Enhancing Code: Refactoring and Adding Type Hints

CS4991 Capstone Report, 2023

Matthew Beck
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia, USA
mtb2tgk@virginia.edu

ABSTRACT
When working in an agile environment, a company's repository can become dense and repetitive. Additionally, the incorrect types of variables can be used when using Python. To combat this while interning with a team at a Boston-based technology firm, I refactored the code to eliminate redundant functionality sourced from different locations in the code. I added type hints to make the code safer. Additionally, I went through my team's entire API and added type hints to the method signatures. By refactoring the code, I eliminated hundreds of lines of code and made the repository less dense. Adding type hints enabled developers to perform static type checking while enhancing autocompletion in developers' IDE. In the future, more code needs to be refactored, and more type hints added.

1. INTRODUCTION

Group work often leads to chaos, and certain aspects, such as organization and communication, must be remembered to promote productive work. This is also true for agile software development. Multiple people work on different facets of the project simultaneously, resulting in a code repository that begins to be filled with redundant code. Additionally, when using a dynamically typed language such as Python, the project becomes vulnerable to incorrect types used throughout the code, introducing bugs.

During my internship this past summer, I quickly joined a team and was assigned to work on their testing platform. Problems were apparent throughout their code: the same functionality was written in multiple locations, all of the function's callers performed the same actions post-call, and incorrect types were being passed in various locations. These issues are critical to address because of the bugs introduced, the potential for bugs, and the overall decrease in developer productivity. Transforming a repository to a concise, type-safe code allows developers to work efficiently and focus on their projects. My approach to this problem was adding type hints to their testing API while refactoring their code to have related functionality in the same place.

2. RELATED WORKS

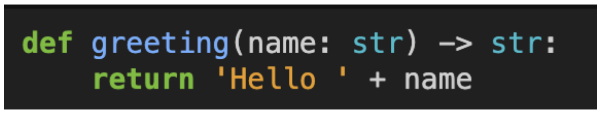
The dynamically typed state of Python and the problems that may arise because of it are very well documented. These issues involve scoping problems and a need for early feedback (Lewis, 2021). These problems can be combated with Python's introduction of type hinting in version 3.5. According to CodersLegacy, adding type hints to Python code has many benefits, such as improving readability, debugging, and IDEs. Issues with instituting type hints include the time it takes to add post-production and an increase in startup time.

The concept of refactoring code has been around since the beginning of computer science, with its benefits heavily researched and documented. Gillis (n.d.) details precisely how refactoring not only makes the developers' lives easier but also makes the code better and less prone to bugs. Refactoring code has been shown to increase readability while reducing complexities. Refactoring makes the code cleaner, more efficient, and maintainable. Perhaps the biggest reason to refactor code is the benefit of developers knowing where standard core functionality lies, dramatically improving efficiency and allowing them to focus on the most significant problems.

3. PROJECT DESIGN

3.1 TYPE HINTS

In Python's documentation, type hints were added in PEP 484. This type hinting enables static type checking for third-party applications. The structure of adding type hints to function declarations involves inserting a colon after each parameter followed by their respective type. The return type can be included by inserting an arrow and the return type between the closing parenthesis and colon of the function header.



```python
def greeting(name: str) -> str:
    return 'Hello ' + name
```

Figure 1: Type Hinting Example

In Figure 1: Type Hinting Example, a greeting function is being defined. This function takes one parameter called name, which is of string type. The parameter name is of string type due to the ": str" added after the parameter. It is also known that this function returns a string as dictated by adding "-> str."

During my summer internship at a Boston-based technology firm, I worked with my team's testing platform to use the framework laid out to add type hints. I was tasked with adding type hints to the main API within the testing platform. To start this project, I began with a discovery phase. During this phase, I spent ample time getting well acquainted with the code within the API, which stretches across approximately 30 files. When working within an Agile environment, you break down projects into tasks called stories. These stories are typically atomic and are assigned a story point, a Fibonacci scale of approximately the complexity of the task. For example, a story with 1 point is considered trivial and can be completed within an hour or two.

Meanwhile, a story with 5 points is considered complex and may take up to a week or more to complete. My manager left the decision up to me whether I wanted to break the project into multiple less complex stories or one significant story covering the entire project. I decided to group the files based on their dependencies during my discovery phase. One file within the repository contained over 3000 lines of code and was the most complex, so I left this story to last to ensure that all of the related dependencies would be type-hinted by the time I began work on that file. For most of these stories, I gave a story point of 3, indicating that I anticipated resolving the story within the week, while for some of the more significant stories, I gave 5 points.

Once the discovery phase was complete and approved by my teammates, I was allowed to begin typing hinting the API. I went story by story, completing each group of files in each story. For each function in a file, I inspected how parameters were used and how the function's return was used to determine the type. This often involved going from function call to function call to determine how each function was being used and the types being passed.

Once I completed my initial pass of the files, I went back through to ensure everything appeared accurate. After that, I used a Python third-party extension named Mypy. This extension enables static type checking, which I used on each file. Mypy would return errors if the return type of a function did not match how it was being used as a parameter in another function.

The company I was working for had a testing requirement for adding tests into the framework. The authoring engineer had to run iterative tests to ensure that the test they had created was accurate. However, since my changes affected the entire testing framework, I was required to run the whole test suite comprising over 2,500 tests. This was by far the most time-consuming part of the project, but necessary to ensure that my changes were stable.

3.2 Refactoring Code

My second major project of the summer included refactoring a variety of functions found within the same API. This project began like the type hinting project with a discovery phase. During this discovery phase, I looked throughout the API for redundant functionality with the help of my mentor. Once redundant code was identified, I was tasked with deriving a plan to merge the functionality to create a more cohesive API. Once I had derived a plan and the plan had been approved, I went through with implementing my plan and then testing the changes I had made.

One of the most considerable functions I had refactored was finding an expression within log records. Two functions performed this functionality in slightly different ways. One instance was passed a time stamp, the path to the log record, and the expression to be found. It started at that time stamp in the log record and then searched until the end of the file for the given expression. The other way they implemented this was by using an object called a log follower. You should provide the log file when this log follower was instantiated. Then, when searching for a given expression, you would call a function that would explore the file for the expression from when the log follower was instantiated until the end of the file.

I merged the timestamp functionality into the log follower methods, mainly with how it was being called. I inspected where the find log record method was being called and where they were creating a timestamp. At this point, I replaced the creation of a timestamp with an instantiation of a log follower. Then, where the actual find log record method was being called, I replaced it with a call to the log followers method

Some obstacles were encountered through this process. Specifically, a problem occurred when a timestamp was retrieved before the log file was created. Since the log file was yet to be made, a log follower could not be instantiated on that log file. To overcome this, I worked with my mentor to use the existing functionality of the log follower functions to start at the beginning of the file.

4. RESULTS

Once I finished a project during my internship, I presented the results to the rest of the team. For the type-hinting project, I demonstrated how static type checking could be used to check the dependencies of types throughout the code. Additionally, I showed how the developers' integrated development environments had improved auto-completion, allowing them to complete their code more efficiently and accurately. While I was finishing the addition of the type hints, I encountered various bugs in the code. In one instance, a parameter was passed as a boolean and then overwritten as an integer. This parameter was used later in the function as a parameter to another function that expected a boolean. Correcting this made the code more stable and improved the team's testing results.

Through refactoring the code, the team saw a variety of benefits. These benefits ranged from return types that were easier to work with to more consistent code that improved the readability and the developers' understanding of the code. Overall, my changes dramatically helped the developers on my team to spend less time worrying about types and which functions to use, allowing them to focus directly on solving more complex problems.

5. CONCLUSION

When I joined my team at the beginning of the internship senior members expressed their concerns over type safety, readability and efficiency of their code repository. I added type hints to their main API which allowed the use of static type checkers. Additionally, by adding these type hints, developers' IDEs became more intuitive with their autocompletion becoming more comprehensive. I also refactored code within the repository, combining like functions and cleaning output types. Through refactoring the code, the repository is more readable and understandable, allowing the developers to focus on the main task at hand. Overall my additions allowed the developers on my team to become more efficient and work at a higher level.

6. FUTURE WORK

The repository where I completed my work is very dense. Hence, I was unable to fully refactor all of the code over the course of the summer. For future steps, more refactoring should be performed in order to improve the repository while reducing its density.

Additionally, the company made heavy use of context managers in their code. Due to the version of python they were using, I was unable to add appropriate type hints to these functions. As they take steps to upgrade the version of python they are using, type hints should then be added to the context manager methods

REFERENCES

CodersLegacy. *Benefits of type hinting in Python*. (n.d.) Retrieved September 29, 2023, from https://coderslegacy.com/python/benefits-of-type-hinting/

Gillis, A. S. (n.d.). *What is refactoring (code refactoring)?* TechTarget. Retrieved September 29, 2023, from https://www.techtarget.com/searchapparchitecture/definition/refactoring

Lewis, M. (2021, December 9). *The struggle of dynamically typed languages. Medium*. Mark Lewis. Retrieved September 29, 2023, from https://drmarkclewis.medium.com/the-struggle-of-dynamically-typed-languages-ef91a87164a1