Formal Verification of Python 2 to 3 Translations

Word Count: 5,079

1 Introduction

Since its creation in 1991, Python has risen to become one of the world's most popular programming languages. This rise has not come without significant challenges, however. One of the most notable challenges that Python encountered was the transition from version 2 of the language to version 3. Python 3 introduced a vast number of new features and quality-of-life improvements for Python programmers. However the vast number of changes also meant that most code written in Python 2 was not compatible with the new version, which resulted in a multi-year long transition period where developers had to either migrate their Python 2 codebases to Python 3 or improvise solutions to let their codebases work in both Python 2 and 3. One migration solution was the use of Python 2 to 3 translators, especially 2to3, the official Python 2 to 3 translator created by the Python Software Foundation (PSF), the organization which manages the development of the Python language. However, 2to3 was not able to address all aspects of the 2 to 3 transition, so the usage of translators like 2to3 eventually fell in favor of maintaining codebases that can work under both versions of Python [10].

Although 2to3 is an effective software program, the vulnerability of software like 2to3 to edge cases — situations which occur so rarely developers may not foresee their software encountering them, thus leading to bugs in their software — has resulting in the rising popularity of *formal verification* — the practice of verifying that a program conforms to a defined specification with the rigor of a mathematical proof. Such verification means all possible cases are accounted for, meaning that edge cases are impossible if the specification is defined well enough. Currently, formal verification is a niche field, with most real-world applications of formal verification in industries such as aviation, rail, and computer chip design where having software that can be proven to do what developers believe it should do is of the utmost importance [14]. However, research into the application of formal verification into other areas is ongoing. The purpose of this research is to study the effectiveness of using formal verification to create and verify a Python 2 to 3 translator.

2 Literature Review

2.1 Python 2 vs. 3

The core of the Python language is a program known as the interpreter, which reads Python source code written by a programmer and executes the code line-by-line. A 'version' of the Python language is simply a version of the interpreter, and each new version of the interpreter includes updates that allow the interpreter to read new code syntaxes, as well as other upgrades such as performance boosts. Usually, each new interpreter version is *backwards compatible*, meaning code that executed without issue in the previous version also executes without issue and with the same behavior in the new version [10]. Backwards compatibility is highly valued due to the high usage of the language, and the PSF usually attempts to maintain backwards compatibility as much as reasonably possible.

However, in December 2008, Python version 3.0 was released. According to the PSF, Python 3.0 was the first ever "intentionally backwards incompatible" release of Python [13]. Python 3.0 included numerous syntax changes that meant that the Python 3.0 interpreter was unable to run most code written in Python 2, since it could not recognize the old syntax. According to Guido van Rossum, the creator of Python and head of its development at the time, the purpose of such wide-reaching changes was, "fixing well-known annoyances and warts, and removing a lot of old cruft" [13].

To assist in the massive task of converting entire codebases from Python 2 to Python 3, a program known as 2to3 was created by the PSF [5]. 2to3 is a *transpiler* — a program that translates one programming language to another — that converts Python 2 code to Python 3 code. However, 2to3 did not prove as popular as hoped, and the prevailing conversion strategy eventually became to use tools to delicately make a single codebase that could run under Python 2 and Python 3 [10]. The overall transition from Python 2 to Python 3 has also been rocky, as many Python developers even a decade after the transition began are still attempting to maintain the balance between Python 2 and 3 [10]. However, all versions of Python 2 have lost

Python 2	Python 3
`print "Hello World"`	`print("Hello World")`
`exec "print (1 + 1)"`	`exec("print (1 + 1)")`
`u"Hello World"`	`"Hello World"`

Table 1. A few examples of changes from Python 2 to 3, from differences in the syntax of statements for printing out text to an output and executing Python code within a Python program, to differences in how text itself is stored differently between both versions.

support from the PSF, meaning they will not receive any feature updates, or, more critically, any security updates, and all focus is now on maintaining Python 3 [6]. Thus, the importance of converting the remaining Python 2 codebases is great.

2.2 Code Translation

The problem of code translation has been approached in a number of different ways. One way has been to use techniques from the field of natural language machine translation. One 2014 study used the technique of statistical machine translation — translating language based on the probability that a given translation is a correct translation of the original language — to translate from the C# programming language to the Java programming language [8]. According to them, 68% of the code produced by the translator was 'semantically equivalent,' meaning that the translated code was essentially the same as the original code.

Python 2 to 3 translation specifically has not been a widely researched topic within the field of code translation research. However, the one study that has been done on the topic also used the technique of statistical machine translation to translate from Python 2 to Python 3. Researchers from the University of Alberta created the translator and evaluated its effectiveness using a metric called BLEU, which is a metric used by natural language machine translation researchers that is based on the premise that if a given translation is highly similar to known 'correct' translations, then the translation is likely to be correct [1]. Based on this metric, they concluded that their translator was very effective. Yet, they also acknowledged that since Python 2 and 3 are still very similar despite their differences, they already achieve high BLEU scores even without translation.

However, a 2020 research paper from Facebook Research (now Meta Research) that used neural machine translation — translating languages using neural networks — to translate between various programming languages criticized the use of BLEU as a metric for evaluating the effectiveness of code translators, given that a translation that is missing a 'word' or even a single character may still be highly similar to a correct translation, but still give the wrong result or fail to run at all due to the need for code to be highly precise in a much more extreme way than natural languages do [11]. They measured the effectiveness of their translator by creating their own metric known as computational accuracy, which is the percentage of translated code that gives the same output when executed as the original code does when executed. Their translator had widely varying success rates depending on the source and target language used. The most successful pairing source and target languages was Java to C++, and the least successful one was Python to Java [11].

2.3 Formal Verification

Since software of more than trivial complexity started being written, there has been a need for software to be verified to be free of bugs. One of the first models proposed for such verification was Hoare logic, first proposed in 1969 by the British mathematician C.A.R. Hoare. The basic statement in Hoare logic is the Hoare triple, which consists of a command and a precondition and postcondition. If the precondition is met before a command is run, then the command is run, and afterwards the postcondition is met, then the command is considered 'correct' when run with the specified conditions [7]. Hoare logic has been highly influential, and has spawned the field of formal verification.

Formal verification is nowadays most common in safety-critical industries where verification that the behavior software exhibits matches a specification of what it should do is critical. For example, formal verification is used in the transport industry to ensure that the software running on airplanes and trains exactly matches what engineers believe it should do [14]. However, even fields that do not have a critical need for safety have started to adopt formal verification, including the field of compilers. A *compiler* is a type of code translator that translates a programming language that a human programmer can understand into machine code that a computer can recognize and run. Researchers from Inria, a French government-supported computer science research institute, created CompCert, a formally verified compiler for the C programming language [9]. Compiling programming languages to machine code is a complex task that can result in notoriously difficult to catch bugs that only occur in certain conditions, so the goal of the CompCert project is to verify that all code is compiled correctly with no bugs. CompCert was able to achieve similar performance to mainstream C compilers despite needing more computations than a normal compiler would need due to the need to verify the correctness of compilations, showing that formally verified compilers are feasible to implement. CompCert was so successful that Airbus eventually used CompCert to compile the code that ran on their airplanes, since knowing that the code an engineer writes will always be correctly translated into machine code is a major component in being sure that the code running on airplanes is safe [12].

Research into formal verification of code translators is currently limited outside of CompCert and a project about the formal verification of parts of the LLVM compiler framework from researchers at the University of Pennsylvania [9][15]. There is currently no research about formally verified *transpilers* specifically. This research aims to fill that gap. A Python 2 to Python 3 transpiler was chosen as the research target due to the overall similarity of both versions, despite their large differences, making implementation easier. Such research will hopefully lead to even more research into the application of formally verified transpilers and of formally verified software in general, and hopefully ultimately result in the greater proliferation of formally

verified software that can be verified to be free of bugs. The aim of this create experimental research is to answer the questions: to what extent is a formally verified Python 2 to 3 transpiler accurate compared with 2to3, to what extent is a formally verified Python 2 to 3 transpiler faster or slower compared with 2to3, and to what extent is formal verification of a Python 2 to 3 transpiler feasible? The hypothesis for this question was that a formally verified Python 2 to 3 transpiler would be more accurate than 2to3, faster than 2to3, and would be feasible to implement.

3 Methodology

3.1 Transpiler Creation

To create a formally verified Python 2 to 3 transpiler, the Coq theorem prover was used, due to its use in previous research into formally verified code translators [9][15]. Coq is a theorem prover, which is a type of software that allows for mathematical theorems to be created and proven using a computer. Coq is a multipurpose theorem prover which originally was used in the development of formal mathematical proofs, but its use has since expanded to general purpose formal verification of software. In addition to mathematical concepts, general-purpose programs can also be written in Coq as well as specifications about what such programs should do [3]. These specifications are treated the exact same as mathematical theorems in the Coq theorem prover, and just as a theorem requires several steps before it can be proven, so does a specification about a program. Coq is an interactive theorem prover; when proving either a theorem or a specification in Coq, successive steps (for example, rewrite one side of an equation or evaluate some expression) are issued one-by-one until the theorem or specification is proven. The benefit of an automated theorem prover such as Coq comes from its ability to do tedious computations that can be trusted to be correct, thus reducing the risk of a tiny error ruining a large proof, which has happened many times for theorems which have large proofs.

Originally, the research goal was to create a formally verified transpiler that could perfectly replicate the functionality of 2to3, meaning that the formally verified translator would include every *fixer* — individual rules in 2to3 that translate one small syntactical change from 2 to 3 — that 2to3 does. Due to time constraints, that proved infeasible, and instead the research goal was narrowed to only creating a formally verified transpiler for 2to3's `print` fixer. In other words, the formally verified transpiler, dubbed "fix-print," only translated *print statements* — code statements which print text on a screen and can be used to output text to other outputs — from Python 2 to 3.

First, a specification, shown in Table 2, for how print statements should be translated was developed based on the Python 2to3 documentation [5]. Then, a *function* — a specifically defined piece of code that receives an input and returns an output — was written in Coq that took in Python 2 print statements and translated them to their Python 3 equivalents. After that, theorems each describing one part of the specification were entered into Coq. For example, one specification theorem was that for all print statements, fix-print's translated code would still begin with the word "print." After each specification theorem was defined, they were then proven in Coq using various strategies.

3.1.1 Code Extraction.

Due to Coq's nature as a theorem prover rather than a dedicated programming language, executing the transpiler in Coq directly was difficult. To remedy this, Coq's code extraction capabilities were used. Coq's code extraction capabilities allow for programs written in Coq to be translated into fully-fledged programming languages where they can be executed more easily [4]. Proofs of program correctness are not extraction in this process, only the programs themselves. However, if a program written in Coq is formally verified to meet specifications, the resulting extracted code should also meet specifications despite not being written in Coq. fix-print was extracted to the OCaml programming language in order to be run, which is also how the CompCert compiler was able to be run despite also being written in Coq [9].

Python 2	Python 3
`print "Hello World"`	`print("Hello World")`
`print "Hello World",`	`print("Hello World", end='')`
`print >>x, "Hello World"`	`print("Hello World", file=x)`

Table 2. Specification of how print statements should be translated from Python 2 to 3. For all text inputs to print, they should be surrounded with parentheses in Python 3. If the Python 2 print statement has a trailing comma (indicating that the text input should be outputted without switching to a new line, which is the default behavior), then the Python 3 translation should include `end=' '` within the parentheses. If the Python 2 print statement has two greater than signs followed by a name before the text input (indicating that the text input should be outputted to a file with that name rather than the screen), then the Python 3 translation should include `file=<name of file>` within the parentheses.

3.2 Evaluation

fix-print was evaluated against the 2to3 test suite, which is a collection of tests the developers of 2to3 wrote to test the effectiveness of 2to3. Specifically, fix-print was tested against the 16 tests that were written to test the effectiveness of the `print` fixer, which are listed in Appendix A. For each test, data about fix-print's and 2to3's accuracy and run time performance was collected and stored in CSV files in order to compare the effectiveness of both translators. Accuracy and performance data for each translator is recorded in Appendix B.

Testing was done on a desktop PC running the Arch Linux operating system with a bash terminal. Each test consisted of one line containing one or more print statements, which were each stored in their own individual files. A helper script found in Appendix D written in Python was used to aid collection of accuracy and performance data for both translators.

3.2.1 Accuracy. The metric used for evaluating the accuracy of both translators was *computational accuracy*, a metric devised by Roziere et al. in their evaluation of their AI-based code translator [11]. Translator output code was considered computationally accurate if it outputted the same result as the original input code. Computational accuracy was chosen as the method of evaluation over bilingual evaluation understudy (BLEU), a method of evaluating the accuracy of natural language machine translators that has also been used to evaluate the effectiveness of machine code translators. BLEU is a measure of the similarity of a translator's output to 'correct' reference translations of the same input, with a higher BLEU score indicating higher similarity to the correct translations. Although BLEU has been used the metric by which past code translation research have evaluated their translators, it is unsuitable as a metric to evaluate code translators due to the fact that small differences between output code and correct code, such as typos or lack of one or two characters, may result in code that produces different output or which does not run at all, yet despite that would still attain high BLEU scores due to its similarity with the correct reference code [11]. As such, computational accuracy is a preferable metric to measure the effectiveness of a code translator.

To evaluate both translators' computational accuracy, the helper script first ran Python 2 on each test to determine the intended output of each test. Then, the helper script ran 2to3 and fix-print on each test and recorded both the translated code and what the translated code outputted. If the translated code's output was identical to the Python 2 output, the translator was marked as computationally accurate for that test. Both translators' translated code, code outputs, and accuracy for each test are recorded in Appendix B.1.

3.2.2 Performance. To evaluate both translators' run time performance, the helper script measured how long each translator took to translate each test five times. The mean of those times was then taken to reduce the effect of outliers influencing the run time. To time each translator, the `time` utility of the `bash` command shell which comes preinstalled with each Arch Linux installation was used. All performance testing was conducted on the same machine in the same sitting to reduce outside factors such as how long the machine had been running or what other programs were running on the machine influencing the run time of each translator. Performance data is recorded in Appendix B.2.

4 Results

4.1 Accuracy

Translator	Overall	Without Trailing Comma Tests	Without Trailing Comma and Multiple Print Statements Tests
2to3	0.69	0.92	0.90
fix-print	0.62	0.83	1.00

Table 3. Computational accuracy of 2to3 and fix-print.

Overall, the findings suggest that 2to3 outperforms fix-print in terms of computational accuracy, not supporting my original hypothesis that a formally verified Python 2 to 3 transpiler would be more accurate than 2to3, although the two translators proved fairly close in accuracy, as seen in Table 3. However, when excluding tests with trailing commas, the accuracy of both translators increased, and when further excluding tests with multiple print statements, fix-print attained a higher accuracy than 2to3, supporting my original hypothesis. The rationale for both of these exclusions is as follows.

During testing, it was discovered that the version of Python 2 (the `python2-bin` package in the Arch User Repository (AUR), a collection of software packages intended for users of the Arch Linux operating system, the platform used during testing) used in testing was bugged. Trailing commas at the end of print statements should result in a newline character — a special character that tells the computer to move to a new line, similar in effect to hitting enter on a keyboard — not being printed. The Python 2 documentation states that is the intended behavior, and therefore both 2to3 and fix-print attempt to replicate

that when translating. However, the version of Python 2 used outputted a newline character even when a trailing comma was present, contrary to its documented behavior, thus rendering otherwise accurate translations from both translators inaccurate. A version of Python 2 obtained directly from the official Python website was also tested, however that version exhibited the same incorrect behavior as the version from the AUR, thus it was not used for data collection. Therefore, for more accurate results, the accuracy of both translators excluding tests with trailing commas was calculated as well.

Excluding the trailing comma tests, 2to3 was still more accurate than fix-print. However, there were two tests that were unique within the test suite, as they contained multiple print statements instead of only a single one, as every other test did. As fix-print was only specified to translate single print statements, it naturally produced incorrect translations for these two tests. Excluding these two tests, fix-print attained a higher accuracy than 2to3, supporting my original hypothesis, due to the fact that 2to3 failed the `test-idempotency-1` test where fix-print did not, as seen in Tables 5 and 6. In Python 2, `print ()` outputs `()`, as the empty parenthesis represent an empty tuple, which is a special type of list in Python. To replicate this behavior, the correct Python 3 translation should be `print(())`, as Python 3 print statements require an extra pair of parentheses around whatever content is to be outputted. fix-print is specified to always do as such, however in this case 2to3 translated `print ()` as `print()`, which is incorrect as that translation simply prints nothing, as there is no content within the parentheses.

2to3's failure to translate the `test-idempotency-1` test correctly is intentional, as translating with 2to3 is intended to be *idempotent*, meaning that if Python 2 code is fed into 2to3, producing Python 3 code, then feeding that code back into 2to3 should result in no changes. This is important as it allows for consistency and reproducibility of translations. If 2to3 always surrounded the content to be printed in print statements as fix-print is specified to do, then it would not be idempotent as it would always add parentheses in print statements, even for Python 3 code it already translated. This is a clear example of how formal verification is not a panacea for all code issues. Formal verification that all content in print statements would be surrounded by parentheses did specify that all resulting translations would be correct. However it also made the translator nonidempotent, which is also a good quality to have, especially in situations in large codebases where some code may be translated and some may not be, and it is desired that translated code is not further modified. Thus, any formal verification of a program is only as good as its specification. If the specification is faulty or implies an undesirable effect (such as nonidempotency), then the verified program will contain such flaws despite being 'mathematically proven to be correct.' In real codebases, it is unlikely that `print ()` is ever used as it has no useful effect, and 2to3 translates virtually every other print statement accurately, thus in real scenarios where a Python 2 to 3 translator is needed, 2to3's emphasis on idempotency at the cost of a few edge cases being inaccurately translated is likely preferable to fix-print's guaranteed correctness at the cost of changing already translated files in a codebase each time a new translation is run.

4.2 Performance

fix-print was faster than 2to3 at translation, with fix-print translating code around two orders of magnitude faster than 2to3. This supports the hypothesis that a formally verified transpiler would perform better than a non-formally verified compiler. However, it is of vital importance to reemphasize the fact that both transpilers are written in two different programming languages. 2to3 was written in Python, whereas fix-print is written in Coq, with code extracted to OCaml. Due to the fundamental nature of both languages, programs written in OCaml will nearly always be faster than similar programs written in Python, due to the fact that Python programs are relatively slow due to the nature of how the Python language executes programs compared to other languages. Thus, a conclusion that formal verification leads to a performance gain is unwarranted, since the languages that both transpilers were written in affected performance much more than whether formal verification was used or not. However, the data suggests that formal verification is not a significant drag on performance. This lends credibility to advocates of formal verification, as possible performance regressions that might come along with the adoption of formal verification seem to not occur.

4.3 Feasibility

Formal verification added a considerable time cost to the development of a transpiler compared to if formal verification was not used. Development of fix-print took around three weeks for only a single 2to3 fixer, of which there are 52 in total. A similar transpiler developed without formal verification would have taken at most half of that time in the researcher's estimation. For programmers more experienced in formal verification, formal verification of such a project would likely take a lesser proportion of time; one programmer who worked on a formally verification project estimated that it took 10% more effort than if it were done without formal verification [14]. Formal verification of a Python 2 to 3 transpiler is likely technically feasible, as shown in this project, and it did certainly show effectiveness in catching edge cases. However, unlike in the transportation or compiler fields, Python 2 to 3 translation is not very safety critical. Unlike compilers which translate human-readable source code to computer-readable but human-unreadable, Python 2 to 3 translations have human-readable inputs and outputs, resulting in comparatively easy human verification of translations compared with compiled code. Thus, a major impetus

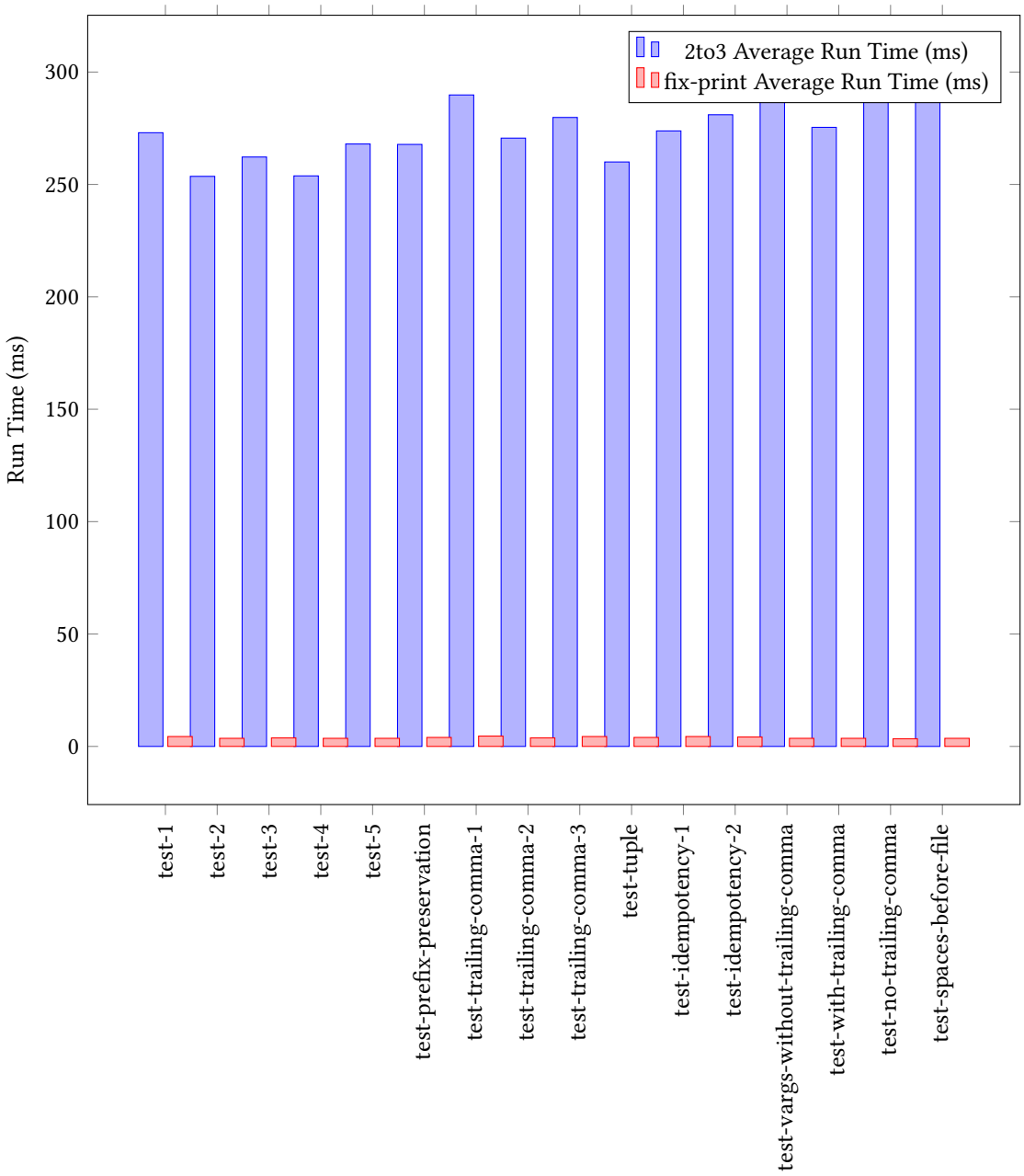

Figure 1. Mean run times of both 2to3 and fix-print for each test.

for formally verifying code translators – verifying that code is translated perfectly correctly – is significantly reduced for Python translations as human verification is comparatively feasible, reducing the need for formal verification. Thus, although a formally verified Python 2 to 3 transpiler is technically feasible, given the additional time investment versus the comparatively low benefit from formal verification, formal verification is likely not the overall best approach to create a Python 2 to 3 transpiler. That being said, even for this relatively trivial project, formal verification was able to fix an edge case, showing that it has great potential to be useful in other applications.

5 Conclusion

Virtually all software in use today has bugs, many of which lie dormant until conditions line up perfectly for those bugs to manifest themselves. This is the case for 2to3, which translates some rarely used code structures inaccurately. With formal

verification, the translation of print statements was made mathematically verifiably conformant to a specification of how they should be translated, resulting in a more accurate translator that performed better. However, perfect accuracy comes at the cost of harder maintainability of translated code, showing that formal verification is not a silver bullet for all software bugs. Despite that, formal verification has shown itself useful in identifying and correcting hidden bugs, and greater use of formal verification is likely to uncover even more bugs in all sorts of software. Formal verification has seen ever wider use in software development, and this project has shown that formal verification can bring benefits even outside of safety-critical applications. Code translators of all sorts, from compilers to transpilers, can all benefit from formal verification, especially when translations involve many minute details that all need to be correct. Code translators are perfect candidates for expanded use of formal verification, yet they are only one type of software out of many which can benefit from formal verification.

5.1 Limitations

5.1.1 Validity. The validity of fix-print rests on the validity of Coq and its code extraction capabilities. Should Coq or its code extraction capabilities prove flawed, the validity of fix-print would also be flawed. However, given Coq's extensive history and use in both mathematics and computer science, such an outcome is unlikely.

5.1.2 Scope. Due to time limitations, fix-print only covered part of what 2to3 currently does. fix-print only covers 1 out of 2to3's 52 fixers, and so there is much that needs to be added to make fix-print a truly viable alternative to 2to3. In addition, the parsing stage of fix-print, where the unstructured text of code is turned into structured data for the translator to actually translate, was not formally verified, so fix-print is not yet completely formally verified from end to end. However, this is common in formal verification projects, where the most important logic of a program is formally verified first and peripheral items are verified later.

5.1.3 Tests. fix-print was only tested for computational accuracy against 2to3's test suite, which only includes 16 tests. Although the tests try to model the most common print statement structures found in real code, with such a small number of tests there is likely to be some sort of uncommon structure that is missed in the tests. Since fix-print was not tested on any actual codebases in current use, research on fix-print's effectiveness in real codebases should be done to further test its real-world accuracy and performance.

5.2 Future Research

5.2.1 Other Verification Languages. Formal verification using the Coq proof assistant was chosen as the method of formal verification for this project due its use in previous research into formally verified code translators [9][15]. As a theorem prover, Coq is based in math as much as it is based in programming, and Coq is frequently used to proven pure mathematical theorems as well as theorems about the correctness of software programs. However, there are other methods of formal verification possible. One example is the Java Modeling Language (JML), an addition to the popular Java programming language that allows Java developers to define specifications for their existing Java code in Java itself [2]. This is a simpler approach to formal verification for Java developers, since it allows them to write both their code and their specifications in one language, especially important for those working on large Java codebases where switching programming languages for formal verification is infeasible. Another possible approach is Dafny, a programming language designed for formal verification. Dafny is structurally much more similar to traditional programming languages than Coq is, but allows for specification of what code should do. Both JML and Dafny are more accessible than Coq for most programmers, thus building formally verified code translators with such tooling may be more feasible than building one with Coq, thus making it a possible area of future research.

5.2.2 Translating Different Languages. Another area of possible future research is in the formal verification of translators between two vastly different programming languages. The need for CompCert (the formally verified C compiler) arose from the fact that C code and machine code are extremely different and machine code is largely not readable for humans, meaning it is difficult for a person to verify that all code was translated perfectly according to specification, thus requiring the help of formal verification to ensure translations are perfectly conformant to specifications. While Python 2 and 3 have quite a few differences, at the end of the day they are still extremely similar and human verification that translations from Python 2 to 3 are correct is very feasible. Translating between programming languages with significant differences is harder to hand-verify, and thus is a promising area for future research into formally verified code translators.

References

[1] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. 2015. Using machine translation for converting python 2 to python 3 code, (Oct. 2015). DOI: 10.7287/PEERJ.PREPRINTS.1459V1.

[2] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, Rustan Leino, and Erik Poll. 2005. An overview of jml tools and applications. *International Journal on Software Tools for Technology Transfer*, 7, 3, (June 2005), 212–232. https://www.microsoft.com/en-us/research/publication/overview-jml-tools-applications/.

[3] Coq. [n. d.] What is coq? https://coq.inria.fr/about-coq.

[4] Jean-Christophe Filliâtre and Pierre Letouzey. [n. d.] *Program extraction*. https://coq.inria.fr/doc/v8.13/refman/addendum/extraction.html.

[5] Python Software Foundation. [n. d.] *2to3 — Automated Python 2 to 3 code translation*. https://docs.python.org/3/library/2to3.html.

[6] Python Software Foundation. [n. d.] Sunsetting python 2. https://www.python.org/doc/sunset-python-2/.

[7] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM*, 12, 10, (Oct. 1969), 576–580. DOI: 10.1145/363235.363259.

[8] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Onward! 2014). Association for Computing Machinery, Portland, Oregon, USA, 173–184. ISBN: 9781450332101. DOI: 10.1145/2661136.2661148.

[9] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM*, 52, 7, (July 2009), 107–115. DOI: 10.1145/1538788.1538814.

[10] Brian A. Malloy and James F. Power. 2017. Quantifying the transition from python 2 to 3: an empirical study of python applications. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (ESEM '17). IEEE Press, Markham, Ontario, Canada, 314–323. ISBN: 9781509040391. DOI: 10.1109/ESEM.2017.45.

[11] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (NIPS'20) Article 1730. Curran Associates Inc., Vancouver, BC, Canada, 11 pages. ISBN: 9781713829546.

[12] J. Souyris. 2014. Industrial use of compcert on a safety-critical software product. (2014). https://projects.laas.fr/IFSE/FMF/J3/slides/P05_Jean_Souyiris.pdf.

[13] Guido van Rossum. [n. d.] What's new in python 3.0. https://docs.python.org/3/whatsnew/3.0.html.

[14] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. 2009. Formal methods: practice and experience. *ACM Comput. Surv.*, 41, 4, Article 19, (Oct. 2009), 36 pages. DOI: 10.1145/1592434.1592436.

[15] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the llvm intermediate representation for verified program transformations. *SIGPLAN Not.*, 47, 1, (Jan. 2012), 427–440. DOI: 10.1145/2103621.2103709.

A Tests

Test	Code	Python 2 Output
test-1	`print 1, 1+1, 1+1+1`	`b'1 2 3\n'`
test-2	`print 1, 2`	`b'1 2\n'`
test-3	`print`	`b'\n'`
test-4	`print 1; print`	`b'1\n\n'`
test-5	`print; print 1;`	`b'\n1\n'`
test-prefix-preservation	`print 1,  1+1,  1+1+1`	`b'1 2 3\n'`
test-trailing-comma-1	`print 1, 2, 3,`	`b'1 2 3\n'`
test-trailing-comma-2	`print 1, 2,`	`b'1 2\n'`
test-trailing-comma-3	`print 1,`	`b'1\n'`
test-tuple	`print (1, 2, 3)`	`b'(1, 2, 3)\n'`
test-idempotency-1	`print()`	`b'()\n'`
test-idempotency-2	`print('')`	`b'\n'`
test-vargs-without-trailing-comma	`print >>sys.stderr, 1, 2, 3`	`b'1 2 3\n'`
test-with-trailing-comma	`print >>sys.stderr, 1, 2,`	`b'1 2'`
test-no-trailing-comma	`print >>sys.stderr, 1+1`	`b'2\n'`
test-spaces-before-file	`print >> sys.stderr`	`b'\n'`

Table 4. Tests included in the 2to3 test suite, with the Python 2 code to be translated and intended output of that code.

B Data

B.1 Computational Accuracy

Test	2to3 Translation	2to3 Output	2to3 Output Correct
test-1	`print(1, 1+1, 1+1+1)`	`b'1 2 3\n'`	True
test-2	`print(1, 2)`	`b'1 2\n'`	True
test-3	`print()`	`b'\n'`	True
test-4	`print(1); print()`	`b'1\n\n'`	True
test-5	`print(); print(1);`	`b'\n1\n'`	True
test-prefix-preservation	`print(1,  1+1,  1+1+1)`	`b'1 2 3\n'`	True
test-trailing-comma-1	`print(1, 2, 3, end='')`	`b'1 2 3 '`	False
test-trailing-comma-2	`print(1, 2, end='')`	`b'1 2 '`	False
test-trailing-comma-3	`print(1, end='')`	`b'1 '`	False
test-tuple	`print((1, 2, 3))`	`b'(1, 2, 3)\n'`	True
test-idempotency-1	`print()`	`b'\n'`	False
test-idempotency-2	`print('')`	`b'\n'`	True
test-vargs-without-trailing-comma	`print(1, 2, 3, file=sys.stderr)`	`b'1 2 3\n'`	True
test-with-trailing-comma	`print(1, 2, end='', file=sys.stderr)`	`b'1 2 '`	False
test-no-trailing-comma	`print(1+1, file=sys.stderr)`	`b'2\n'`	True
test-spaces-before-file	`print(file=sys.stderr)`	`b'\n'`	True

Table 5. 2to3 translation and output of translated code for each test.

Test	fix-print Translation	fix-print Output	fix-print Output Correct
test-1	`print(1,1+1,1+1+1)`	`b'1 2 3\n'`	True
test-2	`print(1,2)`	`b'1 2\n'`	True
test-3	`print()`	`b'\n'`	True
test-4	`print(1;print)`	`b''`	False
test-5	`print(;print1;)`	`b''`	False
test-prefix-preservation	`print(1,1+1,1+1+1)`	`b'1 2 3\n'`	True
test-trailing-comma-1	`print(1,2,3,end='')`	`b'1 2 3 '`	False
test-trailing-comma-2	`print(1,2,end='')`	`b'1 2 '`	False
test-trailing-comma-3	`print(1,end='')`	`b'1 '`	False
test-tuple	`print((1,2,3))`	`b'(1, 2, 3)\n'`	True
test-idempotency-1	`print(())`	`b'()\n'`	True
test-idempotency-2	`print(('',))`	`b'\n'`	True
test-vargs-without-trailing-comma	`print(1,2,3,file=sys.stderr)`	`b'1 2 3\n'`	True
test-with-trailing-comma	`print(1,2,end='',file=sys.stderr)`	`b'1 2 '`	False
test-no-trailing-comma	`print(1+1,file=sys.stderr)`	`b'2\n'`	True
test-spaces-before-file	`print(file=sys.stderr)`	`b'\n'`	True

Table 6. fix-print translation and output of translated code for each test.

B.2 Performance

Test	2to3 Run Time 1 (ms)	2to3 Run Time 2 (ms)	2to3 Run Time 3 (ms)	2to3 Run Time 4 (ms)	2to3 Run Time 5 (ms)	2to3 Mean Run Time (ms)
test-1	274	307	274	255	255	273
test-2	246	249	269	253	251	253.6
test-3	256	261	248	268	278	262.2
test-4	249	252	261	250	257	253.8
test-5	282	265	255	263	275	268
test-prefix-preservation	287	285	265	255	247	267.8
test-trailing-comma-1	267	263	319	307	293	289.8
test-trailing-comma-2	282	262	252	279	278	270.6
test-trailing-comma-3	291	252	267	300	289	279.8
test-tuple	255	252	271	269	253	260
test-idempotency-1	262	276	273	274	284	273.8
test-idempotency-2	316	265	273	260	291	281
test-vargs-without-trailing-comma	266	279	331	275	329	296
test-with-trailing-comma	295	265	281	282	254	275.4
test-no-trailing-comma	255	260	314	337	285	290.2
test-spaces-before-file	268	329	257	276	322	290.4

Table 7. Run times of 2to3 for each test and their means.

Test	fix-print Run Time 1 (ms)	fix-print Run Time 2 (ms)	fix-print Run Time 3 (ms)	fix-print Run Time 4 (ms)	fix-print Run Time 5 (ms)	fix-print Mean Run Time (ms)
test-1	3	5	4	5	5	4.4
test-2	3	3	3	4	5	3.6
test-3	3	4	3	4	5	3.8
test-4	3	4	3	4	4	3.6
test-5	4	4	3	4	3	3.6
test-prefix-preservation	3	4	4	4	5	4
test-trailing-comma-1	5	5	4	4	5	4.6
test-trailing-comma-2	4	4	5	3	3	3.8
test-trailing-comma-3	4	4	5	4	5	4.4
test-tuple	3	5	3	3	6	4
test-idempotency-1	6	5	3	5	3	4.4
test-idempotency-2	3	3	6	4	5	4.2
test-vargs-without-trailing-comma	4	4	3	4	3	3.6
test-with-trailing-comma	3	3	5	4	3	3.6
test-no-trailing-comma	3	4	3	3	4	3.4
test-spaces-before-file	3	3	4	4	4	3.6

Table 8. Run times of fix-print for each test and their means.

C fix-print Source Code

```
Require Import String.
Require Import List.
Import ListNotations.
Require Import Coq.Program.Wf.
Require Import Extraction.
Require Import ExtrOcamlBasic.
Require Import ExtrOcamlNativeString.

Inductive leaf : Type :=
| ENDMARKER : leaf
| NAME : string -> leaf
| NUMBER : string -> leaf
| STRING : string -> leaf
| NEWLINE : leaf
| INDENT : leaf
| DEDENT : leaf
| LPAR : leaf
| RPAR : leaf
| LSQB : leaf
| RSQB : leaf
| COLON : leaf
| COMMA : leaf
| SEMI : leaf
| PLUS : leaf
| MINUS : leaf
| STAR : leaf
| SLASH : leaf
| VBAR : leaf
| AMPER : leaf
| LESS : leaf
| GREATER : leaf
| EQUAL : leaf
| DOT : leaf
| PERCENT : leaf
| BACKQUOTE : leaf
| LBRACE : leaf
| RBRACE : leaf
| EQEUQAL : leaf
| NOTEQUAL : leaf
| LESSEQUAL : leaf
| GREATEREQUAL : leaf
| TILDE : leaf
| CIRCUMFLEX : leaf
| LEFTSHIFT : leaf
| RIGHTSHIFT : leaf
| DOUBLESTAR : leaf
| PLUSEQUAL : leaf
| MINEQUAL : leaf
| STAREQUAL : leaf
| SLASHEQUAL : leaf
| PERCENTEQUAL : leaf
| AMPEREQUAL : leaf
```

```
| VBAREQUAL : leaf
| CIRCUMFLEXEQUAL : leaf
| LEFTSHIFTEQUAL : leaf
| RIGHTSHIFTEQUAL : leaf
| DOUBLESTAREQUAL : leaf
| DOUBLESLASH : leaf
| DOUBLESLASHEQUAL : leaf
| AT : leaf
| ATEQUAL : leaf
| OP : leaf
| COMMENT : string -> leaf
| NL : leaf
| RARROW : leaf
| AWAIT : leaf
| ASYNC : leaf
| ERRORTOKEN : leaf
| COLONEQUAL : leaf.

Inductive tree : Type :=
| LEAF : leaf -> tree
| NODE : list tree -> tree.

Fixpoint concat_strings (l : list string) : string :=
match l with
| [] => ""
| x :: xs => x ++ concat_strings xs
end.

Definition to_string (t : tree) : string :=
match t with
| LEAF (NAME n) => n
| LEAF (NUMBER n) => n
| LEAF (STRING s) => "'" ++ s ++ "'"
| LEAF LPAR => "("
| LEAF RPAR => ")"
| LEAF COLON => ":"
| LEAF COMMA => ","
| LEAF SEMI => ";"
| LEAF PLUS => "+"
| LEAF MINUS => "-"
| LEAF STAR => "*"
| LEAF SLASH => "/"
| LEAF AMPER => "&"
| LEAF LESS => "<"
| LEAF GREATER => ">"
| LEAF EQUAL => "="
| LEAF DOT => "."
| LEAF PERCENT => "%"
| LEAF BACKQUOTE => "`"
| LEAF LBRACE => "{"
| LEAF RBRACE => "}"
| LEAF EQEUQAL => "=="
| LEAF NOTEQUAL => "!="
| LEAF LESSEQUAL => "<="
```

```
| LEAF GREATEREQUAL => ">="
| LEAF TILDE => "~"
| LEAF CIRCUMFLEX => "^"
| LEAF LEFTSHIFT => "<<"
| LEAF RIGHTSHIFT => ">>"
| LEAF DOUBLESTAR => "**"
| LEAF PLUSEQUAL => "+="
| LEAF MINEQUAL => "-="
| LEAF STAREQUAL => "*="
| LEAF SLASHEQUAL => "/="
| LEAF PERCENTEQUAL => "%="
| LEAF AMPEREQUAL => "&="
| LEAF VBAREQUAL => "|="
| LEAF CIRCUMFLEXEQUAL => "^="
| LEAF LEFTSHIFTEQUAL => "<<="
| LEAF RIGHTSHIFTEQUAL => ">>="
| LEAF DOUBLESTAREQUAL => "**="
| LEAF DOUBLESLASH => "//"
| LEAF DOUBLESLASHEQUAL => "//="
| LEAF AT => "@"
| LEAF ATEQUAL => "@="
| LEAF COLONEQUAL => ":="
| _ => ""
end.

Definition fix_print (l : list tree) : list tree :=
match l with
| LEAF (NAME "print") :: n => LEAF (NAME "print") :: LEAF LPAR ::
    (match n with
    | LEAF RIGHTSHIFT :: LEAF (NAME f) :: n' =>
    match rev n' with
  | LEAF COMMA :: n'' => rev n'' ++  [LEAF COMMA ; LEAF (NAME "end") ; LEAF EQUAL ; LEAF (STRING " ") ; LEAF COMMA;
        LEAF (NAME "file") ; LEAF EQUAL ; LEAF (NAME f)]
    | _ => match n' with
          | LEAF COMMA :: n'' => n' ++ [LEAF COMMA; LEAF (NAME "file") ; LEAF EQUAL ; LEAF (NAME f)]
            | _ => n' ++ [LEAF (NAME "file") ; LEAF EQUAL ; LEAF (NAME f)]
            end
    end
    | _ => match rev n with
   | LEAF COMMA :: n' => rev n' ++ [LEAF COMMA ; LEAF (NAME "end") ; LEAF EQUAL ; LEAF (STRING " ")]
    | _ => n
    end
    end) ++ [LEAF RPAR]
| _ => [LEAF (NAME "print") ; LEAF LPAR ; LEAF RPAR]
end.

Lemma output_starts_with_print : forall n : list tree,
    nth 0 (fix_print (LEAF (NAME "print") :: n)) (NODE []) = LEAF (NAME "print").
destruct n.
- simpl. reflexivity.
- simpl. reflexivity.
Qed.

Lemma output_begins_with_parentheses : forall n : list tree,
```

```
    nth 1 ((fix_print (LEAF (NAME "print") :: n))) (NODE []) = LEAF LPAR.
destruct n ; auto.
Qed.

Extraction "/home/user/fix-print/src/tests/fix-print.ml" fix_print.
```

D Helper Script

```
import subprocess

datafile = open("data.csv", "w")
print(
    "Test | Python 2 Output | 2to3 Translation | 2to3 Output | 2to3 Output Correct | 2to3 Run Time 1
    file=datafile,
)

tests = [
    "test_1",
    "test_2",
    "test_3",
    "test_4",
    "test_5",
    "test_prefix_preservation",
    "test_trailing_comma_1",
    "test_trailing_comma_2",
    "test_trailing_comma_3",
    "test_tuple",
    "test_idempotency_1",
    "test_idempotency_2",
    "test_vargs_without_trailing_comma",
    "test_with_trailing_comma",
    "test_no_trailing_comma",
    "test_spaces_before_file",
]

regular_tests = [
    "test_1",
    "test_2",
    "test_3",
    "test_4",
    "test_5",
    "test_prefix_preservation",
    "test_trailing_comma_1",
    "test_trailing_comma_2",
    "test_trailing_comma_3",
    "test_tuple",
]

idempotency_tests = [
    "test_idempotency_1",
    "test_idempotency_2",
]

file_redirection_tests = [
    "test_vargs_without_trailing_comma",
    "test_with_trailing_comma",
    "test_no_trailing_comma",
    "test_spaces_before_file",
]
```

```
    for test in regular_tests:
        two_to_three_runtimes = []

        for j in range(5):
            python_two_output = subprocess.run(
                ["python2", test], capture_output=True
            ).stdout

            two_to_three_test = subprocess.run(
                f"time 2to3 {test}",
                shell=True,
                executable="/bin/bash",
                capture_output=True,
                encoding="utf-8",
            )

            for line in two_to_three_test.stdout.splitlines():
                if line.startswith("+"):
                    two_to_three_translation = line.lstrip("+")

            for line in two_to_three_test.stderr.splitlines():
                if line.startswith("real"):
                    two_to_three_runtimes.append(int(line[-4:-1]))

        two_to_three_average_runtime = sum(two_to_three_runtimes) / 5

        two_to_three_translation_output = subprocess.run(
            ["python", "-c", two_to_three_translation], capture_output=True
        ).stdout

        two_to_three_output_correct = python_two_output == two_to_three_translation_output

        print(
            f"{test} | {python_two_output} | {two_to_three_translation} | {two_to_three_translation_outp
            file=datafile,
        )


    for test in idempotency_tests:
        two_to_three_runtimes = []

        for j in range(5):
            python_two_output = subprocess.run(
                ["python2", test], capture_output=True
            ).stdout

            two_to_three_test = subprocess.run(
                f"time 2to3 {test}",
                shell=True,
                executable="/bin/bash",
                capture_output=True,
                encoding="utf-8",
            )
```

```
        with open(test) as test_file:
            two_to_three_translation = test_file.read()

        for line in two_to_three_test.stderr.splitlines():
            if line.startswith("real"):
                two_to_three_runtimes.append(int(line[-4:-1]))

    two_to_three_average_runtime = sum(two_to_three_runtimes) / 5

    two_to_three_translation_output = subprocess.run(
        ["python", "-c", two_to_three_translation], capture_output=True
    ).stdout

    two_to_three_output_correct = python_two_output == two_to_three_translation_output

    print(
        f"{test} | {python_two_output} | {two_to_three_translation} | {two_to_three_translation_outp
        file=datafile,
    )

for test in file_redirection_tests:
    two_to_three_runtimes = []

    for j in range(5):
        python_two_output = subprocess.run(
            ["python2", test], capture_output=True
        ).stderr

        two_to_three_test = subprocess.run(
            f"time 2to3 {test}",
            shell=True,
            executable="/bin/bash",
            capture_output=True,
            encoding="utf-8",
        )

        for line in two_to_three_test.stdout.splitlines():
            if line.startswith("+"):
                two_to_three_translation = line.lstrip("+")

        for line in two_to_three_test.stderr.splitlines():
            if line.startswith("real"):
                two_to_three_runtimes.append(int(line[-4:-1]))

    two_to_three_average_runtime = sum(two_to_three_runtimes) / 5

    two_to_three_translation_output = subprocess.run(
        ["python", "-c", "import sys;" + two_to_three_translation],
        capture_output=True,
    ).stderr

    two_to_three_output_correct = python_two_output == two_to_three_translation_output

    print(
```

```
            f"{test} | {python_two_output} | {two_to_three_translation} | {two_to_three_translation_outp
            file=datafile,
        )

    print(
        "Test | Python 2 Output | fix_print Translation | fix_print Output | fix_print Output Correct |
        file=datafile,
    )

    for fix_print_test in regular_tests + idempotency_tests:
        fix_print_runtimes = []

        for j in range(5):
            python_two_output = subprocess.run(
                ["python2", fix_print_test], capture_output=True
            ).stdout

            fix_print_test_run = subprocess.run(
                f"time ./{fix_print_test}.oc",
                shell=True,
                executable="/bin/bash",
                capture_output=True,
                encoding="utf-8",
            )

            fix_print_translation = fix_print_test_run.stdout

            for line in fix_print_test_run.stderr.splitlines():
                if line.startswith("real"):
                    fix_print_runtimes.append(int(line[-4:-1]))

        fix_print_average_runtime = sum(fix_print_runtimes) / 5

        fix_print_translation_output = subprocess.run(
            ["python", "-c", fix_print_translation], capture_output=True
        ).stdout

        fix_print_output_correct = python_two_output == fix_print_translation_output

        print(
            f"{fix_print_test} | {python_two_output} | {fix_print_translation} | {fix_print_translation_
            file=datafile,
        )

    for test in file_redirection_tests:
        fix_print_runtimes = []

        for j in range(5):
            python_two_output = subprocess.run(
                ["python2", test], capture_output=True
            ).stderr

            fix_print_test_run = subprocess.run(
                f"time ./{test}.oc",
```

```
        shell=True,
        executable="/bin/bash",
        capture_output=True,
        encoding="utf-8",
    )

    fix_print_translation = fix_print_test_run.stdout

    for line in fix_print_test_run.stderr.splitlines():
        if line.startswith("real"):
            fix_print_runtimes.append(int(line[-4:-1]))

fix_print_average_runtime = sum(fix_print_runtimes) / 5

fix_print_translation_output = subprocess.run(
    ["python", "-c", "import sys;" + fix_print_translation],
    capture_output=True,
).stderr

fix_print_output_correct = python_two_output == fix_print_translation_output

print(
    f"{test} | {python_two_output} | {fix_print_translation} | {fix_print_translation_output} |
    file=datafile,
)
```