Py2Cy: A Genetic Improvement Tool To Speed Up Python

James Zhong
University College London
London, United Kingdom
james.zhong.18@ucl.ac.uk

Max Hort
University College London
London, United Kingdom
max.hort.19@ucl.ac.uk

Federica Sarro
University College London
London, United Kingdom
f.sarro@ucl.ac.uk

ABSTRACT

Due to its ease of use and wide range of custom libraries, Python has quickly gained popularity and is used by a wide range of developers all over the world. While Python allows for fast writing of source code, the resulting programs are slow to execute when compared to programs written in other programming languages like C. One of the reasons for its slow execution time is the dynamic typing of variables. Cython is an extension to Python, which can achieve execution speed-ups by compiler optimization. One possibility for improvements is the use of static typing, which can be added to Python scripts by developers. To alleviate the need for manual effort, we create *Py2Cy*, a Genetic Improvement tool for automatically converting Python scripts to statically typed Cython scripts. To show the feasibility of improving runtime with *Py2Cy*, we optimize a Python script for generating Fibonacci numbers. The results show that *Py2Cy* is able to speed up the execution time by up to a factor of 18.

CCS CONCEPTS

• **Computing methodologies** → **Optimization algorithms**.

KEYWORDS

Genetic Improvement, Python, Performance, Execution Time

ACM Reference Format:
James Zhong, Max Hort, and Federica Sarro. 2022. *Py2Cy*: A Genetic Improvement Tool To Speed Up Python. In *Genetic and Evolutionary Computation Conference Companion (GECCO '22 Companion), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3520304.3534037

1 INTRODUCTION

Python is a high-level, general purpose programming language [43] used by a variety of developers, ranging from non-experts to advanced programmers [39]. Nowadays, Python is popular in several application domains, among others the scientific computing community [6, 31], due to its ease of use, dynamic typing, and wide range of custom libraries.

The advantage of a high-level programming language, such as Python, is an enhanced productivity which simplifies code writing and prototyping [3, 6, 31]. Dalcin et al. [12] went as far as to argue that Python "become de facto standard for computation-driven scientific research".

While Python allows for fast prototyping and writing of source code, the resulting programs often lack fast execution time [39]. Features, including dynamic typing and automatic memory management, come at the cost of a reduced performance and slow execution times [41].

```python
def factorial(x):
    y = 1
    for i in range(x):
        y *= i+1
    return y
```

Listing 1: Python Code.

```python
cdef int factorial(int x):
    cdef int y = 1
    cdef int i
    for i in range(x):
        y *= i+1
    return y
```

Listing 2: Cython Code.

Normally, Python uses CPython as its source code interpreter to transform Python code into bytecode [38]. One approach for improving the execution time of Python programs is the use of custom interpreters that are closer to the compiler language C and can therefore be executed faster than CPython [9]. Cython is a popular example of such an approach, which is designed to achieve C-like runtime performance for code written in Python with optional additional C-inspired syntax, thereby allowing for speed-ups of Python programs [45].

While Python programs can be transformed into Cython without changes to the source code, further improvements can be achieved by including Cython constructs such as static typing of variables, which can then be resolved at compile-time rather than runtime [7, 45]. An example Python script is shown in Listing 1 and its corresponding Cython version is shown in Listing 2. Both scripts are used to compute the $factorial$ of a number n with the only difference being that the Cython code specifies that the variables x, y, i as well as the output y are integers. To distinguish Cython from Python scripts, Cython scripts use the file ending *.pyx* while Python use *.py* (e.g., Listing 2 could be named *factorial.pyx*). This code modification can reduce the running time of the program yet requires the intervention of a developer to be made. For this small example, the developer manual effort is trivial, however it becomes more and more demanding for larger real-world programs.

To combine the ease of building software, which Python offers, with a fast execution time, we propose *Py2Cy*, a tool for automating the process of converting Python to Cython programs. In particular, we aim at automatically incorporating Cython specific features, such as static-typing, in the source code of Python programs. By doing so, developers can focus on writing functional Python code, which then is converted to faster Cython programs without any manual effort.

GECCO '22 Companion, July 9–13, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9268-6/22/07…$15.00
https://doi.org/10.1145/3520304.3534037

For this purpose, we use Genetic Improvement [24, 25], a Search Based Software Engineering [20] technique proposed to automatically adapt source code in order to improve its non-functional properties. In the past, GI has been used to improve software intended for a variety of tasks, including data mining, financial modeling and image processing [32]. Moreover, GI was successful in reducing the execution time of programs [26, 33].

To summarise, the main contributions of our work are:

- the first proposal to speed up Python programs running time by using GI;
- *Py2Cy*, an automated GI tool for converting Python to Cython;
- a feasibility study of the proposed tool;

The *Py2Cy* source code and our experimental data is publicly available to allow for reproduction and extension of our work: https://github.com/SOLAR-group/Py2Cy.

The rest of the paper is organized as follows. Section 2 provides the reader with an overview of the existing work in this area. Section 3 provides some background on Cython and presents our proposed tool *Py2Cy*. The experimental design and results are described in Section 4. Following, Section 5 concludes the paper and gives some recommendations for future work.

2 RELATED WORK

In this section, we present related work on approaches for speeding-up Python programs. Additionally, we outline existing work on Genetic Improvement, including techniques that have been applied to improve Python programs.

2.1 Speeding-up Python

Python is a high-level programming language that trades execution time in favour of ease to use. To combat this shortcoming, several approaches have been applied [29].

Among others, the execution time of Python programs has been reduced by the use of parallelization [14, 39]. For example, Jacob and Singer [23] proposed an automated framework to enable the parallelization of loops and achieved speed-ups over the native Cpython interpreter of Python programs.

Garg and Amaral [15] achieved speed-ups by executing selected parts of Python program on the GPU instead of running the entire program on a CPU. For numeric and scientific problems, high performance libraries, such as NumPy, SciPy and Theano can be used [21, 42, 44]. These libraries provide Python APIs which implemented functions in C or C++ [6].

Furthermore, several compilers have been proposed in the past, which provide a faster compilation time than the native CPython interpreter [2, 17, 34, 41]. In addition to the proposal of new Python interpreters, several benchmarks exist to compare their performance [2, 30, 36, 38]. PyPy [37] is a popular Python implementation written in RPython, which is statically compiled and adds features, such as garbage collection and a Just-in-Time compiler [29]. Another popular extension for Python is Cython, which allows for an extension of Python with C data types and will be the focus of our experiments. More information on Cython is provided in Section 3. Our work is the first to propose the use of Genetic Improvement to speed up the execution of Python programs as a means of automated program translation from Python to Cython.

2.2 Genetic Improvement

Genetic Improvement (GI) is a Search Based Software Engineering (SBSE) [20] technique which applies modifications to program source code to achieve improvements of non-functional and functional properties. Frequently, programs are changed by deleting or replacing existing, or inserting new source code lines [11, 26, 27]. GI has been successfully applied to large range of domains [32] to reduce programs execution time [26, 33], memory consumption [46], and energy consumption [11].

Langdon and Harman [26] applied GI to a DNA sequencing system and found variants of the program that could operate 70 times as fast as the initial program without deterioration in functionality. In particular, mutations have been applied to lines of codes that are executed frequently when running the program.

In contrast to work which applied GI to programs offline and transfer improvements to live systems later on, Haraldsson et al. [19] integrated GI in live system (Janus Manager) which allowed for continuous self-improvements. During daytime, when users access the systems, interactions are recorded and collected, such that GI can improve the systems afterwards, based on caught exceptions. Ultimately, developers decide which patches to integrate, which simplifies maintenance.

To support practitioners, tools have been provided. One example for this is Gin, a toolbox for GI experiments for the Java ecosystem. Gin can be used to automatically transform and test Java projects [10]. GenProg is a tool that can be used for automated program repair [28]. Program variants are evolved and evaluated against existing test suites to retain functionality and resolve defects. An et al. [4] created the tool Python General Framework for Genetic Improvement (PyGGI), to allow for an easy use of GI techniques for multiple programming languages, such as Java, C, or Python. They showed, in cases studies, that PyGGI is able to be used for program repair and the improvement of non-functional performance characteristics, such as running time. In addition to lexical modification (i.e., physical lines of program source code), Version 1.1 of PyGGI supports syntactic modification of Python (i.e, modification of statements in the Abstract Syntax Tree) [5]. Version 2 enabled XML-based intermediate program representation for C, C++, C and Java programming languages [3].

Ackling et al. [1] proposed pyEDB (python evolutionary debugger), a method for automated repair of Python programs. They represented Python programs as Abstract Syntax Trees and applied modifications to repair two types of defects: incorrect relational operators (<, , >, , ==), and incorrect variable names.

Haraldsson et al. [18] applied GI to repair three small Python programs and investigate the relation between fitness (number of tests passed) and number of incremental changes to the program. Frequently, the fitness of programs did not change after a single mutation of the programs.

Unlike existing work for GI on Python [1, 18], we do not intend to use GI for improving functional performance of Python apps (e.g., fixing bugs) but aim to speed-up execution times. In particular, we use GI to convert Python to Cython programs and integrating statistical typing (i.e., inserting new lines of code).

3 PROPOSED APPROACH

In this section, we first give an overview of the Cython interpreter, and then explain how we realised *Py2Cy* using GI to automatically convert Python programs into Cython ones. At first, we outline the program representation and then operators to adapt the program.

3.1 Cython

Cython is an extension to Python, which provides a programming language and an optimized compiler [38]. Cython extends Python with access to C data types and functionalities [12]. Using features such as static typing of variables, the execution time of Python programs can be sped-up, because the interpreter does not need to determine variable data types anymore [38].

Generally speaking, Python code is already valid Cython code, as Cython is a superset of the Python programming language [38, 45]. Any part of the code that Cython cannot determine statically is compiled with Python semantics instead. Therefore, Python code can be executed by the Cython interpreter, but it can also be extended with Cython constructs to achieve performance improvements [45]. According to the Pareto Principle, most computer programs spend 80% of the run-time executing only 20% of the code [16], so editing small parts of the program can in theory achieve drastic improvements.

The *cdef* statement can be used to define local and module level variables. They can also be used to declare any C datatype, including the standard ones: char, short, int, long, long long as well as their unsigned versions, bint (boolean) and pointers. In addition to this, it can also be used to include user-defined extension types. Functions that are declared using *cdef* can take either Python object or C values as parameters and also return either Python object or C values. Lastly, *cpdef* is a hybrid function between Python's def and Cython's *cdef*, allowing both Python and C functions to call it.

To run the Cython code, it needs to first compile into a C/C++ file. It then compiles the C/C++ file into an extension module which is directly importable from Python.

3.2 Cython Syntax Tree

Cython parses its source code into an Abstract Syntax Tree (AST) during the compilation process, with each node representing a specific syntax, language feature or certain operation for code generation. Although the AST's structure and methods are not intended for public consumption in the current iteration of Cython, it can be accessed using its compiler source code libraries. Every node in the AST extends from the base Node type, where the root of each tree is a Module node and each node contains a "child attributes" variable defining its child nodes. A sample code snippet of Cython code and its AST are shown in Listing 3 and Listing 4. Notably, all nodes representing segments of code that can be statically typed also have an equivalent untyped node with a different name and structure, with the exception of the method argument node. When the Cython AST is generated from pure Python code, all nodes will use the untyped version. The equivalent typed and untyped nodes of each language feature can be seen in Table 1.

The conversion from Cython code to a visual form of its AST can be seen between listings 3 and 4. We can see that each script starts with the root ModuleNode, and each inner scope is defined by a body. In addition, the positions are given for each node in the form $pos = (name : r : c)$, where *name* is the name of the file, and r and c are the row and column of the node's position in the file. The AST also depicts how each typed node has both a base-type node and a declarator child node as its attributes, seen in line 2 and 3 of Listing 4. The base-type node is for assigning a type and the declarator is used to define the name of the method or variable. There is also a variable assignment in Line 2 and an undeclared variable in Line 3 of the Cython code in Listing 3 that translates to lines 10-16 of the AST. StatListNodes are nodes that store all the variable declarations that are in a group within the same scope. As seen in line 13 of the AST, an additional default value is given to a variable that has been assigned .

Code Feature	Untyped Node	Typed Node
Method Definition	DefNode	CFuncDefNode
Method Arguments	CArgDeclNode	CArgDeclNode
Variable Declaration	ExprNode	CVarDefNode
Unassigned Variable	ExprStatNode	CVarDefNode

Table 1: Untyped and Typed node names

```
1 cdef long factorial (int x):
2     cdef long a = 0
3     cdef long b
4     ...
```

Listing 3: Cython Code Sample

```
1  - (root): ModuleNode(pos=(fib:1:0))
2    - body: CFuncDefNode(pos=(fib:1:5))
3      - base_type: CSimpleBaseTypeNode(pos=(fib:1:5))
4      - declarator: CFuncDeclaratorNode(pos=(fib:1:19))
5        - base: CNameDeclaratorNode(pos=(fib:1:9))
6        - args[0]: CArgDeclNode(pos=(fib:1:20))
7          - base_type: CSimpleBaseTypeNode(pos=(fib:1:20))
8          - declarator: CNameDeclaratorNode(pos=(fib:1:26))
9      - body: StatListNode(pos=(fib:2:4))
10       - stats[0]: CVarDefNode(pos=(fib:2:9))
11         - base_type: CSimpleBaseTypeNode(pos=(fib:2:9))
12         - declarators[0]: CNameDeclaratorNode(pos=(fib
       :2:14))
13           - default: IntNode(type=<CNumericType long>)
14       - stats[1]: CVarDefNode(pos=(fib:3:9))
15         - base_type: CSimpleBaseTypeNode(pos=(fib:3:9))
16         - declarators[0]: CNameDeclaratorNode(pos=(fib
       :3:14))
17     ...
```

Listing 4: Cython AST Sample

All modifications to the tree are performed using tree visitor transforms. To create custom modifications to the tree, the visitor class supplied can be extended from. The visitor will match the visitor methods for each node using its name in the form: visit_exampleNode, where example is the name of the specific node class. In addition, there are also helper methods to easily convert the source code to and from its AST representation.

3.3 The *Py2Cy* Tool

This section outlines our *Py2Cy* tool, which follows a procedure inspired by PyGGI [3, 4] and GIN [10], as illustrated in Figure 1.

Py2Cy first converts a Python script into its AST as an intermediate representation and performs type injections before converting

it back into source code. It then attempts to compile the code dynamically and run it to test its runtime and output. To implement type injection, the file is first preprocessed so that new *cdef* declaration nodes (*CVarDefNode*) are created for every variable name and inserted into their respective scopes. These declarations act as the possible modification points for the type insertions. Once the position of the node and the new type is chosen, the visitor class visits the tree to find and edit the appropriate node and the type is modified by changing the `CSimpleBaseTypeNode` attached that that node. Once the patch is complete, the new AST is written to a temporary *.pyx* file.

To compile the code, the tool uses the `subprocess` module to spawn an intermediate shell process and run the "Cythonize" command on the *.pyx* file. This compiles it into a C/C++ file and then compiles the C/C++ file into an extension module which is directly importable from Python. If the code compiles, then it is dynamically imported into the code and run to test its run time speed, intended output value and other metrics. Once the patch is evaluated, another patch may be created based on the results.

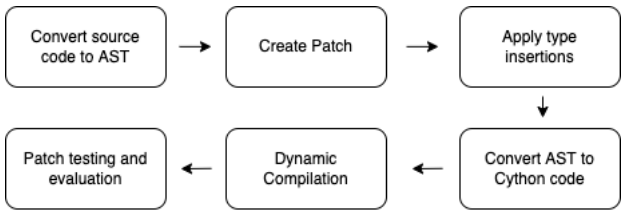

Figure 1: *Py2Cy* Pipeline

4 EXPERIMENTS

In this section, we describe the feasibility study we have carried out to assess whether our proposed tool, *Py2Cy*, can be used to reduce the running time of Python code by using Cython and static typing. In particular, we aim to answer the following research question:

To what extend *Py2Cy* is able to improve the runtime of Python programs?

To answer this question, we consider a small program (see Section 4.1) and perform a search (see Section 4.2) to generate Cython variations of the same program with additional static typing.

4.1 Program to Optimize

This section presents the Python program we aim to optimize in our experiments.

While C scripts have been used for evaluating GI for improving runtime [3], we could not find suitable Python benchmarks for runtime improvements. Therefore, we selected a program to optimize according to two criteria: 1) existing scripts; 2) at least one test case available. The first criteria ensures that we do not impact the performance of our framework by influencing the source code. The second criteria allows for a verification of the program output such that no faults are inserted after modification.

Ultimately, we chose Project Euler [1] as a source for our Python scripts. This is in accordance with Fritz and Hage [13], who investigated Project Euler scripts for type inference. Project Euler provides a collection of challenging computational problems where each has a verifiable output. Furthermore, Python scripts are available and designed by members of the community.[2] Listing 5 illustrates the Python script we investigate. In particular, we chose the tasks of generating the first n Fibonacci numbers. We consider four variables for optimization (n, a, b, i) which are distributed over two scopes.

```
1 def fib(n):
2     a = 0
3     b = 1
4     for i in range(n):
5         a, b = b, a+b
6     return a
```

Listing 5: Computing the nth Fibonacci number.

4.2 Computational Search

In this Section, we describe the search procedure and modification operators applied to convert Python to Cython scripts.

Existing approaches for GI frequently use three types of mutation: deleting of a line of code, insertion of a line of code, replacement of a line of code [11, 26, 27]. For our investigation, we assume that the available Python source code is bug-free, and we solely aim to reduce execution time by including Cython constructs (i.e., static typing). Therefore, we only use one mutation operator, which adds one out of eight static data types to an existing variable of the script: char, short, int, long and float.

Due to the oversee-able search space (i.e., the investigated Python script has four variables, see Listing 5), we decided to perform an exhaustive search to generate all possible modifications of the source Python script. Additionally, since the method is only called once in each iteration, its return type is not inserted as it would provide negligible differences.

While our exhaustive search is able to produce various Cython scripts, we need to determine if improvements over the original Python script have been achieved. In particular, the Cython scripts are not able to achieve improvements if any of these rules apply:

- Compilation Error;
- Incorrect output;
- Slower execution time.

Given that the solution for the problem at hand is known (i.e., nth Fibonacci), we can perform a test for the output of the script and determine the correctness of the output.

4.3 Validation and Evaluation Method

Due to noisy measurements when determining the execution time of Python programs [8], we repeat our experiments 30 times. We then compare the average run time of the modified Cython programs with two baseline, the unmodified Python program and an untyped Cython program. We use boxplots to illustrate runtime improvements.

For our experiments, we consider two exemplary values for n: 25, 75. By doing so, we want to show that *Py2Cy* is able to generate

[1] https://projecteuler.net/
[2] https://github.com/TheAlgorithms/Python/tree/master/project_euler

Fibonacci Term	Successful	Compilation Errors	Incorrect Value
25	117	268	240
75	28	268	329

Table 2: Computational Search Results.

different Cython scripts, tailored to the problem at hand. Moreover, investigating two cases gives insights on potential reasons for incorrect outputs, such as integer overflows.

4.4 Results

To answer our research question, we use an exhaustive search to create and evaluate 625 Cython scripts for both finding the 25th and 75th Fibonacci numbers.

Table 2 shows statistics obtained from the search. In total, there were 268 compilation failures due to incompatible datatype errors (e.g., a string variable is used for numeric operations) or invalid type conversions. Moreover, some Cython scripts produced a different output than our original Python script. The reason for this can be found in integer overflows, such as the solution is beyond the scope of integers. This explains why the results for finding the 75th Fibonacci number contains 89 more incorrect values than the search for the 25th Fibonacci number. The former requires the use of the long data type to calculate the larger terms. For the remaining Cython scripts, we measure the running time to check whether our framework can be used to automatically improve Python scripts by adding static typing with Cython.

Figure 2 illustrates a log boxplot of 30 repeated runtime measurements for each optimal patch found by the scripts and compares them to an untyped Cython and pure Python version. The exact average run-time obtained for typed Cython compared to Python are $2.08^{*}10^{-7}$s and $9.56^{*}10^{-7}$s, respectively for n=25, which means the use of static typing sped Python up by a factor of 4.5. For n=75 the runtimes are $2.09^{*}10^{-7}$s compared to $3.81^{*}10^{-6}$s, showing a more dramatic increase in speed by around a factor of 18 with typed Cython. This increase in the speed factor suggests that Cython's improvements are more prominent in programs that require heavier computation. It is also worth noting that the untyped Cython showed slight increases in speed on average compared to Python but was still significantly slower than its typed variation. Therefore, adding static types is necessary to achieve a substantial difference.

Listings 6 and 7 show the optimal patches found by the *Py2Cy* and used to compute the runtimes.

```
def fib(short n):
    cdef int a
    cdef int b
    cdef short i
    a = 0
    b = 1
    for i in range(n):
        (a,b) = (b,a+b)
    return a
```

Listing 6: Best patch n=25.

```
def fib(short n):
    cdef long a
    cdef long b
    cdef short i
    a = 0
    b = 1
    for i in range(n):
        (a,b) = (b,a+b)
    return a
```

Listing 7: Best patch n = 75.

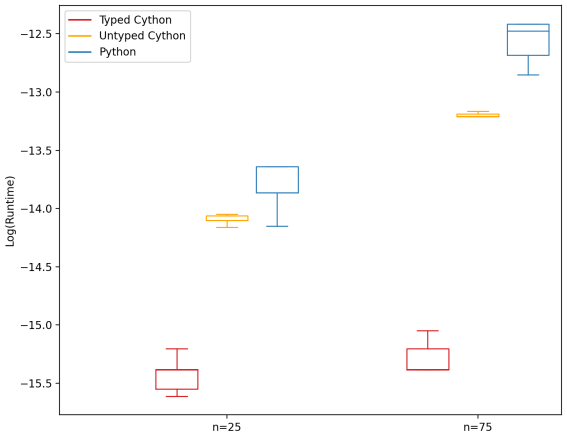

Figure 2: Run-times Boxplots for computing Fibonacci numbers.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we investigated the effectiveness of using Cython with static typing to improve the runtime of Python programs.

For this purpose, we proposed *Py2Cy*, a GI tool for inserting static types into the Python AST for obtaining an automatic conversion to Cython. We applied *Py2Cy* to convert a Python script (i.e., the sum of n Fibonacci numbers) into a Cython script with static typing. Our results showed that *Py2Cy* is successful in reducing the runtime by a factor of 4.5 and 18 for calculating the 25th and 75th Fibonnaci term, respectively, from a pure python script.

We aim to further develop *Py2Cy* to include more static-typing features. Firstly, it would be useful to automatically declare loop iteration index variables before the loops, as this is common Cython practice. The number of types to test in the patches will also be expanded to allow for editing programs with a larger variety of possible types. Additionally, Cython pointers can be looked at to achieve typing for arrays and adding *cpdef* declarations can also be considered, to make methods callable from both Cython and Python.

Although *Py2Cy* will only be able to achieve efficiency improvements to Python, it can be used in conjunction with PyGGI [3] to make functional improvements such as fixing bugs before its translation to statically typed Cython.

While we were able to show the feasibility of applying *Py2Cy* for runtime improvements, there are further tasks that can be performed to strengthen our findings in future work. Further experiments can be conducted on additional problems of higher complexity (i.e., more variables to type), which will increase the search space. With the consideration of complex optimization problems and larger search spaces, the use of sophisticated search algorithms, such as Hill Climbing [3] or Genetic Algorithms, becomes interesting. Moreover, static inference tools [13, 22, 35, 40] could be considered such that the search focuses on likely variable types rather than choosing types at random.

ACKNOWLEDGMENTS

M. Hort and F. Sarro are supported by the ERC grant 741278 (EPIC).

REFERENCES

[1] Thomas Ackling, Bradley Alexander, and Ian Grunert. 2011. Evolving patches for software repair. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. 1427–1434.

[2] Joël Akeret, Lukas Gamper, Adam Amara, and Alexandre Refregier. 2015. HOPE: A Python just-in-time compiler for astrophysical computations. *Astronomy and Computing* 10 (2015), 1–8.

[3] Gabin An, Aymeric Blot, Justyna Petke, and Shin Yoo. 2019. PyGGI 2.0: Language independent genetic improvement framework. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1100–1104.

[4] Gabin An, Jinhan Kim, Seongmin Lee, and Shin Yoo. 2017. PyGGI: Python General framework for Genetic Improvement. (2017), 536–538.

[5] Gabin An, Jinhan Kim, and Shin Yoo. 2018. Comparing line and AST granularity level for program repair using PyGGI. In *Proceedings of the 4th International Workshop on Genetic Improvement Workshop*. 19–26.

[6] Nader Al Awar, Steven Zhu, George Biros, and Milos Gligoric. 2021. A performance portability framework for Python. In *Proceedings of the ACM International Conference on Supercomputing*. 467–478.

[7] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2010. Cython: The best of both worlds. *Computing in Science & Engineering* 13, 2 (2010), 31–39.

[8] Mahmoud A Bokhari, Bobby R Bruce, Brad Alexander, and Markus Wagner. 2017. Deep parameter optimisation on android smartphones for energy minimisation: a tale of woe and a proof-of-concept. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, 1501–1508.

[9] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. 18–25.

[10] Alexander EI Brownlee, Justyna Petke, Brad Alexander, Earl T Barr, Markus Wagner, and David R White. 2019. Gin: genetic improvement research made easy. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 985–993.

[11] Bobby R Bruce, Justyna Petke, and Mark Harman. 2015. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 1327–1334.

[12] Lisandro D Dalcin, Rodrigo R Paz, Pablo A Kler, and Alejandro Cosimo. 2011. Parallel distributed computing using Python. *Advances in Water Resources* 34, 9 (2011), 1124–1139.

[13] Levin Fritz and Jurriaan Hage. 2017. Cost versus precision for approximate typing for Python. In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. 89–98.

[14] Juan J Galvez, Karthik Senthil, and Laxmikant Kale. 2018. CharmPy: A python parallel programming model. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 423–433.

[15] Rahul Garg and José Nelson Amaral. 2010. Compiling python to a hybrid execution environment. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. 19–30.

[16] Mechelle Gittens, Yong Kim, and David Godwin. 2005. The vital few versus the trivial many: examining the Pareto principle for software. In *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, Vol. 1. IEEE, 179–185.

[17] Serge Guelton, Pierrick Brunet, Mehdi Amini, Adrien Merlini, Xavier Corbillon, and Alan Raynaud. 2015. Pythran: Enabling static optimization of scientific python programs. *Computational Science & Discovery* 8, 1 (2015), 014001.

[18] Saemundur O Haraldsson, John R Woodward, Alexander EI Brownlee, and David Cairns. 2017. Exploring fitness and edit distance of mutated python programs. In *European Conference on Genetic Programming*. Springer, 19–34.

[19] Saemundur O Haraldsson, John R Woodward, Alexander EI Brownlee, and Kristin Siggeirsdottir. 2017. Fixing bugs in your sleep: How genetic improvement became an overnight success. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 1513–1520.

[20] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and software Technology* 43, 14 (2001), 833–839.

[21] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

[22] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-based type inference for Python 3. In *International Conference on Computer Aided Verification*. Springer, 12–19.

[23] Dejice Jacob and Jeremy Singer. 2019. ALPyNA: acceleration of loops in Python for novel architectures. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. 25–34.

[24] William B Langdon. 2015. Genetic improvement of software for multiple objectives. In *International Symposium on Search Based Software Engineering*. Springer, 12–28.

[25] William B Langdon. 2015. Genetically improved software. In *Handbook of Genetic Programming Applications*. Springer, 181–220.

[26] William B Langdon and Mark Harman. 2014. Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation* 19, 1 (2014), 118–135.

[27] William B Langdon and Justyna Petke. 2017. Software is not fragile. In *First Complex Systems Digital Campus World E-Conference 2015*. Springer, 203–211.

[28] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

[29] Ami Marowka. 2018. Python accelerators for high-performance computing. *The Journal of Supercomputing* 74, 4 (2018), 1449–1460.

[30] Riccardo Murri. 2014. Performance of Python runtimes on a non-numeric scientific code. *arXiv preprint arXiv:1404.6388* (2014).

[31] Travis E Oliphant. 2007. Python for scientific computing. *Computing in science & engineering* 9, 3 (2007), 10–20.

[32] Justyna Petke, Saemundur O Haraldsson, Mark Harman, William B Langdon, David R White, and John R Woodward. 2017. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (2017), 415–432.

[33] Justyna Petke, Mark Harman, William B Langdon, and Westley Weimer. 2014. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In *European Conference on Genetic Programming*. Springer, 137–149.

[34] Russell Power and Alex Rubinsteyn. 2013. How fast can we make interpreted Python? *arXiv preprint arXiv:1306.6047* (2013).

[35] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 209–220.

[36] Jose Manuel Redondo and Francisco Ortin. 2014. A comprehensive evaluation of common python implementations. *IEEE Software* 32, 4 (2014), 76–84.

[37] Armin Rigo and Samuele Pedroni. 2006. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 944–953.

[38] Alexander Roghult. 2016. Benchmarking Python Interpreters: Measuring Performance of CPython, Cython, Jython and PyPy.

[39] Alex Rubinsteyn, Eric Hielscher, Nathaniel Weinman, and Dennis Shasha. 2012. Parakeet: A just-in-time parallel accelerator for Python. In *4th {USENIX} Workshop on Hot Topics in Parallelism (HotPar 12)*.

[40] Michael Salib. 2004. *Starkiller: A static type inferencer and compiler for Python*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[41] Leonhard Spiegelberg, Rahul Yesantharao, Malte Schwarzkopf, and Tim Kraska. 2021. Tuplex: Data Science in Python at Native Code Speed. In *Proceedings of the 2021 International Conference on Management of Data*. 1718–1731.

[42] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, et al. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688* (2016).

[43] Guido Van Rossum et al. [n.d.]. Python Programming Language.

[44] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* 17, 3 (2020), 261–272.

[45] Ilmar M Wilbers, Hans Petter Langtangen, and Åsmund Ødegård. 2009. Using cython to speed up numerical python programs. *Proceedings of MekIT* 9 (2009), 495–512.

[46] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. 2015. Deep parameter optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. 1375–1382.