STATIC TYPING IN PYTHON

Lappeenranta–Lahti University of Technology LUT

Bachelor’s Programme in Software and Systems Engineering, Bachelor's thesis

2025

Vasilii Lupuliak

Examiner: Doctor Erno Vanhala

Abstract

Lappeenranta–Lahti University of Technology LUT
LUT School of Engineering Sciences
Software Engineering
In co-operation with partner university:
HEBUT University of Technology
Vasilii Lupuliak

Static typing in Python

Bachelor's thesis
2025
34 pages, 2 figures, 1 appendix
Examiner: Doctor Erno Vanhala

Keywords: Python, static typing, dynamic typing, type checking.

Python is well-known as a dynamically typed language, meaning that type checks are moved into runtime and type annotations in the code are not required. However, it can still be statically typed with the help of type hints and external type checking tools. This research explores this practice in-depth, looking at the benefits it offers, and studies several documented cases of applying static typing in large, commercial Python projects.

Existing academic research in this area demonstrates the impact of static typing on software quality by performing various experiments and empirical studies. Overall, the use of type hints and static type checkers is proven to prevent a large portion of type-related errors.

Reports from various software companies highlight the importance of static types in large Python projects, emphasizing their positive effect on code maintainability, readability and type safety. Static typing especially enhances collaboration between developers, clarifying the interfaces via which various modules of the software interact with each other.

Abbreviations

Abbreviations

IDE	Integrated Development Environment
PEP	Python Enhancement Proposal
VS	Visual Studio

Table of contents

1 Introduction 6
1.1 Research questions 7
1.2 Structure of the thesis 7
2 Background 9
2.1 Static typing 9
2.1.1 Advantages of static typing 9
2.1.2 Disadvantages of static typing 10
2.2 Dynamic typing 11
2.3 Gradual typing 11
2.4 Python type system 11
2.5 Evolution of Python 12
3 Related work 13
3.1 Dynamic typing 13
3.2 Existing tools for static typing in Python 14
3.2.1 MyPy 14
3.2.2 PyType 15
3.2.3 Pyre 15
3.2.4 Pyright 15
4 Research method 17
5 Real-world adoption 18
5.1 Open-source projects 18
5.2 Industrial applications 19
5.2.1 Dropbox 19
5.2.2 Microsoft 20
5.2.3 Instagram 21
5.3 Type hints 21
5.3.1 The typing module 22
6 Results 23
6.1 Theoretical findings 23
6.2 Experimental results 23
7 Discussion 26

8 Conclusions 27
References 29

1 Introduction

Type systems in programming languages can be divided into two major categories: static and dynamic [18]. In statically typed languages, such as C, C++, and Java., types are assigned to variables and functions in the code and never change. This allows to statically check the type correctness of the code without running it. Commonly known advantages of these type systems include code readability, better performance and easier type error detection [18].

Dynamic type systems, such as those of Python, JavaScript or Ruby, determine the types during runtime, additionally allowing a single entity to change types throughout the execution. Some benefits of this approach are:

- Simple and concise code, especially for smaller scripts
- Program components can be easily reused for arbitrary data types
- Working with unstructured or semi-structured data (e.g. from APIs) is much easier
- Faster and cheaper prototyping at early stages of development

For the longest time, there has been a "cold war" between the adepts of the two strictly separated systems. However, as programming languages evolved, many useful dynamic features were adapted by static languages, and vice versa [15]. A particularly interesting trend is the introduction of static type checks into dynamically typed languages, which, in theory, contradicts the pre-existing logic and paradigms. Despite this, some dynamic languages provide direct support for type designation and checking (e.g. GDScript [15]), while developers working with other languages rely on external static type checking tools (e.g. MyPy, PyType, for Python or DRuby/PRuby for Ruby). This study focuses on the reasons behind this phenomenon, as well as evaluation of some measurable benefits it provides.

A crucial quality that unites all of the tools mentioned in the previous paragraph is the support for partially type-annotated code. This allows for gradual introduction of static typization to the codebase without interrupting the normal workflow. For example, Dropbox has added type annotations for over 4 million lines of code in the span of three years, greatly

increasing safety, performance and maintainability of the code [14]. Another proof of the benefits of static typization was found in a study directly comparing statically and dynamically typed languages. In a series of experiments, the participants had to solve identical problems in Java and Groovy. Certain types of tasks has proven to be much easier in Java, because type errors appearing during compilation are significantly easier to understand and correct when compared to runtime errors in dynamically typed languages, such as no-such-method exceptions [9, 13].

1.1 Research questions

This work explores the phenomenon of using static typing in dynamic languages, focusing on Python in particular, as it is by far the most popular dynamically typed language. The goal of this study is to understand the reasons, the scale and the benefits of this practice. More specifically, the study aims to answer the following research questions.

RQ1. Why is static typing used in Python projects?

RQ2. How widespread and how successful the adoption of static typing is among Python developers?

RQ3. How noticeable are the performance benefits that can be achieved by the means of static typing and supporting tools?

1.2 Structure of the thesis

The thesis has 8 chapters and 1 appendix. The first chapter contains a brief overview of the phenomenon being studied, including some minimal background information, as well as the research questions that the thesis aims to answer.

The second chapter presents the necessary information about different type systems, including their benefits and disadvantages.

The third chapter focuses on existing research and developments on the topic, including both theoretical scientific work and various tools developed to facilitate static type checking in Python.

In the fourth chapter the chosen research method is described and justified.

In the fifth chapter the existing research on static typing adoption among open-source Python developers is documented, as well as published cases of static typing used for large industrial projects. In addition, it contains more detailed descriptions of the syntax and modules used for adding type hints to Python code, which is mainly added as an instruction for the readers that aim to use static typing in their projects. It also provides a better understanding of the practice under consideration.

The sixth chapter contains results, including both the information found through reviewing written sources and the experimentally obtained data.

In the seventh chapter the thesis is analysed by stating the limitations of the research and exploring the ways in which it can be improved or continued.

The eighth and final chapter summarizes the thesis, reiterating the key points discovered during the research.

2 Background

In this chapter the relevant history and background information is presented in more detail.

2.1 Static typing

2.1.1 Advantages of static typing

The commonly known advantages of static type systems include the following.

- Type errors are treated as compilation errors and are detected before execution.

  Static analysis has two significant advantages over dynamic analysis. Firstly, the latter requires running the software, which can prove to be very time- and resource-consuming. Static analysis, on the other hand, derives the information from the code itself, checking all possible execution paths. This also leads to the second advantage, which is the fact the certain type errors in dynamic languages may only appear under certain conditions, leading to even harder detection and replication.

- Type signatures serve as a form of documentation and increase readability.

  While documentation in the form of a user manual or comments inside the program can be missing entirely or outdated, the types provide useful information about the objects [18]. It is especially relevant for libraries and other code that is used or modified by the people outside of the team that created it. Some of the most evident examples of why it is important are generic argument names, such as items or id. [14]

- Data types provide strong guarantees and restrictions for the values used.

  Knowing the data type of an object greatly simplifies the effort required to safely process it. Important questions such as “Can this value be null?”, “Does this object have a certain field or method?” are answered in advance, sparing the need to manually check all of it in the code. An additional benefit it provides is higher test coverage achieved on the same set of tests, stemming from the fact that there are

simply less possible input values because of the restrictions imposed by defining the data types.

- Static typing integrates well with IDEs' autocompletion features.

  Most modern IDEs provide lists of available fields/methods based on the type of an object, or signal if the developer uses one that does not exist for that data type. Some also suggest contextual tips, such as argument and variable names based on the name of the data type, or which existing variables to use in a function call.

2.1.2 Disadvantages of static typing

Static typing requires significantly more attention and care put into the software at the early stages of development. Clearly defining the types early on can prove to be a difficult task, especially when working with complex custom classes. Since it is mandatory to specify the argument types for each function, polymorphism in static type systems requires the use of either generics or complex class hierarchies. Correcting mistakes made in the software design stage can be impossible without rebuilding class hierarchies, code duplication or having redundant properties in some objects purely to satisfy type constraints. All of this often leads to higher skill requirements for the developers working with statically typed code, especially in the early stages of development. For this reason, early prototypes are cheaper and easier to make in dynamically typed languages, which can be become increasingly hard to maintain with the growth in the size of the codebase [15].

Another issue with static typing is working with data that has no defined format that is known prior to the execution. For example, if some unstructured data is acquired from an API call or an input stream, it either has to be treated as a very broad data type, prohibiting the use of certain functions that the developer knows can be applied to it, or a new type has to be defined for each possible kind of response.

Overall, static typing improves the safety and readability of the code at the expense of versatility and simplicity.

2.2 Dynamic typing

Dynamic type systems present an alternative approach and are even sometimes defined as systems with no static typing [18]. They do not require specifying types in the code and, most importantly, allow the same code to work with a variety of different types. This makes programs versatile and reusable, but moves all of the type checking to runtime, meaning that type errors often cannot be found before the code is executed, which can lead to critical errors in production, sometimes causing large-scale software failures in the production environment.

2.3 Gradual typing

Gradual typing is a concept defined in [28], which serves as a middle ground between the two type systems mentioned above. While it allows having untyped variables and functions, some parts of the code may be type-annotated and statically type checked before the code execution. Python is a purely dynamic language, but when combined with type hints and some kind of a static type checking tool it becomes gradually typed. It is reassured by the internal design of the type checkers, virtually all of which adhere to this philosophy, following the "gradual guarantee" [25] and thus supporting untyped parts in the code.

Due to being not as strict as classic static typing, it can still cause unnoticed type errors that would arise during the execution time. If too many instances are left unassigned, the benefits can become insubstantial. However, it allows gradually annotating progressively larger parts of the code, yielding immediate value and positive results. Depending on the goal, the developers may annotate the entirety of the codebase step-by-step, or only focus on the parts that are the most sensitive or prone to type errors.

2.4 Python type system

Python’s type system is inherently dynamic, meaning that variable types are determined and enforced during program execution rather than at compile time. Variables do not have an intrinsic type and instead are references to objects, which themselves carry type information. For example, a variable x may initially hold an integer (e.g. x = 5) and later be reassigned to

a string (x = "hello") without any syntactic restrictions. This flexibility is a key point of Python's design philosophy, improving readability and increasing the development pace.

In addition, Python adopts structural subtyping (more commonly referred to as duck typing), where an object's suitability for a given operation is determined by the presence of certain methods or attributes rather than its explicit type [24]. For example, any object implementing the __len__() method can be passed to the len() function, regardless of its class hierarchy. While this approach enhances polymorphism and code reusability, it can also lead to runtime errors if type-related assumptions are violated.

2.5 Evolution of Python

Python is an open-source programming language, so new features are implemented based on the public demand coming from the Python developer community. Python Enhancement Proposals (PEP) serve as a standardized way to formulate these feature requests. They exist in the form of documents that include a detailed description of the proposed feature and the reasoning behind it. After such a document is created by a group of authors, it is submitted to a specialized repository where it is reviewed and, if approved, assigned a number and published. These proposals are then gradually implemented in the language by the core development team behind Python [22].

A particularly important proposal in the context of static typing is PEP 484. It has introduced universal support for type annotations, which are referred to as type hints in the document. These type hints are completely optional and are not processed by the interpreter, but they can still improve readability and greatly facilitate type checking performed by external tools. It is also suggested to use the module called typing to create more complex types [24].

PEP 484 was implemented in Python 3.9 in 2015. Some modifications were performed over the following years. For example, union types, which show that an object can have one of two predefined types, are now available without importing the typing module. However, the core syntax for type hints still remains largely the same, hence the importance of this PEP.

3 Related work

There have been numerous developments in the field of static type checking of Python code. This chapter reviews and summarizes the already established practices, as well as the current research on this topic.

3.1 Dynamic typing

Chen et al. studied the impact of dynamic typing related practices [1, 2], such as reassigning a variable with a value of a different type or dynamically deleting attributes from an object. These practices are allowed by the interpreter, but they can have negative effects on the software quality. The study found a statistically significant positive correlation between the use of those practices and the number of errors (most commonly `TypeError` and `AttributeError`).

A study by Xu et al. [30] on the effect of type checkers showed that out of 40 type-related bugs in a software three type checkers were able to find 35% with no type hints and 72.5% after the code was annotated. Another study by Khan et al. [12] performed this research on a much larger scale, examining a representative sample of 400 defects randomly selected from a corpus of over 10 000 issues from 500 000 Python projects. The results demonstrate that over 15% of the defects can be prevented using type checkers, with 6 annotations on average required to catch each defect.

Guo et al. [8] performed a series of experiments to estimate the effectiveness of type inference tools for automatic code annotation and type checking. The particular importance of the topic is in its appeal to a wide range of developers, since type inference does not require any effort from the programmer, and the tools instead deduct the types based on code analysis. The results demonstrate that the tools manage to correctly annotate around 50% of the code on their own. However, there is a trade-off between coverage and accuracy, meaning that a balance must be struck between the strictness of the inference tool and its capacity to annotate more ambiguous parts of code.

Reimann and Kniesel-Wünsche [14] acknowledge the problem of static type checking in data science in particular, since many data science libraries do not provide type signatures for their functions, and these libraries are used by programmers of various skill levels. They approach this issue by designing their own domain-specific language, that uses type inference and type hints (where available) to statically check the functions from external libraries and organize them into type-safe pipelines.

3.2 Existing tools for static typing in Python

It is worth mentioning that the emergence of the PEP 484 standard was heavily inspired by the external type checking tools that were already existing at the time [24]. Before it was proposed and a unified way for declaring types was established, type checkers would either rely on type inference or on more unsystematic function type annotations introduced in PEP 3107 [23], as well as type comments or other custom notations. The most notable type checking tools are listed in the subsections below.

3.2.1 MyPy

MyPy was originally developed by Jukka Lehtosalo in 2012 as a type checker for a modified variant of Python. In 2013 he joined Dropbox and the development of the tool was continued there. The team behind MyPy was largely responsible for creating the PEP 484 standard, which brought the Python type annotations into conformity with the demands of the community [14]. Since then the type checker (as well as many others) has been using the default Python type hints, removing the need for any external constructions. Although MyPy checker was widely adopted inside Dropbox itself, the project has always remained open source, allowing the general public to use it for their needs.

To provide an additional performance boost, a MyPyc compiler was designed alongside MyPy. It utilizes the type system of the latter to efficiently compile Python modules to lower-level CPython C extension modules. Many operations can be sped up this way, including function calls (which are translated to C function calls), as well as working with primitive types or native classes.

3.2.2 PyType

PyType is another tool designed to statically check Python programs for type-related issues. It is an open source project developed by a team from Google. Unlike MyPy, it relies on type inference more than on type annotations specifically, meaning that it will assign broader types (e.g. union types, such as Int | Str) to unannotated objects in order to avoid type errors [7]. This design allows to use it as a powerful type inference tool for type annotation as shown in [30]. It prioritizes practicality and ease of use over strictness, making it suitable for projects where gradual typing is critical. Like most other tools, it fully supports standard Python syntax and type hints added in PEP 484.

3.2.3 Pyre

Pyre, created by Meta, focuses on scalability and incremental type checks for large codebases, such as that of Instagram [21]. It can interactively highlight type errors in real time, facilitating detecting defects as early into development as possible. Pyre also introduced Pysa, a static analysis tool for identifying potential security risks in the code. While Pyre is highly performant, its adoption outside Meta has been slower compared to MyPy and Pyright, possibly due to its focus on Facebook's internal workflows.

3.2.4 Pyright

Pyright is a high-performance static type checker developed by Microsoft with the main focus on speed and developer experience [32]. Unlike MyPy, Pyright runs in-memory, significantly reducing latency and making it capable of providing instant, responsive feedback, which is essential for its use in VS Code. It supports all of the type-related PEPs, including the latest experimental ones. More about the use of Pyright can be found in section 5.2.2.

In summary, Dropbox is developing and maintaining MyPy, which it uses to type check almost the entirety of its codebase [14]. Meta is developing Pyre, a static type checker that

they use for their own code at Facebook and Instagram [4]. Another open-source type checker, PyType, is developed by Google [7]. Their official style guidelines strongly suggest the use of both type annotations and static type checkers where possible [6]. Lastly, Pyright is developed and maintained by Microsoft and used in their products, such as VS Code.

4 Research method

This chapter describes the methods, tools and sources used in this research.

Since the goal of the research is to study an optional practice that some developers choose to utilize, the case study research method was found to be the most fitting for that purpose. This allows for in-depth analysis of real-life adoption cases of static typing in Python projects and valuable insights on the situation with very large commercial projects, which would be impossible to acquire by performing experiments. In addition to this, some experiments were performed to incorporate a more hands-on approach and better understand the subject of study from personal experience.

According to one of the most fundamental books on case study research [31], this method is best suited for situations in which “your questions require an extensive and in-depth description of some social phenomenon” [31]. A multiple-case research study allows to capture different perspectives on the prerequisites, context, implementation process and consequences of using static typing in Python projects. It can also help the reader form an understanding of which problems can be resolved by adopting the practice.

For the practical part, a simple benchmark program in Python was written and type annotated, and it included the code to measure execution time and print it out. The program was then run using regular Python interpreter. Afterwards, it was compiled with MyPyc into a C extension, which was imported in a new Python file. The snippets of the code can be found in Appendix 1.

5 Real-world adoption

This section lists the known cases of static typing in Python being adopted in different projects. While open-source usage can be studied, analysed and quantified, the industrial applications are mostly known from reports and articles by the companies' own software engineers.

5.1 Open-source projects

A comprehensive study was performed in 2020, analysing over 70,000 publicly available Python repositories to determine how often and in what ways developers use types in Python [26]. The results demonstrate that by the time when the study was performed the type hints had not yet been widely accepted and utilized by the open-source Python community. Less than 2,700 repositories (3.8%) had any type annotations at all, while only 318 (12%) of those passed type checks by MyPy. [26] This could be due to type checks being too strict, discouraging the developers from type checker tools altogether, or to the fact that type hints are used for reasons other than finding type errors, namely documentation, readability and IDE autocompletion.

A developer survey carried out in 2024, on the other hand, yielded totally different results. According to the findings, 88% of the respondents always or often use types in general and 67% use MyPy specifically. The most common reasons for using types are "Better IDE Support" (59%), "Preventing Bugs" (49.8%) and "Documentation" (49.2%). [19] However, the drastic difference with the findings of the previous study may not only be due to the time gap between the two but also can be caused by several other factors. Firstly, a voluntary survey about typing in Python can mostly attract developers that are interested in types and actively use them, making the sample non-representative. [19] Secondly, since the survey was conducted by a collaborative effort including Microsoft and JetBrains, a disproportionately large part of the respondent group could be their clients using their IDEs (over 84% of the sample use PyCharm or VS Code), meaning that they are encouraged to use types for IDE integration. Still, despite those factors, the results may indicate an increase in the popularity of static types in Python among the open-source developers.

5.2 Industrial applications

Naturally, a lot less information is available about the situation in the corporate sector. However, there is evidence suggesting that static typing is used extensively by software companies using Python, with larger ones going as far as to develop their own Python static type checkers. Below are some of the published cases of applying Python type hints and type checking tools in large industrial projects.

5.2.1 Dropbox

Dropbox's engineering team encountered significant challenges as their Python codebase grew to millions of lines of code [14]. Like many companies primarily using Python, they initially benefitted from the language's dynamic nature and flexible type system during different features' early development stages. However, as the codebase expanded to over 4 million lines of code maintained by hundreds of developers, the lack of static types in the code became increasingly concerning. Runtime type errors could not be found before execution and caused issues in production, while the absence of type annotations made the code harder to understand and modify safely, especially by the developers that did not write it originally. These issues became particularly noticeable during large-scale refactors or when hiring new programmers who needed to understand complex, untyped code.

A team of developers at Dropbox embarked on a prolonged journey to combat this problem, with an initiative to gradually annotate and type check the entirety of the codebase. MyPy, the most popular type checking tool for Python, was being developed as a side product of this effort. While it has always been an open-source product, its core development team consisted of Dropbox employees, allowing for constant modification and tailoring according to received feedback. To improve software performance the responsible team additionally developed a compiler called MyPyc that proved to be very effective in speeding up statically typed code, further motivating various departments to participate in statically typing their code.

The results of this gradual typing initiative were substantial. Early detection of type-related errors, lead to a reduced number of runtime errors and better overall system reliability. The added type annotations served as instantly visible documentation that made the codebase

more understandable, particularly for developers recently assigned to work on a certain part of code. IDEs were independently developing overtime to use type hints for rudimentary type checks and, more importantly, better autocompletion, suggestion and navigation features. Continuous integration was also significantly facilitated, as MyPy could verify that newly made modifications do not endanger the type safety of other parts of the codebase. Significant performance improvements were achieved through code compilation with the MyPyc compiler. While the transition required significant effort, Dropbox's experience demonstrated that even very large Python codebases could successfully adopt static typing through careful, incremental implementation.

5.2.2 Microsoft

Microsoft’s Visual Studio Code, and its Python extension in particular, has gained popularity as an IDE for Python development, in part due to its built-in support for static typing. To provide fast and accurate type checking both for their own internal code and for the VS Code users, Microsoft developed Pyright, a static type checker with a focus on speed and ease of use. It proved to be the right design for its use in the VS Code’s Python extension, where low delay and fast and responsive real-time performance is crucial for a smooth coding experience. Pyright also introduced strict and partial type-checking modes, allowing developers to switch between varying levels of type strictness based on what the project requires. Microsoft’s development and wide adoption of Pyright demonstrates the company’s perceived value in static typing in Python and the benefits of its integration into developer tools, which include improved autocompletion and early type error detection. Additionally, since Pyright was developed as an open source project, it receives contributions from the Python community, and is widely spread among companies and individual developers alike. Its integration into VS Code promotes the practice of static typing among the users of that IDE, who, in turn, praise its capabilities and features (see section 5.1).

5.2.3 Instagram

Instagram is one of the most popular social networks in existence, with over 2 billion active monthly users worldwide [29]. The server codebase consists of several million lines of Python code and is worked on by hundreds of developers [10]. Furthermore, the company maintains a continuous integration pipeline, with the goal of having no more than an hour for the code to get from a developer commit to being applied in production. This requires extensive code verification that can be done in a very limited timeframe, for which static type checking plays and essential role.

Being a part of Meta, Instagram mostly relies on the Meta-developed type checking tool called Pyre. However, in addition to using it for regular type checking, software engineers at Instagram combine it with other tools, such as automatic code modifiers (or "codemods"), to derive the missing type annotations from Pyre error message and insert them into the code [11]. It is accompanied by other heuristic practices, such as inferring function return types with isInstance() or explicitly specifying return type as None where possible. These practices have reportedly shown impressive result, such as helping annotate over 10% of all the functions in the Instagram codebase, which, in turn, allowed for automatic refactoring of large amounts of deprecated code. This implies that type annotations are often very easy to generate, which, in turn, means that with gradual typing a large portion of the codebase could be annotated with relatively low effort.

5.3 Type hints

Default Python syntax supports type annotations for variables, function arguments and function return types. Type hints are written using “`:`”and “`->`”, for example

--

```
a: int = 3

def sum(array: list[float]) -> float:
```

Type hints can contain primitive types, e. g. `int, str`, standard container types: `list, dict, tuple`, and any user-defined or imported classes. Starting from version 3.10,

Python supports the use of the "|"operator, which serves as a built-in way of defining union types. For example, a function that can return an `int` or `null` can be annotated as

```
def lookup(key: str, dictionary: dict[str, int]) -> int | null:
```

5.3.1 The typing module

The typing module, which is available with the standard Python library, provides additional functionality related to type annotations and type checking. Some of the structures that it provides have been implemented in the default syntax, so they are now obsolete. For example, `Union` or `Optional` from the typing module can be rewritten with the use of the "|" operator (in Python 3.10 and later versions) This transition highlights the increase in understanding of the value provided by static typing among the Python community.

However, the typing module still provides a wide range of useful tools and is regularly updated with new features, leaving it instrumental for static typing in Python. The important concepts from the module are, among others,

- `Any` type, which can be used for the parts of the code that are either not ready to be typed, or their types are entirely unknown prior to execution.
- `TypeVars`, a versatile tool to define and use type variables, for example, in generics.
- More generic abstract container types, such as `Sequence` or `Iterable` can be used for type annotations in cases where the exact container type is unknown or unimportant

Other importable modules from the default Python library (e.g. `collections`) also offer a number of type definitions which can be imported and used in the code.

6 Results

This chapter presents the obtained results and answers the research questions posed in the thesis.

6.1 Theoretical findings

With the rapid growth of Python codebases, many companies and developers begin to understand the value of static typing. Introduction of type hints has proven to significantly improve readability, especially in cases where multiple developers have to work on or use the same code. More specifically, function type signatures provide immediate context for how to use a function and what is the result, without the need to read external documentation. For smaller projects, the most important and commonly mentioned benefit is improved IDE support, unlocking features such as autocompletion and context-based recommendations of attribute and method names for custom objects.

The additional use of static type checkers helps prevent and correct type errors, as demonstrated by both academic studies and real-world examples. Most type checking tools support the gradual typing paradigm, allowing for the code to be only partially annotated, ignoring the untyped parts. This is essential when introducing typing to very large projects, as it can provide immediate results and facilitate gradual step-by-step type annotation.

6.2 Experimental results

Python has no built-in support for static typing, meaning that type hints will have no impact on performance on their own, as they are discarded during execution. However, external tools can not only make statically typed code safer, but also boost its execution speed. It can be done by optimizing the program, making it less versatile, but improving performance for the specific types which are known to be used in it. One such tool is MyPyc compiler, which uses type hints and type inference to compile Python code into C extensions.

In the practical part of the research, MyPyc compiler has been used with various type-annotated structures in order to see the performance benefits yielded in each case. The results demonstrate a noticeable decrease in execution time when the compiler is used, especially effective in the case of recursion, since type-annotated Python functions can be efficiently converted to functions in C, making function calls much faster [17]. Type-annotated Python loops can also sometimes be compiled into for-loops in C, similarly leading to a noticeable increase in execution speed. Figure 1 demonstrates the execution times of the same code, with the blue bars representing the time it takes to run the code with the standard Python interpreter, and the orange bars demonstrate the time that the same programs take when type annotated and compiled with MyPyc. Figure 2 illustrates the ratios of those two values.

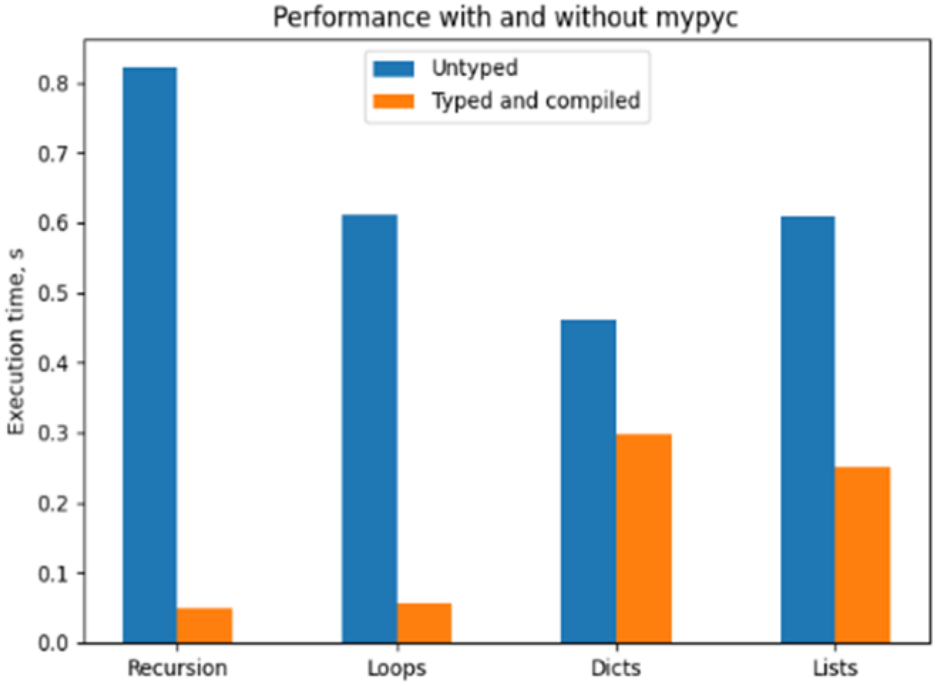

Figure 1. Execution time with MyPyc for typed Python code

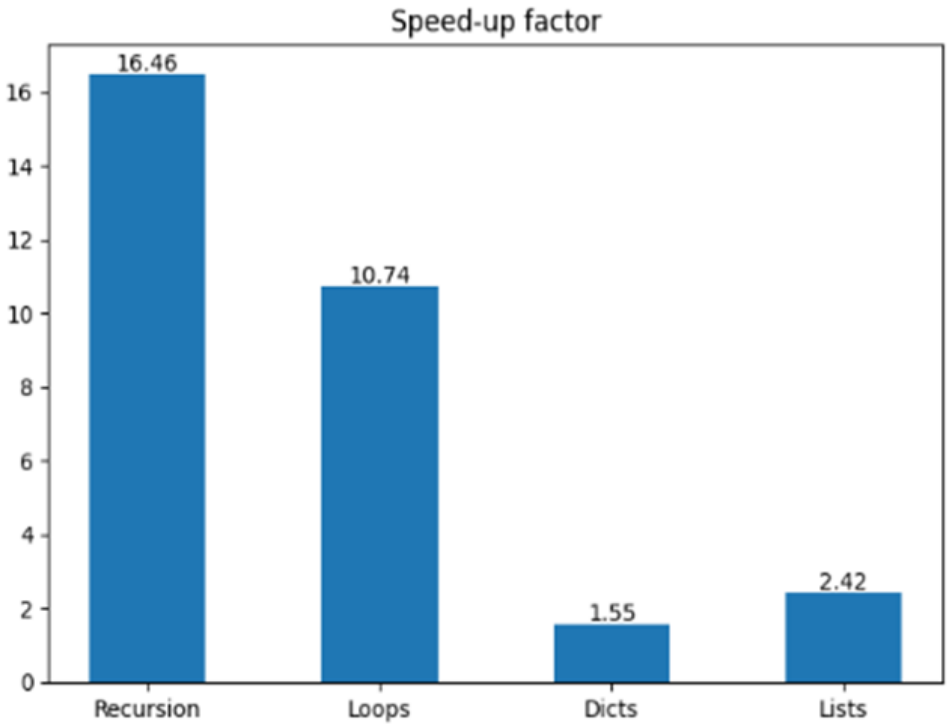

Figure 2. Speed-up factor with MyPyc for typed Python code

Overall, the measurements demonstrate that an increase of up to 16 times in execution speed can be achieved with the help of static type checking and tools relying on it. This emphasizes the importance of static type checking in Python as a way of improving software performance.

7 Discussion

This chapter presents the potential ways in which the research can be improved and expanded, as well as the limitations of the current work.

While the subject is explored relatively well based on the available information, the information itself is somewhat scarce. This is especially noticeable when describing applications in commercial projects, presumably due to confidentiality, e.g. non-disclosure agreements signed by the developers, or overall low priority of publishing results, with the main focus understandably being on actual software development. Another potential problem could be that successful adoption cases may be documented more often than unsuccessful. There could be two reasons for that, the first one being that as the articles can aim to present the achievements and maintain a positive image of the company publishing it, and the second being that a successful implementation of static typing is a more substantial topic for an article.

To expand the research, a high-quality survey on the use of type hints could be carried out, significantly clarifying the actual popularity and scale of the use of static typing in Python. However, selecting a representative sample of Python developers is worthy of being a thesis subject on its own, so it is left out of the scope of this research. Regardless, it would be the best way to definitively answer the question of how widespread the described phenomenon actually is.

In the experimental part a future continuation could be in taking a large project and statically type the entirety of it, measuring the acquired performance benefits and, possibly, other quality metrics.

Another potential path for future research is to extend the scope beyond Python and look at the situation in other dynamically typed languages. For example, transition from JavaScript to TypeScript provides a variety of benefits [3]. Another case could be made with GDScript, a language specifically used for the Godot game engine, which also supports optional type annotations and even provides performance benefits out-of-the-box for statically typed code [5] (unlike Python, which requires external tools for that).

8 Conclusions

This thesis studies the relatively new phenomenon of using static typing in Python code and the impact it has on Python software development. Although it is not extensively documented, there still are available materials describing different aspects of this practice, including successful adoption cases, experimental research of its impact on software quality, studies and surveys approximating how widely it is used.

Python is a dynamically typed language, meaning that variable and function types are not known prior to execution and can change in runtime. However, it supports a feature called type hints, which allows to arbitrarily assign types to variables and functions according to the rules and syntax described in several PEP standards. While they are discarded during program execution, they can be used by developers working on the code and external code analysis tools.

Python's dynamic type system makes it versatile and concise, however, maintainability issues arise when the codebase size grows to thousands or even millions of lines of code. Introduction of static typing alleviates this problem in several ways. Firstly, type signatures increase readability of the code by instantly showing information that a developer needs but may not know, such as argument types and return type for functions. Secondly, static type checking tools allow to examine the code for type errors without executing or testing it, saving a large amount of time and resources, as well as providing strong guarantees about the code's type safety. Lastly, static type annotations facilitate the use of autosuggestion and autocompletion features in modern IDEs, which prompts some developers to use static typing even for smaller projects.

Additionally, there are tools that improve performance by removing the need to dynamically check the types as it is done in normal Python execution. One such tool is MyPyc, a compiler that converts statically typed python code to importable C extensions. In the experiments performed for this thesis the runtime of certain example benchmarks was decreased by up to 16 times.

There is conflicting information about the frequency of type hints and static type checking being used in open-source Python projects. One study indicates that in 2020 only around

3.8% of open-source Python projects used type hints in any regard, while in a survey from 2024 88% of the respondents claimed to be using types regularly. The scale of static typing being used in commercial projects can be indicated by occasional published reports from different companies [14, 20, 11], as well as by a number of large corporations going as far as designing their own static type checkers.

Overall, static type annotations improve readability of Python code and is especially important for large projects. Although static type checks are not enforced by the language or the interpreter, there is a wide variety of external tools available to perform type checking and improve the safety and performance of the software.

References

1. Chen, Z., Chen, L., Yang, Y., Feng, Q., Li, X. & Song, W. 2024. Risky Dynamic Typing Related Practices in Python: An Empirical Study. ACM Transactions on Software Engineering and Methodology. Vol. 33, no. 6, doi: 10.1145/3649593

2. Chen, Z., Li, Y., Chen, B., Ma, W. Chen, L. & Xu, B. 2020. An Empirical Study on Dynamic Typing Related Practices in Python Systems. IEEE/ACM 28th International Conference on Program Comprehension (ICPC), Seoul, 2020, pp. 83-93, doi: 10.1145/3387904.3389253.

3. Cherny, B., 2019. Programming TypeScript: making your JavaScript applications scale. O'Reilly Media. ISBN: 9781492037606.

4. Facebook, 2021. ELI5: Pyre - Fast Error Flagging for Large Python Codebases. [online] Available at: https://developers.facebook.com/blog/post/2021/05/10/eli5-pyre-fast-error-flagging-python-codebases/ [Accessed 18 July 2025].

5. Godot Engine, 2025. Static Typing in GDScript. [online] Available at: https://docs.godotengine.org/en/4.4/tutorials/scripting/gdscript/static_typing.html [Accessed 18 July 2025].

6. Google, n.d. Google Python Style Guide. [online] Available at: https://google.github.io/styleguide/pyguide.html [Accessed 18 July 2025].

7. Google, n.d. Pytype GitHub Repository. [online] Available at: https://github.com/google/pytype [Accessed 18 July 2025].

8. Guo; Y., Chen, Z., Chen; L., Xu; W., Li; Y., Zhou, Y. & Xu, B. 2024. Generating Python Type Annotations from Type Inference: How Far Are We? ACM Transactions on Software Engineering and Methodology. Vol. 33, no. 5, doi: 10.1145/3652153

9. Hanenberg, S., Kleinschmager, S., Robbes, R., Tanter, É. and Stefik, A. 2013. An empirical study on the impact of static typing on software maintainability. Empirical Software Engineering, October 2013 19(5) doi: 10.1007/s10664-013-9289-1

10. Instagram Engineering, 2021. Let Your Code Type-Hint Itself: Introducing Open-Source MonkeyType. [online] Available at: https://instagram-engineering.com/let-your-code-type-hint-itself-introducing-open-source-monkeytype-a855c7284881 [Accessed 18 July 2025].

11. Instagram Engineering, 2021. Static Analysis at Scale: An Instagram Story. [online] Available at: https://instagram-engineering.com/static-analysis-at-scale-an-instagram-story-8f498ab71a0c [Accessed 18 July 2025].

12. Khan, F., Chen, B., Varro, D. & McIntosh, S. 2022. An Empirical Study of Type-Related Defects in Python Projects. IEEE Transactions on Software Engineering, vol. 48, no. 8, pp. 3145-3158, 1 Aug. 2022, doi: 10.1109/TSE.2021.3082068.

13. Kleinschmager, S., Hanenberg, S., Robbes, R., Tanter, É. & Stefik, A. 2012. Do static type systems improve the maintainability of software systems? An empirical study. Conference: Program Comprehension (ICPC), 2012 IEEE 20th International Conference on June 2012, doi: 10.1109/ICPC.2012.6240483

14. Lehtosalo, J. 2019. Our journey to type checking 4 million lines of Python. [online] Dropbox.Tech. Available at: https://dropbox.tech/application/our-journey-to-type-checking-4-million-lines-of-python

15. Meijer, E. & Drayton, P. 2004, October. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. OOPSLA.

16. Microsoft, n.d. Pyright: Static Type Checking for Python. [online] Available at: https://microsoft.github.io/pyright/ [Accessed 18 July 2025].

17. MyPy, n.d. MyPyc Documentation: Performance Tips and Tricks. [online] Available at: https://MyPyc.readthedocs.io/en/latest/performance_tips_and_tricks.html [Accessed 18 July 2025].

18. Pierce, B.C. 2002. Types and Programming Languages. MIT Press. P. 19

19. Pollack A. 2024. Typed Python in 2024: Well adopted, yet usability challenges persist. Engineering at Meta. https://engineering.fb.com/2024/12/09/developer-tools/typed-python-2024-survey-meta/

20. Pussinen, J. 2021. Professional-grade MyPy configuration. TECH Sep 30, 2021 https://careers.wolt.com/en/blog/tech/professional-grade-MyPy-configuration

21. Pyre, n.d. Pyre Documentation. [online] Available at: https://pyre-check.org/docs/ [Accessed 18 July 2025].

22. Python Software Foundation, 2000. Python Enhancement Proposals (PEP 1) - PEP Purpose and Guidelines. [online] Available at: https://peps.python.org/pep-0001/ [Accessed 18 July 2025].

23. Python Software Foundation, 2006. Python Enhancement Proposals (PEP 3107) - Function Annotations. [online] Available at: https://peps.python.org/pep-3107/ [Accessed 18 July 2025].

24. Python Software Foundation, 2014. Python Enhancement Proposals (PEP 484) - Type Hints. [online] Available at: https://peps.python.org/pep-484/ [Accessed 18 July 2025].

25. Python Software Foundation, 2025. Python Standard Library - Typing. [online] Available at: https://docs.python.org/3/library/typing.html [Accessed 18 July 2025].

26. Rak-Amnouykit, I., McCrevan, D., Milanova, A., Hirzel, M., & Dolby, J. 2020. Python 3 Types in the Wild: A Tale of Two Type Systems. Association for Computing Machinery. November 17, 2020, Virtual, USA doi: 10.1145/3426422.3426981

27. Reimann, L., Kniesel-Wünsche, G. 2023. Safe-DS: A Domain Specific Language to Make Data Science Safe. Conference: 2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER) doi: 10.1109/ICSE-NIER58687.2023.00019

28. Siek, J., Taha, W. 2007. Gradual Typing for Objects. In: Ernst, E. (eds) ECOOP 2007 – Object-Oriented Programming. ECOOP 2007. Lecture Notes in Computer Science, vol 4609. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-73589-2_2

29. Statista, 2025. Instagram - Statistics & Facts. [online] Available at: https://www.statista.com/topics/1882/instagram/#topicOverview [Accessed 18 July 2025].

30. Xu, W., Chen, L., Su, C., Guo, Y. Li, Y. Zhou, Y. & Xu, B. 2023. How Well Static Type Checkers Work with Gradual Typing? A Case Study on Python. IEEE/ACM

International Conference on Program Comprehension (ICPC)
doi: 10.1109/ICPC58990.2023.00039

31. Yin, R. K. 2009. Case Study Research: Design and Methods, 4th Edition. Sage.

Appendix 1. Code for experiments

This appendix contains is the code used for measuring the effects of static typing and MyPyc compiler on the execution time of Python programs. To isolate the effects for different language elements and structures, the benchmark functions were simplified as much as possible and mostly contain one operation type. The specific values for the test were selected arbitrarily to make the times for different experiments comparable for easier visualization and understanding, while keeping them large enough to negate fluctuations coming from outside of the studied element. Since compilation does not alter the asymptotic time of execution, it does not cause any problems, and only the speed-up factor is of importance.

```
import time
from typing import Callable, List, Dict

def rec(n: int) -> int:
    if n <= 1:
        return n
    else:
        return rec(n - 1) + rec(n - 2)

def loop(n: int) -> int:
    x:int = 0
    for i in range(n):
        x += 1
    return x

def dic(n: int) -> int:
    d: Dict[str, int] = {'-1': -1}
    for i in range(n):
        d[str(i)] = i
        d[str(i - 1)] += 1
    return d[str(n - 1)]
```

```
def lists(n: int) -> int:
    lst: List[int] = list(range(n))
    result: int = 0
    for i in range(n):
        lst[i] += 1
        result += lst[i]
    return result


test_funcs: List[Callable[[int], int]] = [rec, loop, dic,
lists]
test_sizes: List[int] = [33, 2 * 10 ** 7, 10 ** 6, 6 * 10 **
6]


def run_tests() -> List[float]:
    results: List[float] = []
    for f, n in zip(test_funcs, test_sizes):
        t0 = time.time()
        f(n)
        results.append(time.time() - t0)
    return results


print(", ".join(map(str, run_tests())))
```