Understanding legacy problems in Python: a study of versions, frameworks and update behavior

Author: Ashwin Barendregt
Student number: 2584715
E-mail: a.j.barendregt@student.vu.nl
Study: MSc Information Sciences
Supervisor: Dr. Siewert van Otterloo
Second reader: Jeroen Arnoldus
Date of Submission: August 22, 2023

Abstract. This thesis research was conducted on the topic of Python and framework version update strategies and the update behavior from Python developers. This was done through both version history documentation analysis and survey research. Highlighting the update strategies of Python and three specific frameworks; Django, Flask and Pyramid showed various degrees of consistency in version updates. Python and Django were found to be consistently following a version update release strategy while Flask and Pyramid were found to be far more inconsistent. This lack of consistency in update releases was also found in the speed of Python compatibility update releases. While Django showed to be consistently faster with its Python compatibility updates, Flask and Pyramid were found to be either inconsistent or not transparent with its Python compatibility update releases. Through the survey, several findings were documented. Firstly, respondents showed that they were more likely to keep up with changes made in Python version updates than they were with framework version updates. Secondly, They were found to mainly update their Python and frameworks versions at least once a year, as opposed to once every two or three years (or more). Thirdly, the framework specific responses were highlighted where Django and Flask were found to be the most used frameworks, where users showed similar update behavior to the general Python and framework update behavior findings. Lastly, comparisons were made to the results of existing survey research on aforementioned topics and the general state of Python as a programming language. The latter mentioned comparisons showed similar results to for instance the survey results of the state of JavaScript.

Keywords: Programming · Python · Framework · Versions · Updates · Survey.

1 Introduction

1.1 Motivation

The Python programming language has been among the top programming languages used in the world for a number of years now, as is for instance highlighted through search index methods such as PYPL and TIOBE [1,2]. To get a better understanding of the language itself and how shortcomings and challenges are dealt with by those who work with the language frequently, many factors can be highlighted. One can think of factors such as the availability of libraries and frameworks, the contexts in which the language can be applied, as well as the general amount of people in the world that use the language and share ideas. One other important factor is longevity or in other words, how future proof it is to code with Python. Developers can choose to use a programming language and available frameworks, packages and libraries to develop their ideas, but if the code becomes dated after a certain number of years, their work can ultimately become a burden to use, or will be completely unusable. Through analysis of the history and current plans for rolling out different versions of Python and its available frameworks as well as through conducting a survey related to this topic, a research can be conducted. This will result in a detailed insight into the longevity of the language and the update behaviors and other stances that developers might have regarding the process of transitioning to newer versions.

1.2 Problem definition

Because longevity can be highlighted as an important part of programming, keeping up with newer versions of Python is vital. Developing something with one version of Python and/or a relevant framework, might mean that it eventually needs to be upgraded to a more recent version. This therefore requires certain upgrade strategies in order to facilitate the transition to a newer version. This poses challenges not just regarding the Python language itself but also in relation to the potential use of libraries, packages and frameworks. These challenges can for instance occur when security threats and vulnerabilities are discovered [3]. When these vulnerabilities are discovered, an update for a resource like a Python package and a notice then get released, and it is up to developers to keep track of this and subsequently update their code in order to adhere to the more secure use of the package [4]. If this does not happen, these security vulnerabilities will remain and can for instance lead to system breaches and ill-intentioned exploitations from attackers. Upgrade strategies can also for instance pose challenges when there are more general changes made to Python itself, such as changes in syntax. Blog Posts by developers such as [5], highlight examples of this. Overall, it is therefore beneficial to have knowledge and understanding of the current state of this subject domain. Predominantly regarding Python and framework version update release strategies and how developers might adapt to these strategies in their work.

1.3 Research questions

RQ1: What does the history and rollout of different versions of Python and its available frameworks look like over-time?
SRQ1.1: How fast have frameworks been updated to be compatible with a newly released version of Python?

Through research question 1, the goal is to identify how those behind Python and various frameworks release different versions and for what reasons. This will result in a deeper understanding of how this could impact the work of developers that use the language and frameworks. Particularly on how this can pose challenges within the process of ensuring longevity through keeping up with different version releases of Python itself and frameworks. These questions can be answered through literature and Python/framework documentation - study

RQ2: What do the behaviors and stances look like from Python developers in regard to updating versions of Python and their used frameworks?

The ultimate goal of research question 2 is to identify concrete update behaviors and stances on the state of Python and frameworks in general and their update release strategies. The answers to this research question can be informed through the process of conducting a survey with Python developers. This in turn also offers the opportunity to compare the findings to the existing studies on this topic, and highlight results for specific frameworks.

1.4 Scientific and practical contribution

Because of the descriptive nature of the proposed research, the resulting thesis will contribute to understanding the current state of Python and framework version update strategies and what impact that has on the way that developers react to these strategies, if at all. This subject domain touches on aspects that can both be of interest to programmers inside and outside of the Scientific landscape.

2 Related literature

When it comes to Python and Framework documentation, as well as specific adjacent research done on the proposed topic for this thesis project, a number of them can be highlighted.

In terms of Python and frameworks in general, Python [6], Django [7] and Flask [8] for instance all respectively provide documentation on the history of different versions/releases, and in the case of the frameworks, offer information on the history of releases and which are compatible with which Python version(s). The latter is predominantly done through change logs or release notes, as is also shown in [8]. On top of this documentation style of resources from the developers behind the language and frameworks, there are also publications by outside parties such as with [9], which provides a book facilitating an in-depth overview, along with code snippets, on the differences and compatibility aspects between Python versions 2 and 3.

When it comes to publications that are more focused on the developer point of view, there are also various research efforts to be found. For instance, [10] highlights the willingness from app developers to fully transition from python 2 compatibility to Python 3 through quantitative analysis of app update pass rates in different Python versions. Additionally, there are also publications dedicated to finding solutions for transitions from Python 2 to 3, for instance through code conversion using machine translation [11]. Publications with this kind of goal are also present through presenting general tips and methods, such as this publication from the Python developers team [12]. In terms of the frameworks, there is already research done comparing Django and Flask, as is for instance shown in [13,14], where the benefits and shortcomings of each of the two are highlighted. In terms of the theme of longevity, a study like [15] highlights the evolutions of various Python framework APIs while also taking compatibility issues into account.

Ultimately, there are also many frameworks that can possibly be included. As described in [13], the kinds of frameworks can be separated into three categories: Full stack, Micro and Asynchronous. The main difference between full stack and micro frameworks is that full stack provides tools and components for the entire development process, while micro frameworks have less of those tools and components available. These micro frameworks therefore require for more code to be written, which could in turn pose different challenges in the transitioning process from one version to the next. Asynchronous frameworks are a more recent trend, which are similar to micro frameworks but allow for handling many concurrent processes and connections.

2.1 Security vulnerabilities, standards and regulations

As was already briefly mentioned in section 1.2 , the topic of security is also synonymous with the concept of version histories in programming, especially in relation to web development. This importance is highlighted by for instance regulations related to data protection through the General Data Protection Regulation established by the European Union [35]. The Open Worldwide Application Security Project (OWASP) is a notable foundation that proposes standards for keeping applications secure. Through the annually conducted community survey by OWASP for 2021 [37], one of the most prominent issues that were highlighted was the concept of Vulnerable and Outdated Components. This was therefore also documented in the official OWASP Application Security Verification Standard [36]. As seen in the block quote of this section, the recommendations include that components and tools are up to date. Furthermore, it also mentions the importance of the reliability and integrity of third-party components, depending partially on whether or not they are maintained over time. Anything that requires maintenance over time also come with updates that need to be kept in check and managed.

Block quote 1

In the application Security Verification Standard section V14.2, titled Dependency [36] a number of important rules are established:

- Verify that all components are up to date, preferably using a dependency checker during build or compile time.
- Verify that all unneeded features, documentation, sample applications and configurations are removed.
- Verify that if application assets, such as JavaScript libraries, CSS or web fonts, are hosted externally on a Content Delivery Network (CDN) or external provider, Subresource Integrity (SRI) is used to validate the integrity of the asset.
- Verify that third party components come from pre-defined, trusted and continually maintained repositories.
- Verify that a Software Bill of Materials (SBOM) is maintained of all third party libraries in use.
- Verify that the attack surface is reduced by sandboxing or encapsulating third party libraries to expose only the required behaviour into the application.

The first rule stated in the quoted section of the Security Verification standard also includes the mentioning of dependency checkers. This and other tools also play an important part in checking for - and automating- the update process for dependencies used. Pip and its dependency resolution tool [38] are for instance recommended and a standard part of a Python installation and help with detecting and installing the best compatible versions of certain packages used in a project. Popular development platform GitHub recommends their own tool named ”dependabot” [39], which scans project repositories consisting of many diferent programming languages including Python, in order to detect which used dependencies require updates in order to maintain compatibility.

A further takeaway from the OWASP top 10 overview (A06) [37] is that, despite the presence of these tools, there is still plenty of reliance on a pro-active stance from developers to use these tools in a timely fashion and frequently test for compatibility of versions, old or new. OWASP for instance further recommend the continuous inventory of used versions for frameworks and dependencies, and subscriptions to emails and other online notification methods in order to be alerted of recently discovered security vulnerabilities found in dependencies used in a project.

3 Research strategies and methods

The proposed research can be categorized to have both qualitative and quantitative elements. The qualitative data collection that will be used to answer the proposed research questions are through an elaborate documentation and literature study as well as a survey that highlights the version update stances and behaviors of respondent Python developers.

In order to more broadly analyze the roll-out histories of different versions, the goal for this research is to analyze Python frameworks from multiple categories, namely Full stack frameworks and Micro frameworks. The focus will be put on Frameworks that have been available for a longer period of time, and therefore allow for analyzing version histories and projects that have been around to have compatibility for many Python versions, including the big transition from Python 2 to 3. Since Asynchronous Frameworks have been a recent trend, they do not cover this larger period of deployment and compatibility history with different Python versions, and will therefore not be analyzed for this research.

Full stack frameworks:

- Django [16]
- Pyramid [17]

Micro framework:

- Flask [8]

Django, Flask and Pyramid were chosen as they are all Python frameworks for web development. Furthermore, they have been amongst the most used frameworks in recent years [28,29]. They also all have been around long enough to have gone through periods where it was used for Python 2 only, during its transition to Python 3, and during more recent years where they are fully migrated over to just Python 3 compatibility.

Ultimately, the method will more concretely consists of the following parts:

- Evaluating version rollout plans of Python, over a set period of 10 years.
- Evaluating version rollout for the chosen frameworks over the same period of time.
- Evaluating the speed with which the parties behind the chosen frameworks pushed for compatibility updates.
- Conduct a survey with developers who have experience with working with Python and frameworks, Thereby making efforts to document their stances on upgrading their work to be compatible with newer Python versions.

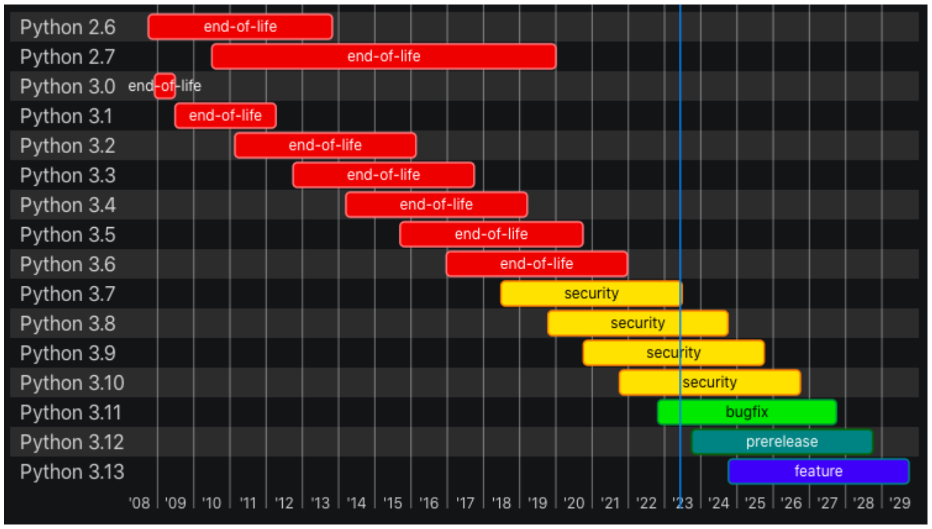


Fig. 1: The Python version release cycle, as officially published by Python [6].

4 RQ1 - The roll out of new Python and Framework versions

4.1 Python version update strategy

Since 2008, there have been as many as 15 different minor Python version releases. With each release having its own cycles of time in which that version is supported before it reaches "end-of-life" status, which essentially indicates how long a version is subject to change through for instance bug and security-fixes [6]. Figure 1 highlights the release cycle of different Python versions, as officially published by Python. As is stated in more detail in official Python documentation found on their websites [18]. These kinds of changes, are referred to as either "minor" or "micro", where minor changes include fixed features that have been on an in-development list, and are included in an annual update push for a particular major version, while micro changes are maintenance-focused, and get pushed in monthly updates. To illustrate; The Python release cycle overview only indicates the major (i.e. version 2 or version 3) and minor (i.e. version x.6 or x.7) levels of a release. However, in between those releases, there have also been pushes for updates on a micro-level which are for instance referred to as 2.6.1 , thereby highlighting the use of a "Major.Minor.Micro" nomenclature. When analyzing the Python release cycle overview, it therefore shows that there is roughly one year between each minor release. Furthermore, it showcases that most of those minor releases have an eventual life-span of roughly five years before they reach end-of-life status, with the exception of versions —2.7 , 3.0 and 3.1—. These Python versions in particular are the last minor release of the major version Python 2 and the early minor releases of major version Python 3 respectively. This transition from one major version to the next can be looked at in more detail to highlight what significant changes have been made and by extension, how this can impact the releases of Python frameworks, and the work of developers in general that use them.

4.2 The transition from Python 2 to 3

As stated in section 4.1 , the lifespan of a Python version before it reaches end-of-life status is roughly five years. The only big exception to this trend has been the particular version Python 2.7, which ended up being supported for roughly 9 years before it reached its end-of-life. This can be seen in figure 1 as well. This means that there were multiple Python 3 versions already available for developers to use during the time that python 2 was still supported. As a result, developers were faced with the option to either completely transition to Python 3.x and thereby abandon Python 2 completely, or to make their coding work compatible with both version until version two permanently reached end-of-life status at the start of 2020. The earlier mentioned book by Nanjekye [9] about the compatibility during the transition from Python 2 to 3 highlights the major aspects that developers had to keep track of when transitioning to Python 3 and making their projects functional for Python 2 users. To summarize, the high profile major changes made by Python with its transition to version 3.x come down to syntax, the deletion or altering in behavior of some built-in functions and its method of encoding characters.

For instance the simplest syntax change made for Python 3 was the use of ”print”, which in Python 2 was a statement, while in Python 3 it was made into a function, and therefore uses parentheses:

```
Python 2:
print "This is a syntax example"

Python 3:
print ("This is a syntax example")
```

In terms of character encoding, Python 2 used the ASCII standard by default, while this was changed to UNICODE in Python 3. Nanjekye also highlights that this was beneficial since UNICODE supports more linguistic deiversity compared to ASCII.

An example of a change in built-in functions can be highlighted in the use of the ”range()” function. In python 2, this function included a standard ”range()” to generate a sequence of numbers temporarily stored in a list and an ”xrange()” variant which allows to create the sequence as a generator object. In python 3, the ”xrange()” function was removed and the standard function ”range()” is now the only one in use, and operates in the same way that ”xrange()” did in Python 2.

4.3 Python version release frequency

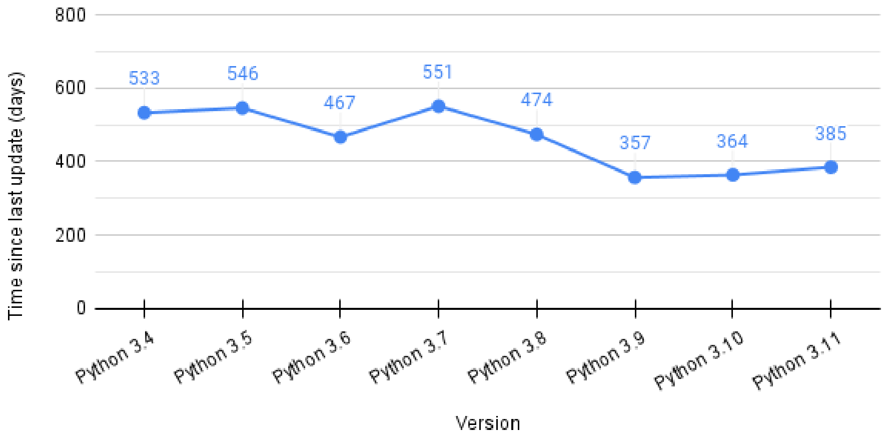


Fig. 2: Python update release frequency

In order to analyze the compatibility updates over-time for the chosen Frameworks in conjunction with different minor Python versions, its is valuable to first look at the release frequency. Figure 2 showcases the minor Python versions released over a period roughly ten years (late 2012 to Summer 2023) and the number of days in between the next version release. For example, Python 3.4 was released 533 days after Python 3.3 while version 3.5 was ultimately released 546 days after 3.4 and so forth, up until the most current Python release 3.11 which was released 385 days after its preceding minor version release. This data corresponds with the dates that were officially published by Python [6]. It can be observed that since the introduction of version 3.7 , The number of days it takes to release a new minor Python versions has been consistently shortened to be closer to a year (365 days) whereas before that release, the number of days were closer to a year and a half (547 days). The publicly shared Python Enhancement Proposals (PEP) index [19], serves as an official historical text record of version changes and release cycle details. The shortened time-span between releases was not something that was introduced out of the blue, but was communicated through a PEP post [20]. Though an exact reasoning for this release strategy change is not given, Python shows transparency when these changes are made, to facilitate that developers can read up on this if they wish to. In response, Python frameworks have to adapt in order for them to be compatible with these newly released Python versions, but this costs time. This will be explored in more detail in sections 4.4 to 4.6 .

4.4 Framework compatibility - Django

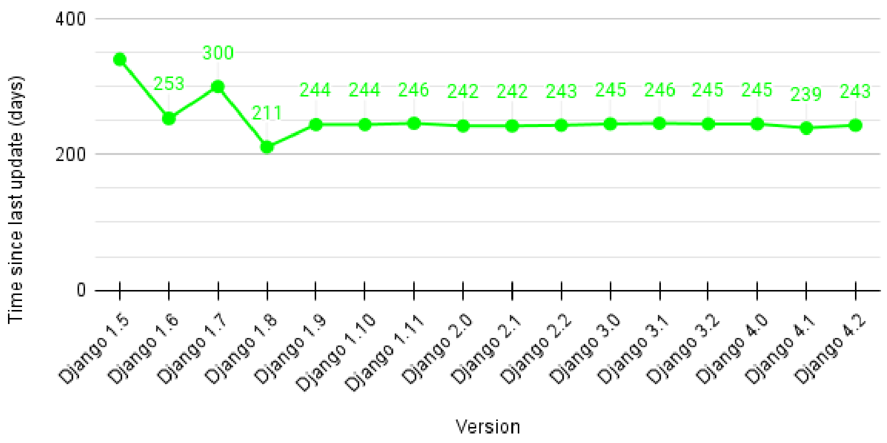


Fig. 3: Django update release frequency

Figure 3 showcases the different Django releases in the same manner as Figure 2, of which the dates were taken from the official Django release history documentation [21]. Over the course of Early 2013 to Summer of 2023, Django released 16 minor version updates. As stated in Django's release process documentation [22], from version 2.0 onward, the Semantic Versioning [23] nomenclature was adopted and modified. This Semantic versioning follows the same "Major.Minor.Micro" structure as used by Python. The modification that Django made was of particular importance for Long-Term Support version releases (LTS). Django clarifies that in order to better showcase compatibility for developers, the version release after an LTS release would be bumped to the next "dot zero" release. This is visible in the plotted data showcased in figure 3 as well, with the jumps from 2.2 to 3.0 and 3.2 to 4.0 . Thereby, versions 2.2 and 3.2 serve as LTS releases that maintain compatibility with versions 3.x and 4.x respectively. It can further be observed that, not unlike what was found in Figure 2, also Django eventually moved towards a roughly streamlined time gaps in between new minor version releases. In the case of Django, this starts with the release of version 1.9 , for which it is stated in [22] that a roughly 8 month time-frame (around 243 days) is used for feature updates. When investigating further into Django reports on their release strategy earlier in the framework's development process, it can be found in [24] that the current release process of roughly 8 months, used to be more flexible around the time of releases closer to Django versions 1.0 to 1.5. Django stated at the time that the release process was for around 9 months or more, depending on the quantity of work that was required for a new minor version release. This means that Django eventually became stricter on their release process and shifted to the 8 month time-frame. Further clarification on why the 8 month strategy in particular is being used by Django is not given. It is thereby also not clarified in any Django documentation if this is in any way related to the release strategy for new Python versions or not.

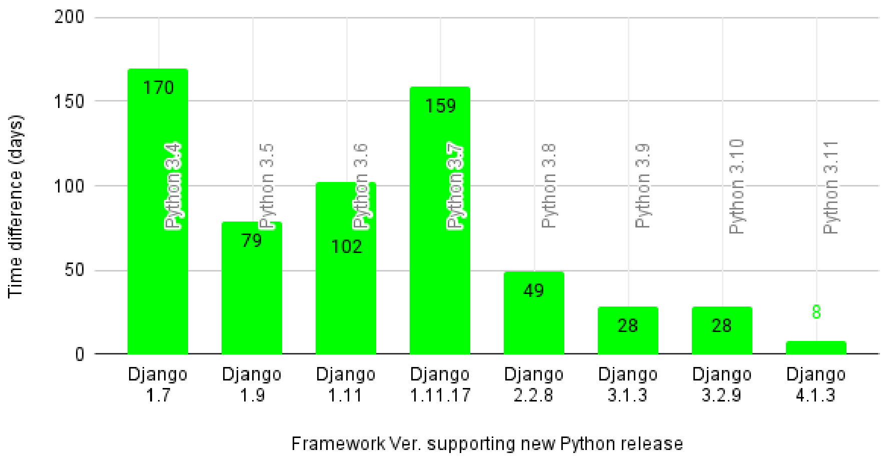


Fig. 4: Django-Python compatibility updates

From the information provided in the Django version release history documentation [21], a plot was also made (going over the same time-frame as Figure 3), displayed in figure 4, highlighting the different Django version that first introduced compatibility with a new minor Python release, as well as the number of days it took to push out the Django update. For example, Django 1.7 was the first Django release to support compatibility with Python 3.4 and the time difference in days between the release date for that Python version and that subsequent Django compatibility update release date was 170 days. The various Django versions displayed on the x-axis, highlight that the Python compatibility updates that are released by the Framework, do not always coincide with their 8-month minor version update schedule highlighted in Figure 3, with the exceptions of versions 1.7, 1.9 and 1.11 . Most Python compatibility updates from Django over the past 10 years, are thus released during patch updates such as 1.11.17 and 2.2.8 . When comparing these results to the overall Django version release strategy highlighted in Figure 3, it becomes clear that Django indeed does not specifically aim to push for for the Python compatibility updates to coincide with their own 8 month minor version release schedule.

4.5 Framework compatibility - Pyramid

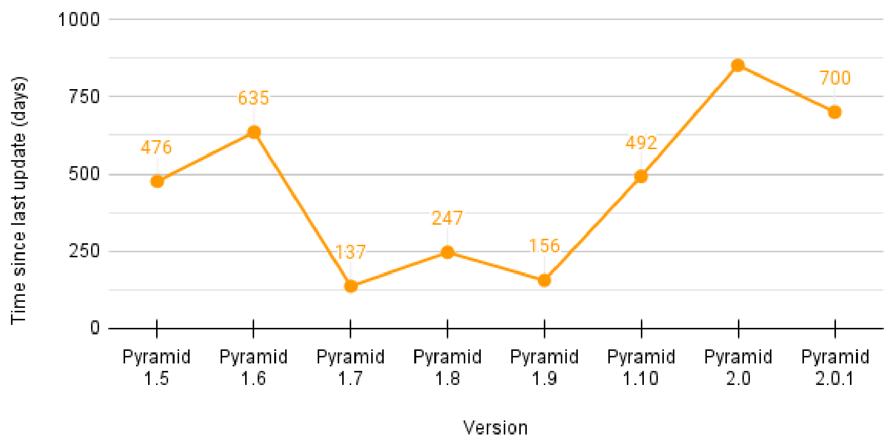


Fig. 5: Pyramid update release frequency

Figure 5 shows, in the same manner as Figures 2 and 3, the gap of time (in number of days) between minor Pyramid releases. This too, goes over the same period of time from 2013 to 2023 and was taken from the official version history documentation published by Pyramid [25]. A first observation is that over the analyzed time-period, Pyramid released a total of 8 minor releases. This is a substantially lower tally compared to Django which saw twice as many minor update releases within a similar time-frame. When comparing the 8 points in the figure 5, it can be observed that the line is far from flat, indicating that the time intervals between these minor update releases were not pushed in a particular predetermined and structural manner, in the way that Django and Python have done. For example, it took 635 days to transition from Pyramid 1.5 to 1.6, while the release of Pyramid 1.7 came substantially faster, with a time-gap of 137 days.

In a similar way to what was shown for Django in figure 4, Pyramid also does not push for Python compatibility updates to coincide with minor or major version releases for the framework. Across the analyzed time-period, the Python compatibility updates are similarly inconsistent, ranging from a fast 2 day time difference between the release of Python 3.6 and the compatibility update Pyramid 1.8a1, to a 482 day difference between the release date of Python 3.10 and Pyramid 2.0.1 . The substantial inconsistencies present in the last four bars in Figure 6 are not entirely unrelated however. It can be observed that Pyramid versions 2.0a0 and and 2.01 both served as Python compatibility updates for two different python versions at the same time. When referring back to the plot in Figure 2, it can for instance be observed that there were 385 days between the release of Python 3.10 and 3.11 . As a result of pushing for compatibility of the two Python versions in one Pyramid version release, it comes relatively shortly after the release of Python 3.11 and relatively long after the releases of 3.10, as the latter version was skipped in a potential earlier compatibility release. A final observation and comparison to Python and Django is the difference in versioning semantics. In contrast to the earlier mentioned Semantic Versioning [23] used by Python and Django, Pyramid opts for its own semantics. Particularly noteworthy with Pyramid's approach is the use for letters for their patch releases, such as 1.10a1 which is visible in figure 6 and can be seen used more extensively throughout the Pyramid version history documentation [25].

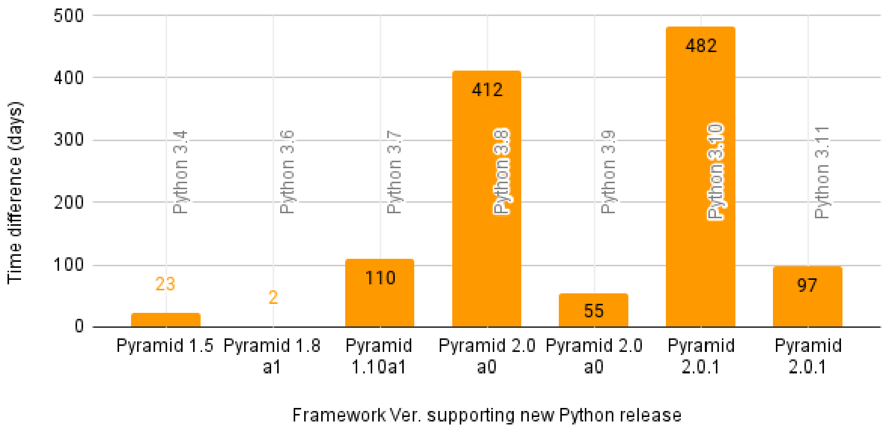


Fig. 6: Pyramid-Python compatibility updates

4.6 Framework compatibility - Flask

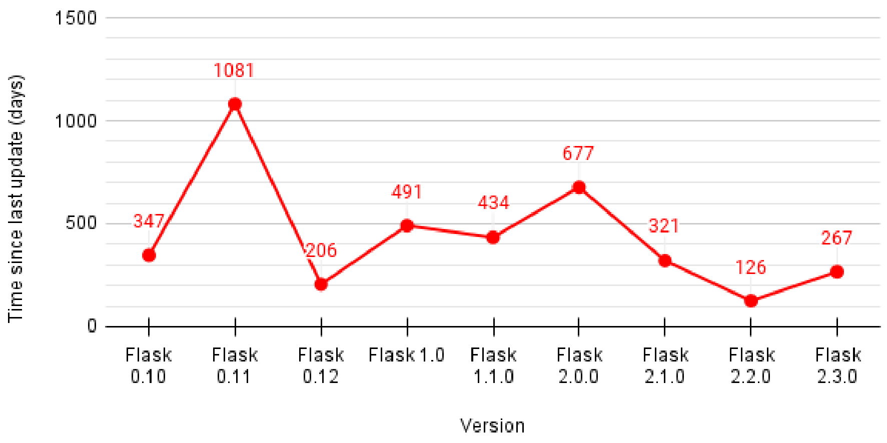


Fig. 7: Flask update release frequency

When it comes to the release frequency from Flask [26], it can be observed from the plotted time gaps between releases shown in figure 7 that there is no structured or predetermined release process for the framework, as the line is far from flat. When researching further documentation for Flask other than the version history, there was indeed no published material on release strategies for minor and or major updates for the framework like Django and Python for instance have been doing. By extension, there was therefore also no clear effort documented by Flask about potentially taking the standardized Python release strategy in mind when pushing for Flask updates.

The extremes in time-gaps shown in figure 7 range from 1081 days in between minor version releases for Flask 0.10 and 0.11, to just 126 days in between the releases for Flask 2.1.0 to 2.2.0 . This substantial gap of 1081 days is especially noteworthy. When looking at the Flask version releases around the time those two updates were released, in the change history documentation [26], there was only one small patch (Flask 0.10.1) released the day after 0.10 came out. After that, there were no new changes introduced to the framework until well over three years later.

When it comes to Flask and Python compatibility, a different approach had to be taken which resulted in a different bar plot in figure 8, compared to the Python compatibility plots in figures 6 and 4.

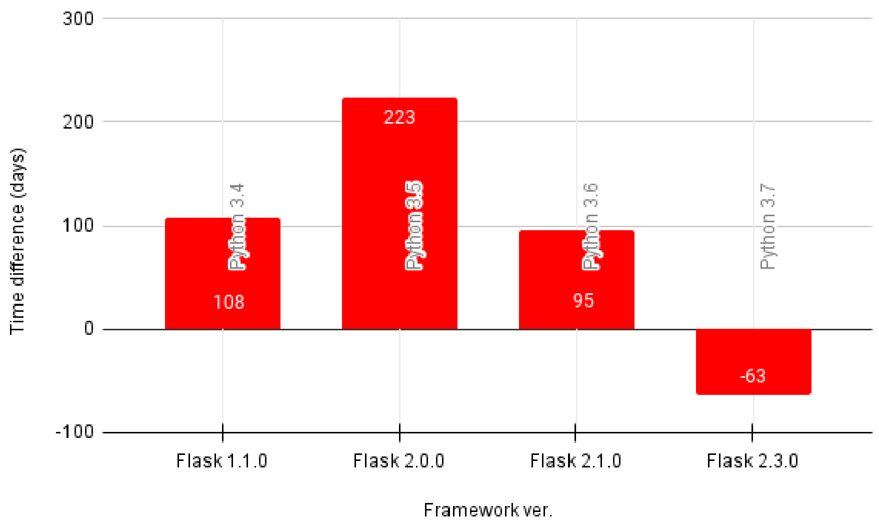


Fig. 8: Flask-Python end of compatibility updates

In contrast to the version history documentation published by Django and Pyramid, Flask does not actually specify what version releases introduce compatibility with new Python versions. The documentation does however specify for certain releases when it drops support for a specific Python release. In terms of actual compatibility with different Python versions, the instruction that Flask gives on their download page [27] highlight that it is recommended to use the latest Python version and that, as of June 2023, Python 3.8 and higher are compatible. This indicates that at least the latest four different minor Python releases are compatible with the framework at the same time, excluding pre-releases.

Because of this different approach by Flask, figure 8 was plotted using the aforementioned data and was compared to the End-of-Life (EOL) dates from different Python versions as opposed to the release dates of these versions. Just like the previous figures, the time span from early 2013 to summer 2023 was used. To give an example, the Flask documentation highlighted that Flask 1.1.0 would drop support for Python 3.4 . The first bar in the plot therefore showcases the time-gap in days between the EOL date for Python 3.4 and the aforementioned Flask release that would drop support for this Python version. In the case of this example, Flask 1.1.0 was released 108 days after the EOL date of Python 3.4. Particularly noteworthy is the bar for Flask version 2.3.0, which showcases a time gap of -63 days. This indicates that the version of Flask dropped support for Python 3.7 before it reached its EOL date. No further indication in the change log for Flask 2.3.0 was given for the decision to drop the support for Python 3.7 that early.

4.7 Practical examples: Outdated dependencies, version requirements and vulnerabilities

Practical examples can be highlighted by analyzing projects that use Python and one or more of the frameworks relevant to this research. One example is an open-source project shared on GitHub called Zappa, which is a tool that facilitates server-less web hosting of Python applications [41]. Zappa added support for Python 3.10, through an update released on May 16, 2023 [42]. When referring back to figure 1, the initial release date for Python 3.10 was in October 2021. It thereby took Zappa about 19 months to add support for that Python version. It is to be noted that Zappa up until that point in its version history did support Python version 3.8 and 3.9 which are still fully supported for security updates by Python up until 2024 and 2025 respectively.

In the same Zappa update, a fix was highlighted for a problem concerning a Django 4.2 support error. In this scenario shared on April 16 2023 [43], a developer was using Zappa together with his Django web application. The person in question recently pushed for a Django update to version 4.2 after GitHub's dependabot highlighted security issues. It took 11 days for a Zappa patch to be released regarding this issue [44], and another month for this patch to be highlighted in the version history documentation previously mentioned, published on May 16, 2023.

In the case of Zappa in general, a pipfile is used for the project that contains the recommended dependency version requirements [45]. This therefore facilitates users to update to all the right dependency versions using the pip installation and update tool. This file was updated just once in 2023, namely in May. Before that, there were a total of three updates made to the requirements in 2022, once in July and two times in October.

There can also be looked at an example of a vulnerability in Python that required a necessary update to fix and how long it took to resolve such a security issue. The example highlighted in the National Vulnerability Database (NVD) under CVE-2023-24329 [46], showcases an issue that struck all Python versions of 3.11.3 and earlier. This particular issue was found in the "urllib.parse" Python function that allowed attackers to bypass blocklisting methods, thereby injecting potentially malicious changes to URLs. As can be seen in the blog post from the person who originally found this issue [47], it was first discovered on 07/20/2022, after submitting the issue to the Python security team, the issue was published to the public github page, where people could test different solutions [48]. After the solutions were officially approved by Python, the security update was included in the release of Python 3.11.4 [49], which was released on 6/6/23, thus nearly one year after the vulnerability was first discovered.

5 RQ2 - Version update behavior and stances of Python and Frameworks: Survey results

5.1 Survey introduction

In order to answer RQ2, a survey was conducted that captured the stances and behaviors that the responding developers have and show on multiple aspects related to the topic of this Essay. The respondents were gathered over the course of two separate rounds. The first round resulted in the bulk of the respondents who were approached by sharing the survey on various general Python forums, which resulted in 50 respondents. The second round found an extra 11 respondents by sharing an online article containing the interim results of the first round, via LinkedIn. The contents of the survey can be divided in multiple facets.

Firstly, respondents were asked about their behavior regarding keeping up with new version releases for Python and where possible, also their most used framework. More specifically, their behavior here refers to manually keeping track of new Python and framework version releases through looking up change logs and other documentation related to version updates. Secondly, respondents were asked about the actual frequency with which they generally update their Python and framework versions. Their given options ranged from "at least once a year" to "every 3 years of more". Thirdly, there was specifically asked about which frameworks were predominantly used by the respondents. This resulted in the possibility to not just gather result data regarding the more general use of frameworks such as with the aforementioned two facets of the survey, but also to highlight specific frameworks such as Django and Flask. Fourthly, the survey also consisted of multiple open questions related to the predominant reasons for why the respondents decide to update their Python and/or framework versions. Additionally they were given the option to elaborate on a specific question regarding the major transitions from Python 2 to Python 3. Lastly, specific to the second round of the survey, the decision was made to also add more general questions that would result in opinions regarding the state of Python and of the respondent's most used framework. This was predominantly done to make potentially meaningful comparisons with existing survey research from both within the Python domain, such as JetBrain's annual Python developer's survey [28,29] and outside of the Python domain, such as the State of JavaScript survey [30].

The remaining subsections of section 5 will go over the results for these facets of the survey individually. It is also important to note that the data set of the survey results is linked in the Appendix section of this essay. Various results that will be discussed in these subsections use proportions and percentages for the plots. This decision was made due to the fact that not every single one of the 61 respondents also had experience with frameworks, and therefore were able to skip over the Framework related questions in the survey. See the percentages of respondents who had experience with a framework in figure 9. By using proportions and percentages, meaningful comparisons between various questions related to Python and Frameworks could be made without the impact of the discrepancies in total number of responses for between the Python- and Framework-related questions.

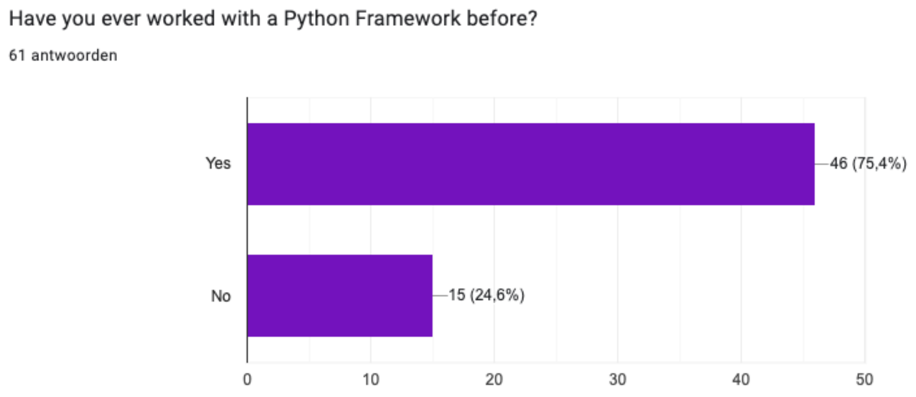


Fig. 9: Nr. of respondents with framework experience

5.2 Staying up-to-date with new Python and framework versions

As mentioned in section 5.1, the first facet of the survey asked the respondents how likely they are to actively research the specific changes that are made right after a new version of Python or their preferred framework is released.

```
Question Python:
Python annually pushes for a new version release (i.e. Python 3.10
to version 3.11). How likely are you to keep up with the annual
releases and changes made for these newly released Python versions?

Question Framework:
If you work (or have worked) with Python frameworks. How likely
are/were you to keep up the specific changes made in a new version
of the framework(s) right after its release?
```

The results of these two questions can be seen in figure 10. The proportions of responses for the "definitely not" and "(probably) no" options are shown on the negative (left) side of the x-axis, while the proportions of responses for "(probably) yes" and "definitely yes' are shown on the positive (right) side. Figure 9 shows, that roughly 3.3% voted for "definitely not", 8.2% for "(probably) no", 44.2% for "(probably) yes" and 21.3% for "definitely yes", in the case of Python. For frameworks, the results were 2.2% , 43.5% , 30.4% and 8.7% respectively. Additionally, there was also an intermediate option, not displayed in Figure 9, for "might or might not" which got 22.9% for Python and 15.2% for frameworks.

These results highlight, in the case of python, that a majority of roughly 65.5% (44.2 + 21.3) say that they do keep up with changes made to Python right after a new version release. On the other side, with frameworks, the majority fell on the negative side with roughly 45.7% (2.2 + 43.5). Noteworthy is also that, with 21.3%, respondents showed a pronounced likelihood of researching a new Python release by voting for "definitely yes". This is a portion more than twice as big as the same option for frameworks, with its 8.7%. Furthermore, the big portion of "(probably) no" voters of 43.5% also sticks out, illustrating that the respondents to the survey are less likely to research changes made in new framework version releases. Though, at the same time, it must also be noted that the 45.7% majority split to the negative side for Frameworks, is closer to an equal split, with the positive side adding up to a portion of 39.1% (30.4 + 8.7). In contrast, the split for Python was starker at 11.5% negative to 65.5% positive. It is to be noted that the remaining missing percentages for both Python and Frameworks that would add everything up to 100 percent respectively, are the remaining respondents who voted to be neutral on the matter.

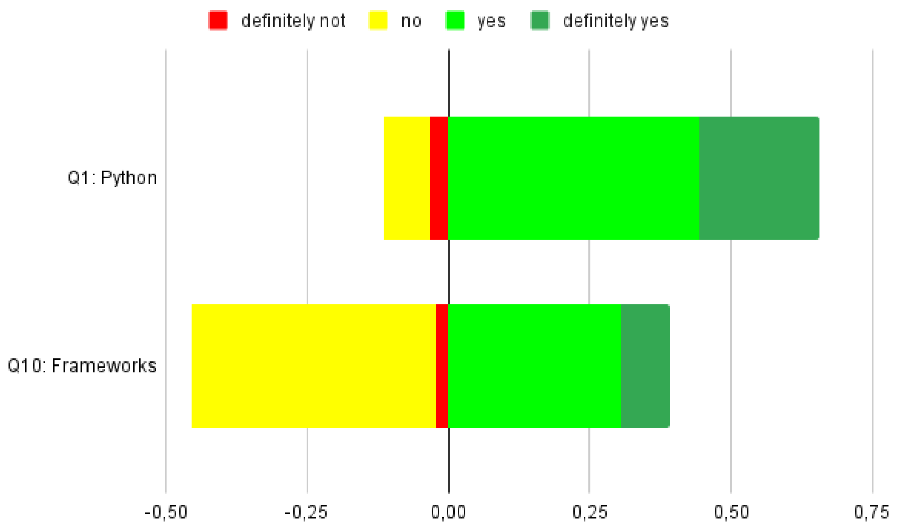


Fig. 10: Staying up-to-date with new Python and framework version

5.3 Upgrade frequency Python and frameworks

The second facet highlighted in section 5.1 is the update frequency of the respondents. two separate general questions were asked, once again, for both Python and Frameworks.

```
Question Python:
Released Python versions tend to be supported for about 5 years
before they reach "end-of-life" status and no more updates can
be pushed for them.
How long do you tend to stick to working in one particular version
of Python before you consider updating to a new version? For instance,
your set version as your Python interpreter in your Integrated
Development Environment (IDE)?

Question Framework:
How long do you tend to stick to one version of your used framework(s)
before considering updating to a newer released version?
```

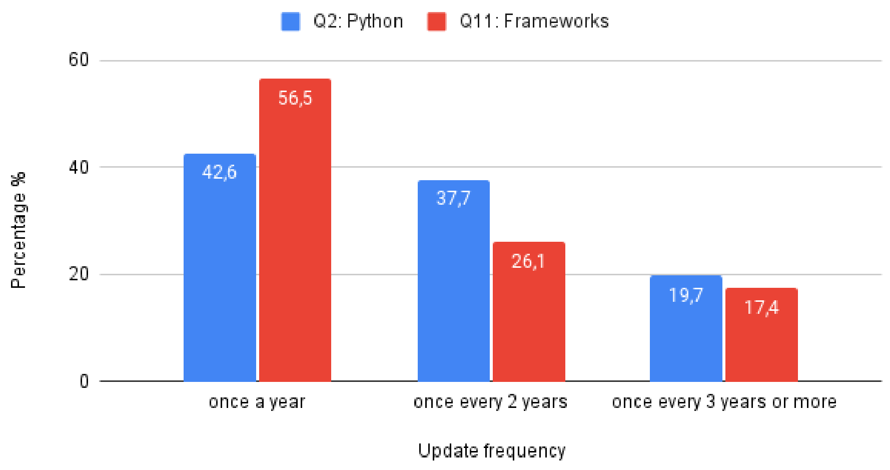
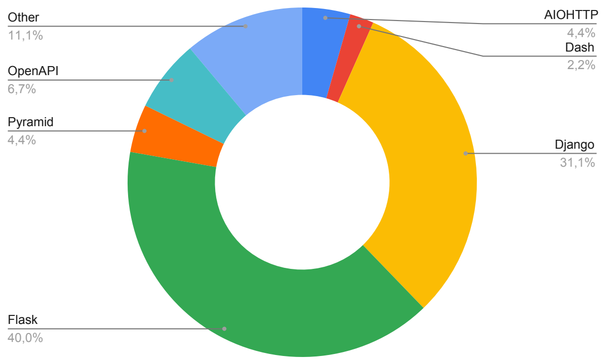


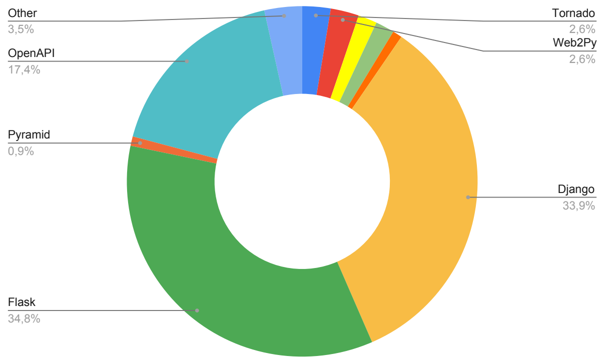
Fig. 11: Update frequency responses for Python and Frameworks in percentages

The results are displayed in Figure 11. From observation, it can be noted for both Python and frameworks, are most frequently updated at least once a year. In the case of frameworks specifically, this is more than half of the respondents with 56.5%. The options for "once every 2 years" and "once every 3 years or more" were subsequently chosen increasingly less often by the respondents. Though, it can be said that the proportions of Python responses for "once a year" and "once every two years" are relatively close. When looking back at the update release frequency from Python, shown earlier in figure 2, one observation was that Python pushes for minor version updates within a year or a year-and-a-half of the previous preceding release. With the fact that respondents to the survey say that they predominantly update to a new Python version within a year or 2 years, it can be argued that this is connected to the aforementioned update strategy by Python to get a new minor release out within a year-and-a-half. The results in figure 11 can further be compared through an existing study such as [31]. This empirical study from 2020 included an analysis of the update delay of libraries used in Java projects. This part of the study showed that the vast majority of their analyzed java projects updated to a newer library release version within half a year. It further highlights that, not unlike the results in Figure 11, the higher the number of days of update delay, the lower the number of project instances became, resulting in a similar down-going trend.

5.4 Framework-specific responses



(a) Figure 12A: Survey responses for this research



(b) Fig 12B: JetBrains Developer ecosystem survey 22 [29]

Figure 12A shows the various python frameworks that the survey respondents had experience with. Flask and Django were by far given as the most popular frameworks, with 40% and 31.1% respectively. Out of the 20 total options given for different frameworks, only 6 unique ones were being used within the group of survey respondents. Aside from the clear outliers with Flask and Django, Open API was given as the runner-up for most used framework, with 6.7%.

As was already briefly mentioned in section 5.1, The survey results for this research can draw parallels to the JetBrains Python developer's and ecosystem survey results that are published annually [28,29] (shown in figure 12B). To start, both in 2021 and 2022, Flask Django and OpenAPI were given as the top-3 most used frameworks. This top 3 is also reflected in the survey results for this research. An argument that JetBrains makes in the results for their 2022 survey [29], is that OpenAPI, as a newer more modern web development framework, has been on the rise for a number of years now. As a competitor to Flask and Django, those two have been on a slow decline in recent years, which might thus be connected to the rise in popularity of FastAPI. It also becomes clear from the JetBrain survey results, that for multiple years in a row, the most popular use of Python and its frameworks has been web development. Out of all the 6 unique entries for different frameworks shown in figure 12A, five of them are also web development frameworks, with one response for Dash being the only exception. Beyond the JetBrains surveys, popular online community platform Stack Overflow also conducts its annual developer survey. In its 2022 survey results [34], only three Python frameworks made it into the top-list under the "Frameworks and Technology" category, which was gathered from all programming languages tagged on their website. Just as the results shown in figures 12A and B, those three were also shown to be Django, Flask and FastAPI.

There can also be looked at the distribution of the results to the used frameworks within the pool of survey respondents. Through calculating the Gini coefficient [32], more can be shown about the distribution (in)equality of the responded used frameworks. The Gini coefficient is most commonly used to calculate the (in)equality of income within a population, but can also be used for other arrays of data, as for instance shown in [33]. The Gini coefficient was calculated by taking the SUMPRODUCT of the calculated array of fractions in column 3 of the table. As explained in [32], a Gini coefficient can be between 0 (maximum equality) and 1 (maximum inequality). As shown in the table, the Gini coefficient for the responded used frameworks resulted in 0.28 .

Framework	**Nr. of respondents**	**Fraction of total**
AIOHTTP	2	≈ 0.04
Dash	1	≈ 0.02
Django	14	≈ 0.31
Flask	18	≈ 0.40
Pyramid	2	≈ 0.04
OpenAI	3	≈ 0.07
Other	5	≈ 0.11
	Total = 45	G ≈ 0.28

Figure 13 showcases the fractions of respondents who voted for whether or not they keep up with changes made in new framework releases. Figure 14 highlights how often the respondents update their used framework. It can further be observed that, as a result of many survey respondents using very few distinct frameworks, most of these distinct entries simply showcase singular peaks for one of the three update frequency options. For instance, AIOHTTP and Dash both were given by 1 unique survey respondent each.

When it comes to comparing Django and Flask, which were chosen the most often by the respondents, some things are also worth noting. In figure 13, Flask is shown to be largely on the negative left side, meaning that the majority of the Flask users within the survey respondents do not tend to keep up with the changes made when a new Flask version is released. On the other hand, the majority of Django users tend to do keep up with new changes made in new releases. From the results highlighted in sections 4.4 and 4.6, it became clear that Django has been implementing a distinctly planned out release schedule and publishes change logs for each of those new releases. Flask on the other hand is shown to be less transparent, especially in regards to Python compatibility, where they simply recommend users to use the most recent three Python releases. Perhaps these different aspects between Django and Flask also contribute to this behavior shown in Figure 13. In terms of update frequency, its further shows in figure 14 that most respondents tend to update their Flask or Django at least once a year, with a declining trend for ”once every 2 years” and ”once every 3 years or more”.

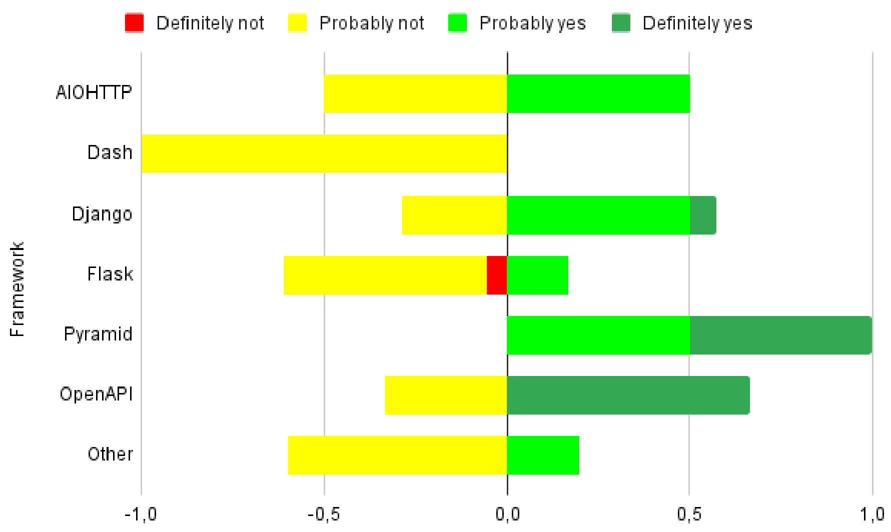


Fig. 13: Keeping up with new framework releases

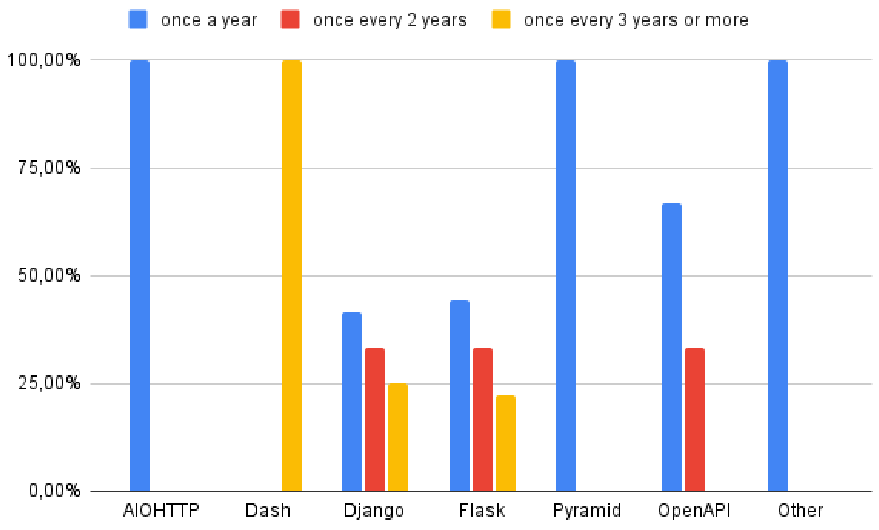


Fig. 14: Update frequency behavior per framework

5.5 Comparisons to State of JS

It was already stated in section 5.1, that the second round of this survey research also contained questions posed in a similar way to the State of JS survey [30]. The questions and results showcased in Figures 15 were posed in an identical way to that of the StateofJS survey, and respondents were given the same 5 options ranging from "disagree strongly" to "agree strongly". In the case of the questions for this survey research, two specific questions were posed.

```
Q1: In general,  do you think that Python as a programming
language is moving in the right direction?
Q2: Do you think that building applications in Python is overly
complex right now?
```

The results in figure 15 showcase that there were no respondents that disagreed with the statement that Python as a programming language is moving in the right direction (Q1). Subsequently, respondents were either neutral on the matter or agreed. When compared to the StateofJS results, it can be noted that these results were largely reflected in a similar manner. For instance in the results for 2022 [30], the majority of the votes went to "agree" with 57%, while most remaining votes went for either "agree strongly" or neutral with around 18% each. The StateofJS survey results do however also show a small percentage of respondents (3.8%) who disagree with the statement that JavaScript is headed in the right direction. The results shown for Q2 illustrate that survey respondents either disagree or are neutral on the subject of building applications in Python being overly complex. One survey entry did however chose to agree on the sentiment that it is overly complex. The results for the StateofJS survey showcase a striking, near equal split on "agree" and "disagree" on this topic. At least from the pool of responses gathered for the survey of this thesis research, that kind of dichotomy in stances is not present for Python.

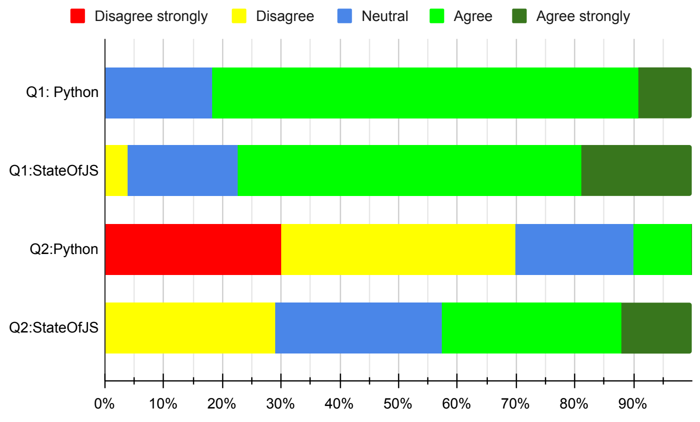


Fig. 15: StateOfJS inspired questions [30]

6 Discussion

6.1 Research complications and limitations

In terms of complications and limitations of this research, a number of aspects can be highlighted.

Firstly, the results shown in figure 8, as also described in its section, shows different content compared to that of figures 4 and 6 because Flask did not specifically highlight any specific release dates for Python compatibility updates, but only specified when they dropped specific Python version support. As a result, more direct comparisons with figures 4 and 6 could therefore not be made. When it comes to the survey, the number of respondents ended up at a total of 61 developers over the course of two rounds of survey sharing. The decision was made for the second round, to also add more questions that more specifically tie to previous survey studies such as the Python Developer's Survey and the JetBrains Java Script survey research. Subsequently only the portion of respondents of the second round were able to answer those questions, which ended up at 11 responses of the total of 61.

Efforts were made for the first round of the survey to more specifically target users of distinct different possible frameworks, but compared to the general Python forums where the survey could be shared without any issues, this proved to not be possible for more specific forums such as for Django. This was mainly because those forums were specifically focused on programming-specific problem sharing and solving within that community. Consequently, efforts to share the survey and introduce this research were taken off those forums.

6.2 Further research recommendations

There are a number of directions in which further research on the topic of this essay can go in. Firstly, a similar research can be repeated for different programming languages. If these languages do not have frameworks available, there can also be looked at dedicated libraries and packages when it comes to compatibility updates in response to major or minor language updates. When it comes to the release strategies in particular, one can choose to focus on the level of transparency that these different languages show. Especially in terms of elaborate documentation for a set update release strategy that users can rely on and proper elaboration when changes to these strategies are made. On top of that, compatibility when it comes to frameworks, libraries and package updates can be analyzed in a similar manner when it comes to transparency.

One can also choose to distinctly focus on one Python framework, where a detailed research can be conducted on the exact contents of the change logs of different version releases. Thereby, update release cycles and stances from users can be evaluated on a more programming-technical level.

Finally, a repeat study of the one described in this essay can also be conducted, with lessons learned, as described in section 6.1, taken into account.

7 Conclusion

7.1 Research question 1

For research question 1, an analysis was done of the update release cycles and strategies from Python, Django, Flask and Pyramid. From the figures showcasing the time gaps between different version releases, it can be concluded that Python and Django have been conducting specific release strategies resulting in similar time gaps between releases, which results in relatively flat lines in figures 2 and 3. On top of this, they showed clear transparency into the specifics of these strategies through documentation. In contrast, the results for the release cycles/strategies for Pyramid and Flask showed far from flat trend lines in figures 5 and 7. From the documentation and change log evaluation, it became clear that the two frameworks do not conduct any kind of set release strategy in the same way that those behind Python and Django do, thereby resulting in these, at times extreme and inconsistent, time gaps between minor version releases. Research question 1 was further answered through analyzing the speed with which the chosen three frameworks pushed for compatibility updates in response to new Python version releases. Django was found to become faster over the years, with their compatibility updates, as highlighted in figure 4. A contributing factor to this was found to be their shift to push for these compatibility updates during patch releases instead of at end of minor version release cycles, which last up to 8 months. Pyramid showed particularly stark ups and downs in its compatibility release speed (figure 4). It was found that this is the result of an apparent strategy to push for compatibility updates for two Python versions at once. As a result, the compatibility updates come relatively soon after the most recent Python release, while the Python release prior to the most recent one, is shown to have been incompatible for a much longer time, resulting in peaks as high as 482 days. Finally, there was the separate case of Flask, which in contrast to Django and Pyramid, only highlighted in their documentation and change logs when a new version release would drop support for a certain Python version. From figure 8, it can be concluded that Flask does not have a specific strategy in place for how long they take to drop compatibility from a End-Of-Life Python version. Based on these findings, it can also be concluded that Clear strategies and transparency as shown by Python and Django in particular would therefore be a recommended style of approach.

7.2 Research question 2

The second research question was answered through conducting a survey about Python developers and their version update behaviors and stances for Python itself and available frameworks. The survey got a total of 61 responses over the course of two rounds. The first round added up to 50 responses and the second round with an additional 11. Of the total of 61 respondents, 46 of them had shown to have experience with Python frameworks. In terms of keeping up-to-date on the changes made for new Python and framework releases (figure 10), multiple conclusions can be drawn. Firstly, the results show that for Python, the majority of 65.5% say they do keep up with these changes. For the frameworks, the responses were in a slight majority to the negative side, but was ultimately closer to an equal split, thereby showing that the respondents do not show an overwhelmingly dominating update behavior for frameworks, within the pool of survey responses. For both the frameworks (46.2%) and Python itself (56.5%), those majority percentages of respondents choose to update their Python version at least once a year. Options for once every two years or once every 3 years or more were shown to be increasingly less common amongst the respondents. When it comes to the framework-specific responses, Flask (40%) and Django (31.1%) were overwhelmingly chosen to be the most commonly used Python frameworks amongst the survey respondents, while a relatively new framework such as OpenAI (6.7%) came out to be the runner-up. It can be concluded that these results were in line with other survey studies conducted on Python frameworks such as [28,29]. In terms of the split of different Python frameworks shown in figure 12, the Gini coefficient was calculated and resulted in a score of roughly 0.28, as shown in the table in section 5.4. With the gini coefficient being between 0 (complete equality in distribution) and 1 (complete inequality), a score of 0.28 gives further statistical indication that the distribution of the different used frameworks by the survey respondents does not lean towards complete inequality. While, Flask and Django make up for approximately 71% of the responses, the other remaining frameworks are still divided over a remaining 29%. When looking at how likely the respondents are to keep up with new changes made for their chosen Frameworks, a number of thing can be concluded. Because many votes went to few frameworks (Django and Flask predominantly), the results showcased a wider range of chosen options. the majority of the Django users within the survey pool of respondents, chose that they do mostly keep up with the changes made between new version releases, while for Flask, users were shown to be less observant of changes. There is to be noted with this discrepancy, that it was found in section 4.4 that Django maintains is strict release cycle where updates are therefore pushed around the same time each year, which allows Django users to prepare to update and keep track of changes around the same time each year, whereas Flask users do not have such consistency to rely on with new version releases. The update frequency from the respondents showcased with Django and Flask especially, a largely consistent result with the results of Python and Frameworks in general, with most people updating once a year, with increasingly less people opting for 2 or 3 years (or more). The results for the questions related to the state of Python were analyzed and compared to the StateOfJS survey results. From this, it can be concluded that, both the results for Python and Java Script showed that the majority of respondents are happy with the general direction of the two languages. When it came to the question whether or not developers find it difficult to make applications in their respective languages, the StateOfJS results showed a striking divide, while the majority of Python respondents showed to be happy or neutral with the application development difficulty for their used language.

References

1. PYPL PopularitY of Programming Language, https://pypl.github.io/PYPL.html. Last accessed June 2023.
2. TIOBE Index for January 2023, https://www.tiobe.com/tiobe-index/. Last accessed January 2023.
3. Alfadel, M., Costa, D. E., Shihab, E. (2021, March). Empirical analysis of security vulnerabilities in python packages. In 2021 IEEE international conference on software analysis, Evolution and Reengineering (SANER) (pp. 446-457). IEEE. https://doi.org/10.1109/SANER50967.2021.00048
4. Sun, S., Wang, S., Wang, X., Xing, Y., Zhang, E., Sun, K. (2023). Exploring Security Commits in Python. ArXiv preprint arXiv:2307.11853. https://doi.org/ https://doi.org/10.48550/arXiv.2307.11853
5. Upgrading a Django 1.8 site to Python 3, https://petegraham.co.uk/django-and-python-3/ Last accessed June 2023.
6. Status of Python versions, https://devguide.python.org/versions/ Last accessed June 2023.
7. Django supported versions, https://www.djangoproject.com/download/ Last Accessed June 2023.
8. Flask Changes, https://flask.palletsprojects.com/en/2.3.x/changes/ Last accessed June 2023.
9. Nanjekye, J.: Python 2 and 3 Compatibility. Apress (2017). https://doi.org/10.1007/978-1-4842-2955-2
10. Malloy, B. A., Power, J. F. (2017). Quantifying the transition from Python 2 to 3: An empirical study of Python applications. In 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (pp. 314-323). IEEE. https://doi.org/10.1109/ESEM.2017.45
11. Aggarwal, K., Salameh, M., Hindle, A. (2015). Using machine translation for converting python 2 to python 3 code (No. e1817). PeerJ PrePrints. https://doi.org/10.7287/peerj.preprints.1459v1
12. Porting python 2 code to python 3, https://docs.python.org/3/howto/pyporting.html Last accessed June 2023.
13. Saabith, A. S., Fareez, M. M. M., Vinothraj, T. (2019). Python current trend applications-an overview. International Journal of Advance Engineering and Research Development, 6(10).
14. Ghimire, D. (2020). Comparative study on Python web frameworks: Flask and Django.
15. Zhang, Z., Zhu, H., Wen, M., Tao, Y., Liu, Y., Xiong, Y. (2020). How do Python framework APIs evolve? an exploratory study. In 2020 ieee 27th international conference on software analysis, evolution and reengineering (saner) (pp. 81-92). IEEE.
16. Django. The web framework for perfectionists with deadlines, https://www.djangoproject.com/ Last accessed June 2023
17. Pyramid. Documentation, https://trypyramid.com/documentation.html Last accessed June 2023
18. Python developers guide, Development Cycle. https://devguide.python.org/developer-workflow/development-cycle/ Last Accessed June 2023.
19. Python, Python Enhancement Proposals index. https://peps.python.org/pep-0000/ Last Accessed June 2023.
20. Python, PEP 602 Annual release cycle for Python. https://peps.python.org/pep-0602/ Last Accessed June 2023.

21. Django, Release notes. https://docs.djangoproject.com/en/4.2/releases/ Last Accessed June 2023.
22. Django, Django's release process. https://docs.djangoproject.com/en/dev/internals/release-process/ Last Accessed June 2023.
23. Semver, Semantic Versioning. https://semver.org/ Last Accessed June 2023.
24. Django Read the Docs, Django's release process. https://django.readthedocs.io/en/1.4.X/internals/release-process.html Last Accessed June 2023.
25. Pylons Project, Pyramid change history. https://docs.pylonsproject.org/projects/pyramid/en/latest/changes.html Last Accessed June 2023.
26. Pallets Projects, Flask changes. https://flask.palletsprojects.com/en/2.3.x/changes/ Last Accessed June 2023.
27. Pallets Projects, Flask installation. https://flask.palletsprojects.com/en/2.3.x/installation/ Last Accessed June 2023.
28. Jet Brains, Python Developers Survey 2021 Results. https://lp.jetbrains.com/python-developers-survey-2021/ Last Accessed June 2023.
29. Jet Brains, Python Developers Survey 2022 Results. https://www.jetbrains.com/lp/devecosystem-2022/python/ Last Accessed June 2023.
30. State of JS 2022, Opinions. https://2022.stateofjs.com/en-US/opinions/ Last Accessed June 2023.
31. Wang, Y., Chen, B., Huang, K., Shi, B., Xu, C., Peng, X., ... Liu, Y. (2020, September). An empirical study of usages, updates and risks of third-party libraries in java projects. In 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 35-45). IEEE. https://doi.org/10.1109/ICSME46990.2020.00014
32. Milanovic, B. (1997). A simple way to calculate the Gini coefficient, and some implications. Economics Letters, 56(1), 45-49. https://doi.org/https://doi.org/10.1016/S0165-1765(97)00101-8
33. Sadras, V., Bongiovanni, R. (2004). Use of Lorenz curves and Gini coefficients to assess yield inequality within paddocks. Field Crops Research, 90(2-3), 303-310. https://doi.org/https://doi.org/10.1016/j.fcr.2004.04.00
34. Stack Overflow, 2022 developer survey https://survey.stackoverflow.co/2022/#section-most-popular-technologies-web-frameworks-and-technologies Last accessed July 2023
35. Voigt, P., Von dem Bussche, A. (2017). The eu general data protection regulation (gdpr). A Practical Guide, 1st Ed., Cham: Springer International Publishing, 10(3152676), 10-5555.
36. OWASP Application Security Verification Standard 4.0.3 (2021). V14.2 Dependency (page 62).
37. OWASP top 10:2021 (2021). A06 Vulnerable and Outdated Components. https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/
38. PIP documentation V23.2.1. Dependency Resolution. https://pip.pypa.io/en/stable/topics/dependency-resolution/ Last accessed August 2023.
39. Github. Dependabot-core. https://github.com/dependabot/dependabot-core Last accessed August 2023.
40. Rich Jones (2023) Zappa, serverless Python. Github. https://github.com/zappa/Zappa
41. Rich Jones (2023) Zappa, serverless Python. Github. https://github.com/zappa/Zappa Last accessed August 2023
42. Rich Jones (2023) Zappa, pull 1247. Github. https://github.com/zappa/Zappa/pull/1246 Last accessed August 2023

43. Rich Jones (2023) Zappa, issue 1230. Github. hhttps://github.com/zappa/Zappa/issues/1230 Last accessed August 2023
44. Rich Jones (2023) Zappa, commit f56c86c. Github. https://github.com/dennybiasiolli/Zappa/commit/f56c86c4e24b59fcb52b788a19af3a4faca8dac8 Last accessed August 2023
45. Rich Jones (2023) Zappa, Pipfile commits. Github. https://github.com/zappa/Zappa/commits/master/Pipfile Last accessed August 2023
46. National Vulnerability Database (2023). CVE-2023-24329 Detail. https://nvd.nist.gov/vuln/detail/CVE-2023-24329 Last accessed August 2023
47. Yebo Cao (2023). Python url parse problem. PointerNull. https://pointernull.com/security/python-url-parse-problem.html Last accessed August 2023
48. Python (2023). Cpython pull 99421. Github. https://github.com/python/cpython/pull/99421 Last accessed August 2023
49. Python (2023). Changelog Python 3.11.4 Final. https://docs.python.org/release/3.11.4/whatsnew/changelog.html#security Last accessed August 2023

8 Appendix

8.1 A

Full spreadsheet with anonymous survey responses and tables used for the figures throughout the research are found here: https://docs.google.com/spreadsheets/d/1tw4Q30L0SPXix6V5hFMyn9LH64tQvtp4zs3NwsJve1A/edit?usp=sharing

8.2 B

The full overview of survey questions can be found here: https://forms.gle/qstVaBPNt8RDeiz39