (picking the right version of a package when multiple packages require a different version of the same package) are all handled during the installation process. We can visualize the data as a graph based on the packages' requirements. A detailed definition of Dependency Analysis can be found in the IBM documentation (IBM Documentation, 2021).

PIP searches the dependencies required by the package and automatically installs them. PIP is a smarter tool than most people think it is. The relationship between packages can be represented on a directed graph. Python's dependency network will be analyzed in this paper.

The dependency requirements between packages can also be represented as a directed graph. A graph, or a network, shows the connection (edges) between many individual things (nodes). For example, addresses and streets are a network where buildings are connected through many streets. However, graphs can be virtual as well. Bitcoin wallets and transactions are also a form of network, and social media accounts and subscribers can also be represented as a graph. In this case, every node represents a python package, and a directed edge represents the required dependency a package needs. A directed graph means that the connections between nodes are one-way only. It is a directed edge or connection because the relationship can not be reversed. If package A requires package B to function, then the directed relationship would be A -> B, but saying B -> A would not make much sense because B works fine without package A. For the simplicity of the research, the required versions of dependencies are ignored. An example of a library that visualizes module dependencies and checks for circular dependencies is Madge, but it's designed mostly for Javascript and CSS instead of Python (Github, Pahen/Madge, 2021).

For this paper, the researchers analyzed a few aspects of a network. The degree of a node is the number of connections a node has. In this case, the degree will represent how frequently a package is used. The PageRank algorithm shows each node's relevance, similar to how Google ranks its search results. The connectivity of a graph is the minimum number of elements that need to be removed in order to separate the graph into two separate ones. Communities are defined as a subset that is densely connected to other nodes in the subset and loosely connected to other communities in the graph.

People use Python due to its simple syntax and universality, which was made possible by the hundreds of thousands of different packages available on PyPI. This research ranked the commonly referenced packages in the dependency network through different measurements. This research also compared the current data with Kevin Gullikson's results in 2016 to see if anything has changed over time (Gullikson, 2016). It is predicted that low-level packages will have the highest usage because of their functionality. This analysis aims to help the development of the Python language by integrating some of the common functions into Python itself, as well as gain insight into how people use the language and help with many Python Enhancement Proposals that currently exist.

Methods

The first step would be to rerun the program to fetch the newest dependency data from PyPI using the same program provided in Gullikson's analysis. However, this is no longer feasible because PyPI discontinued the search function long ago, meaning the program no longer works. Articles of analysis for other languages were examined to form a solution for this problem during the research. Another solution found during the research is the Simple Repository API provided by PyPI. It is a simple index page for all Python packages formatted for program use. The PyPI-simple library was used to retrieve the list of packages, and the PyPI JSON endpoint was called to collect the dependency data of each library.

```
from pypi_simple import PyPISimple
import json
import requests

'''
with PyPISimple() as client:
    index = client.get_index_page()
    print('done')
    with open('2022.json', 'w') as f:
        f.write(json.dumps(index.projects))
'''

with open('2022.json', 'r') as f:
    index = json.loads(f.read())
    url = 'https://pypi.org/pypi/{}/json'
    with open('fetch.csv', 'w') as f:
        for project in index:
            data = requests.get(url.format(project))
            if data.status_code != 200:
                continue
            data = data.json()
            if data['info']['requires_dist'] is None:
                continue
            data = [x.split(' ')[0] for x in data['info']['requires_dist']]
            for each in data:
                f.write(project + ',' + each + '\n')
```

Figure 1, code used to download data from PyPi

The Python code in Figure 1 scraps data from the PyPI website and makes a list of dependencies. After generating the file, the data lists all of the packages on the left side of each line and their dependency on the right. In the case of multiple dependencies, there will be multiple lines of the same package with different dependencies. The graph is relatively large, and some of the analyzing methods might be too inefficient; this problem will be addressed later in the article. This paper uses NetworkX and other data visualization tools to analyze this graph and provide evidence to prove or disprove the hypothesis. The main analyzing approach would be relatively similar to older articles related to this topic, but with the more recent dataset to see how the dataset has shifted after their analysis.

The first noticeable change is that the network has grown significantly since 2016. Back in 2016, Gullikson was able to process 20,522 out of around 74,000 packages. The current graph includes 179,565 out of 383,450 packages and 788,318 edges. Python has likely grown drastically over the years. This became an issue due to the required time to fetch the entire network, but the script was eventually able to download a good portion of the data. During the process, packages that do not require other dependencies are excluded from the graph. It is understood that there are exceptional packages without any required dependencies, such as BeautifulSoup4 (an HTML parsing library), but they do not affect the graph as a whole. Versions of the required packages are also omitted since they are not the main focus of this research.

NetworkX and Matplotlib were used to analyze the data. NetworkX is a Python library for creating and manipulating graphs. It has many built-in methods that simplify the work, and it does all the heavy lifting math behind the scenes. Matplotlib is a plotting library that works very well with NetworkX. Kevin Gullikson's program was rerun for connectivity, degree distribution, and communities of the new graph. The code in Figure 2 is the process of generating the "betweenness" of a dataset.

```
bc = nx.betweenness_centrality(dep_graph)
sorted_dict = sorted(bc.items(), key=operator.itemgetter(1))[::-1]

N = 10
x = np.arange(N)
y = np.array([d[1]*100 for d in sorted_dict[:N]])
xlabels = [d[0] for d in sorted_dict[:N]][::-1]
fig, ax = plt.subplots(1, 1, figsize=(7, 7))

ax.barh(x[::-1], y, height=1.0)
ax.set_yticks(x + 0.5)_ = ax.set_yticklabels(xlabels)
ax.set_xlabel('Betweenness Connectivity (%)')
fig.subplots_adjust(left=0.32, bottom=0.1, top=0.95)

fig.savefig('Figures/Betweenness.png')
```

Figure 2, code used to calculate the betweenness by Kevin Gullikson (2016)

Several plots were generated and compared to the original graph. The analysis showed related results to the older analysis, but new changes were also found in the new graph, demonstrating how Python has evolved in the past few years. Produced charts will be shown in the following section, as well as some analysis and interpretation. However, it is worth noting that the JSON API used in this research might not be

100% accurate and up to date. These errors are small enough to be ignored in such a large dataset.

Results

This analysis came to many interesting conclusions. First, both the network from 2016 and the one from 2022 showed similar results for the degrees of nodes, as can be seen in Figure 3 and Figure 4.

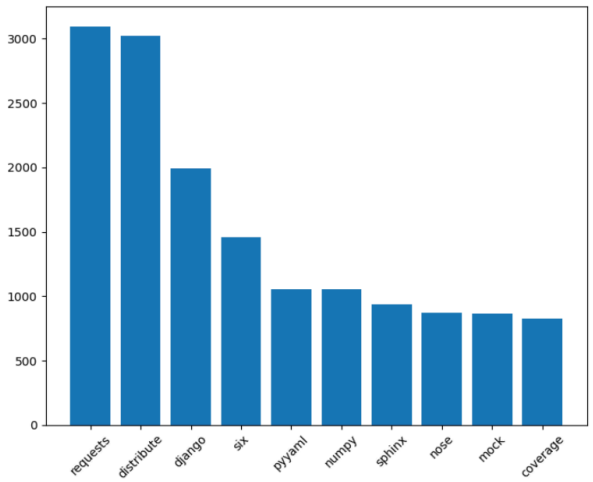

Figure 3, Graph degree in 2016

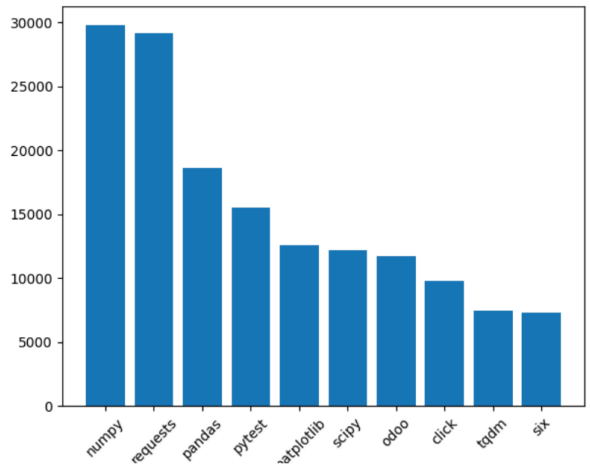

Figure 4, Graph degree in 2022

The data relatively stayed the same, and the degree count showed relatively the same trend. Even though the size of Figure 4 is 10 times bigger, the graph showed a decreasing trend of degrees within the first few packages, which is relatively similar to previous data. However, the major difference is that NumPy became very popular in the last six years, and Django is no longer seen in the plot.

Although the most popular package is still the requests package, other scientific analysis packages such as Numpy and Pandas also rose their rank to the top. A possible reason can be the new breakthroughs and discoveries in AI and machine learning, causing more and more companies and users to get into the field. Further studies are required regarding this hypothesis.

Testing packages also had a high rank in data from both times. Testing packages are mainly to ensure high-quality code and measure production metrics. Examples of these packages would be pytest, coverage, and mock. It is understandable that these universal tools are used quite often since most software in production uses these tools to assist with the development process.

The betweenness connectivity of the graph, the percentage of shortest paths between every two nodes that go through each package, is analyzed next. Figure 5 and Figure 6 showed both graphs' connectivity in 2016 and in 2022.

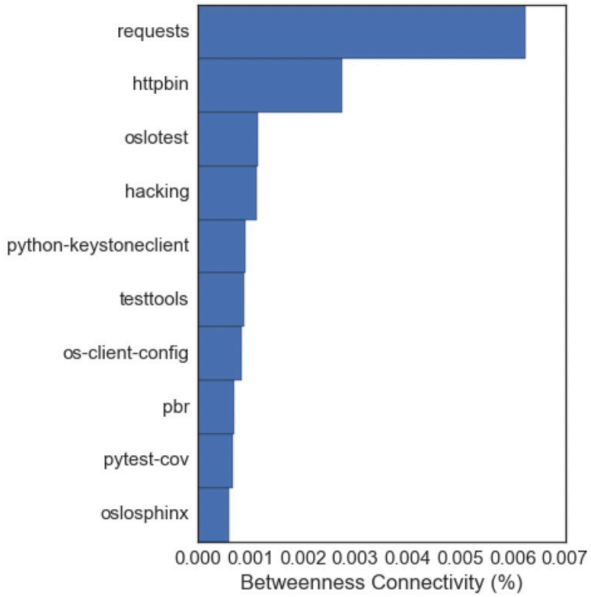

Figure 5, Connectivity in 2016

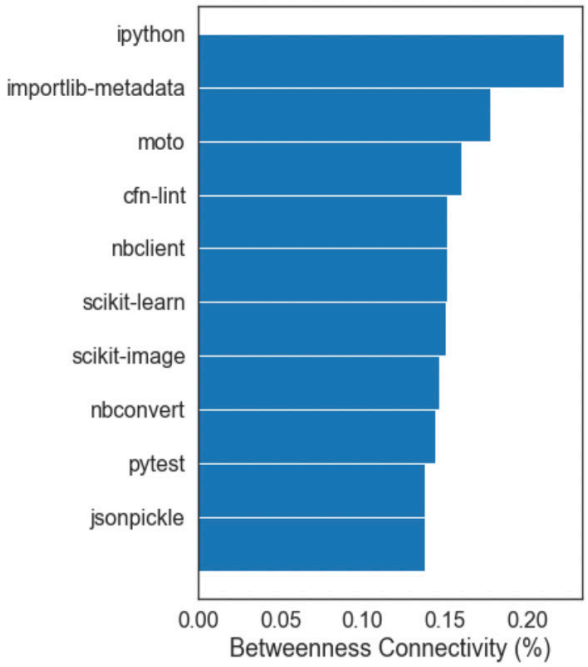

Figure 6, Connectivity in 2022

In Figure 6, the decreasing trend is significantly less dramatic. The Requests package was widespread in the dataset, but IPython and other testing libraries are replacing it. It is possible that the reasons why people use Python have shifted over the years or that the older data has been filtered out of the traditional data. Keep in mind that the network in this research is also significantly larger, which might also give us a more accurate result.

The last plot that was produced in this research is the degree distribution. A degree distribution plot describes how many packages require how many dependencies. The data from 2016 and 2022 are shown in Figure 7 and Figure 8.

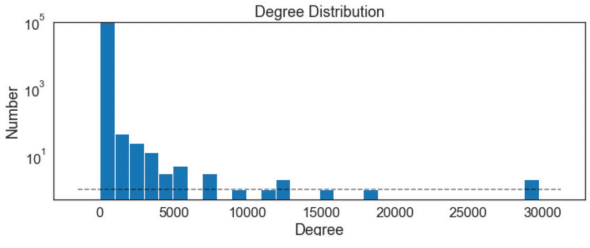

Figure 7, Degree distribution in 2022

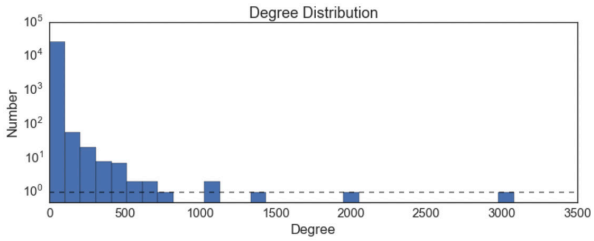

Figure 8, Degree distribution in 2016

The plot showed a similar decreasing trend, with fewer packages requiring more dependencies. It showed that most packages require 0 to 700 packages recursively. As Gullikson mentioned (2016), "A likely explanation is a 'rich get richer' scenario: packages that already have lots of stuff that uses them show up high on Google search results, and so new package developers use them too. For example, almost all scientists use the matplotlib plotting package, and so their code all requires matplotlib".

Conclusion

The results of this research revealed a great deal of exciting data indeed. The first difference is how the size of the graph grew through the years, to nearly 10x its 2016 size. This is strong evidence that Python's popularity has been growing drastically. With more tools being developed, more users will be attracted to the language, and more effort will be put into it. Python has a healthy community that is constantly growing, and this research predicts that it will continue to dominate the programming field.

Many important packages can be seen in this research. The most popular package is the Request package, which can be seen on the top in many plots. One possible reason is that web scrapers usually use Python to write their scripts, and requests became an essential dependency because it allows users to send HTTP requests and simulate actual browser headers easily. However, it still should not be a built-in package because there are many use cases for Python that have nothing to do with networking or web requests.

Python also seemed to become universal over the past six years. Even though the exponential trend still exists, it is seen in many plots that the trend is flattening between the top first few packages. One way

to interpret this is that Python has shifted from web-related development to many other fields. However, this could also be due in part to incomplete data or incorrectly filtered results, which caused biased data. It is best to reexamine this issue before further research is concluded.

It is worth noting that the relations between packages do not completely represent a package's popularity. For example, a library meant for end-user usage might not have a lot of packages depending on it, but it doesn't mean that it is an unpopular package. However, dependencies can show, to some degree, some low-level package's importance in the community and its importance. To fully investigate the popularity of packages, other metrics such as Github stars and the number of pull requests might better represent such topics.

In conclusion, the data analyzed in this research shows many exciting results, and the hypothesis mentioned at the beginning is mostly correct. There are more packages that showed significant usage than expected, but it also revealed how Python is used for all sorts of purposes, and one perspective is only a fraction of what the language is capable of doing. Based on the more detailed results, it is recommended that more research be put into this graph in order to decide whether some packages should be included in Python.

References

Bommarito, E., & Bommarito, M. J. (2019). An Empirical Analysis of the Python Package Index (PyPI). SSRN Electronic Journal. https://doi.org/10.2139/ssrn.3426281

Girardot, O. (2013, January 5). State of the Python/PyPi dependency graph. O. Girardot. https://ogirardot.wordpress.com/2013/01/05/state-of-the-pythonpypi-dependency-graph/

Github, 'Pahen/Madge', 2021, https://github.com/pahen/madge

Gullikson, K. (2016, February 18). Python Dependency Analysis — Adventures of the Datastronomer. Kgullikson88.Github.io. https://kgullikson88.github.io/blog/pypi-analysis.html

IBM Documentation, 2021, https://www.ibm.com/docs/en

Korkmaz, G., Kelling, C., Robbins, C., & Keller, S. (2019). Modeling the impact of Python and R packages using dependency and contributor networks. Social Network Analysis and Mining, 10(1). https://doi.org/10.1007/s13278-019-0619-1