Miguel Agueda-Cabral

Dr. Thomas

CS 4100

16 November 2020

Python at C-Speed

Abstract

The Python programming language is among the top two most popular programming languages in the world [13]. Above Python is C, the most popular programming language [13]. Each of these languages have different strengths and weaknesses. C programs are well known for running quickly and efficiently, however, the same cannot be said for Python [1]. Equivalent programs written in both C and Python have been used to compare the two languages. Using a program which computes all the prime numbers from zero through one thousand, it was shown that Python is orders of magnitude less quick and less efficient when compared to C [12]. This does not hamper Python's usage, however. Python is still rising in popularity [13]. It would be beneficial to incorporate C into Python, or vice-versa, as to improve either language. A demand for speeding up Python has been shown through the support for existing libraries that incorporate such functionality. Take, for example, the NumPy project. NumPy incorporates many mathematical functions into a fast and easy to use library. Thousands of developers have contributed to the NumPy source code, for free [3]. Many more developers use the library daily [3]. NumPy can attribute some of its popularity to the fact that it is implemented in C, then ported to Python. This implementation makes NumPy's mathematical operations run much faster than a pure Python implementation. The phrase, "pure Python," refers to source code that is solely reliant on the Python language. NumPy, as it is implemented in C, is not considered to be pure Python.

Introduction

As a simple demonstration of the performance differences between NumPy and pure Python, separate versions of *Average* and *Root Mean Squared* calculators have been implemented below. For this example, each of these calculators have been implemented twice.

One version of each calculator utilizes NumPy functions whereas the other version uses only Python's built-in loops and mathematical operations. To compare these implementations, large data sets of sequential numbers were generated and then fed into each function. The processing time for each function and the associated data set were saved for the later creation of line charts. The line charts plot each function's run time against the size of the function's input.

```python
def numpy_find_avg(samples):
    """ NumPy implementation of the Average function."""
    average = numpy.mean(samples) # Get average using NumPy's `mean` function.
    return average

def python_find_avg(samples):
    """ Python implementation of the Average function."""
    total_sum = 0 # Initialize sum to 0.
    for x in samples: # Loop over all samples.
        total_sum += x # Add sample to sum counter.
    count = len(samples) # Get number of elements in `samples`.
    average = total_sum / count
    return average

def numpy_find_rms(samples):
    """ NumPy implementation of the Root Mean Squared function."""
    square = numpy.power(samples, 2)
    mean_square = numpy.mean(square)
    rms = numpy.sqrt(mean_square)
    return rms

def python_find_rms(samples):
    """ Python implementation of the Root Mean Squared function."""
    square_sum = 0 # Initialize counter for sum of squares.
    for x in samples: # Loop over every sample in list.
        square_sum += (x ** 2) # Add squared sample to counter.
    count = len(samples)  # Get total number of samples for average calculation.
    mean_square = square_sum / count # Compute average of squares.
    rms = mean_square ** 0.5 # Take square root, raise to 1/2 power.
    return rms
```

Code by Author

Figure 1

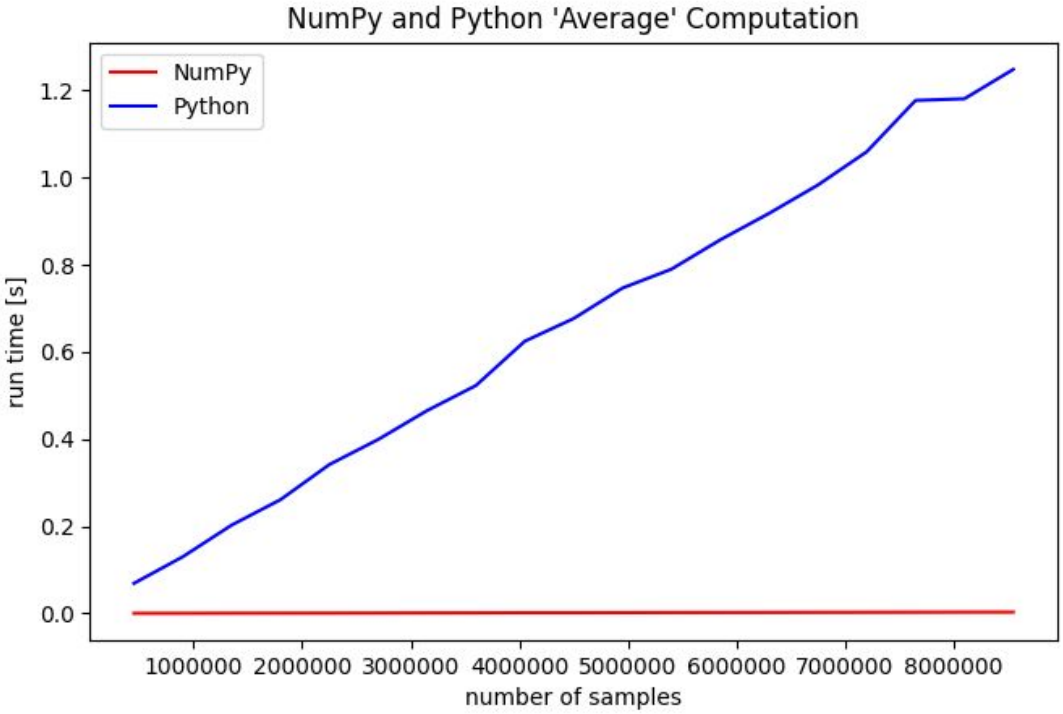

Figure by Author

As depicted in the chart above, the total amount of time required to execute each *average* function becomes drastically different as the number of samples increases. Graphically, it appears that averaging with NumPy requires constant time. The same characteristics are not present in the pure Python implementation. Without using NumPy, the time required to average eight million samples is about eight times greater than the time required to average one million samples. When viewing the NumPy run time chart in isolation, as can be seen in *figure 2*, it is easier to see that the function does not actually require constant time. In fact, the NumPy function's run time increases faster than that of the pure Python function. The NumPy function takes about 12 times longer to process eight million samples than it takes to process one million samples. This increase makes the time required by a pure Python implementation increase more slowly than with NumPy. Regardless of the rate of change with respect to input volume, NumPy is able to perform the computations orders of magnitude faster than pure Python. In computing the average over eight million samples, the calculator implemented using NumPy was approximately 40 times faster than that implemented in pure Python.

Figure 2

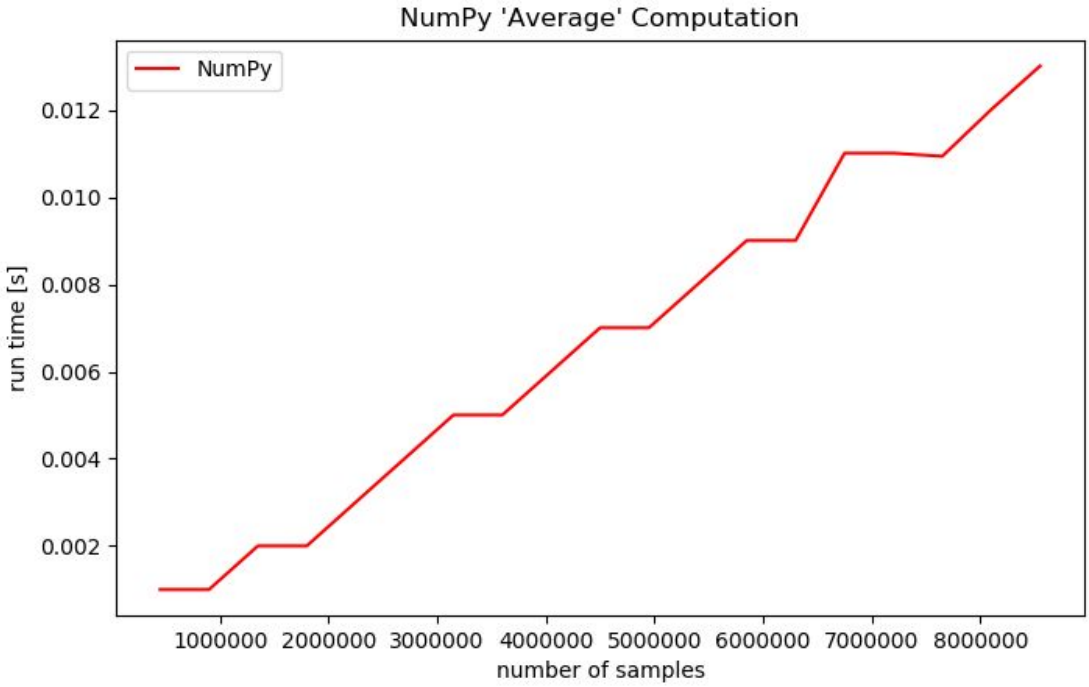

Figure by Author

Figure 3

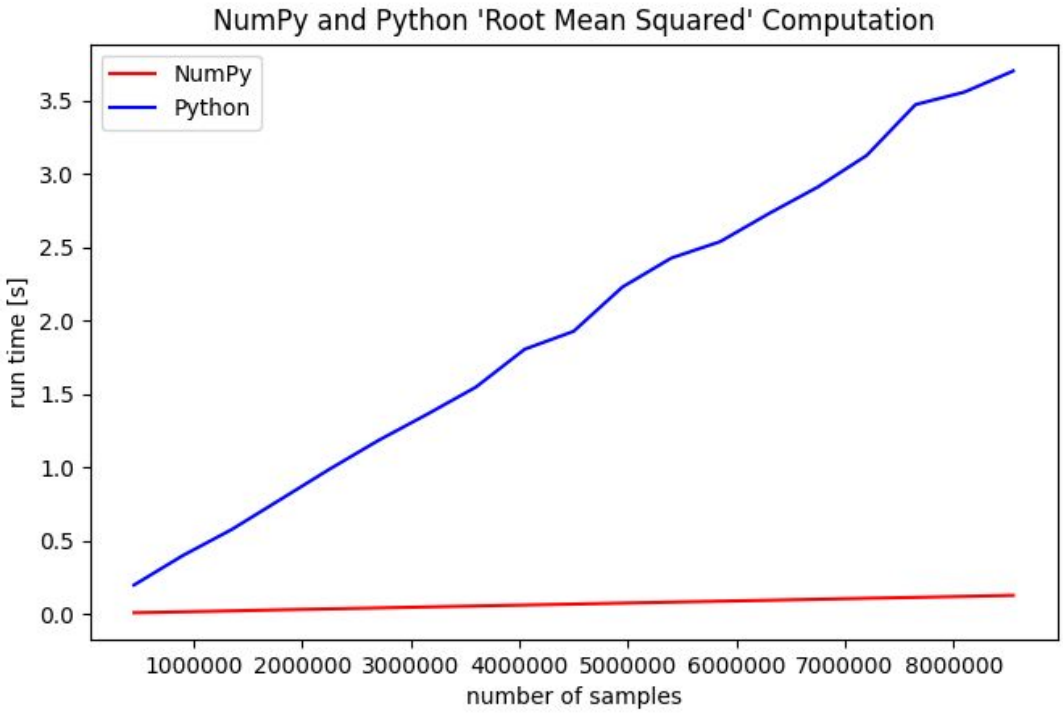

Figure by Author

When carrying out the Root Mean Squared (RMS) computation, the time required by the NumPy implementation increased slightly as the number of samples approached nine million. This is in stark contrast to the increase in time required by the pure Python implementation. In computing the RMS value, both implementation's time requirement increased linearly with the size of the input. Despite the similar complexities of the two functions, the NumPy implementation still outperforms the pure Python implementation. Using NumPy was approximately 30 times faster than using pure Python, as per *figure 3*. The difference in time required to execute each implementation shows only one reason why NumPy has risen in popularity. Through a closer look at the program source code, it can be seen that the NumPy versions of each calculator are implemented in less lines of code than the pure Python implementation. Not only that, the NumPy interface helps improve readability of mathematical notation by representing each function as the English name for some mathematical symbol or operation. All these factors make NumPy, and similar libraries, easy to learn and productive to use [8].

Figure 4

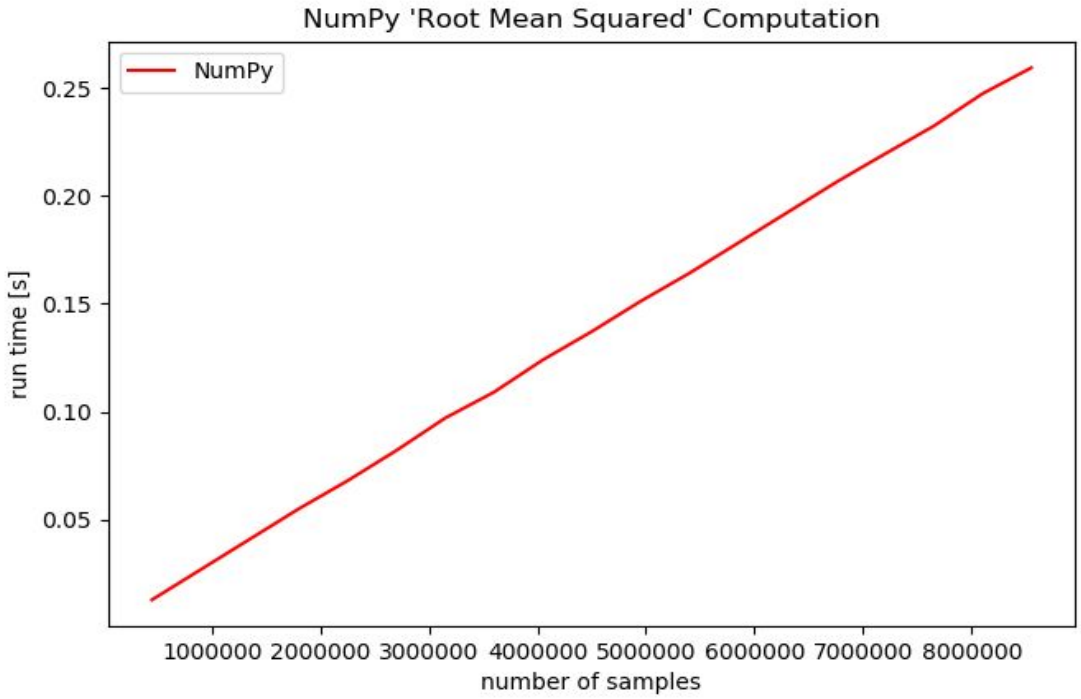

Figure by Author

The run time optimizations achieved by implementing NumPy's mathematical functions are only accessible through the NumPy library. When a function or procedure is not already implemented in NumPy, a developer must be careful to avoid introducing a bottleneck with pure Python functions. Through the incorporation of extension programming and various 3rd party libraries, one can develop customized functions that are on par with those from the NumPy library. Learning to utilize various optimizations can help improve Python program performance. In some cases, minimal to little effort is required to transform an ordinary Python program into a Python program which can run at C-speed.

Background

Before discussing the ways in which Python can be sped up, it may be helpful to understand why it is that Python is comparatively slow. First, Python is a dynamically typed and dynamically scoped language [1]. Dynamic typing allows Python's variables to take on many forms throughout the runtime of a single program. This flexibility allows for ease in expressing "complicated ideas in the syntax" [8]. However, dynamically typed programming languages are not known for being the most optimal in terms of run time [9]. Further, Python function calls use "late binding," which is a result of dynamic scoping [1]. During a program's runtime, function calls incur "the overhead of a lookup in the global hash table for the current context" of the function being called [1]. This is necessary since an object's type is able to change throughout the life of a Python program. Even when an object's type remains static, the Python interpreter still carries out this lookup, adding a constant factor of time to every function call [1]. Python's table lookups do not stop there. The Python interpreter will perform "another lookup in the builtins table if the global lookup misses" [1]. This overhead is incurred every time a global object is accessed.

Second, when speaking about the Python programming language, it can be easy to lose information due to technical details commonly left out of a discussion. When Python is said to be "slow," what is actually meant is that a specific Python implementation is slow. Even then, a Python implementation is not a unique piece of software that can be pointed to. There exist many Python implementations, although only one of them is considered to be default. The default Python implementation is known as CPython. Other implementations include, but are not limited to, IronPython, Jython, PythonNet, and RPython. Each of these implementations differ in both

performance and support. CPython "is the original and most-maintained implementation of Python," as such, improving CPython will be the primary focus of this paper [10].

Extension Programming

There exist several ways to implement C programs into Python. One of these ways is by extension programming using the *C extension interface* [9]. "The C extension interface," and thus, extension programming, "is specific to CPython," meaning that it cannot be accessed from any other Python implementation. Through extension programming, a Python environment can import *extensions* written in C. An extension is a C library that implements *PyObject*s along with additional functionality. A PyObject represents "an arbitrary Python object," which is the base class for all other Python types [11]. PyObjects are implemented in C but can be recognized by the Python interpreter as Python objects. This allows for functions to run as compiled machine-code while still being accessible to the Python environment. This relationship is not only one-sided. Through the same type of extension programming, C programs can make use of Python programs and Python's built-in functions [9]. Extension programming is the default method for extending Python's functionality with C. However, writing an extension is non-trivial and the official Python documentation suggests end-users use a third-party library to create Python extensions in C [9]. Said third-party libraries will make use of Python's extension programming via standardized and already-debugged templates.

Figure 5: Extension Programming Process Diagram

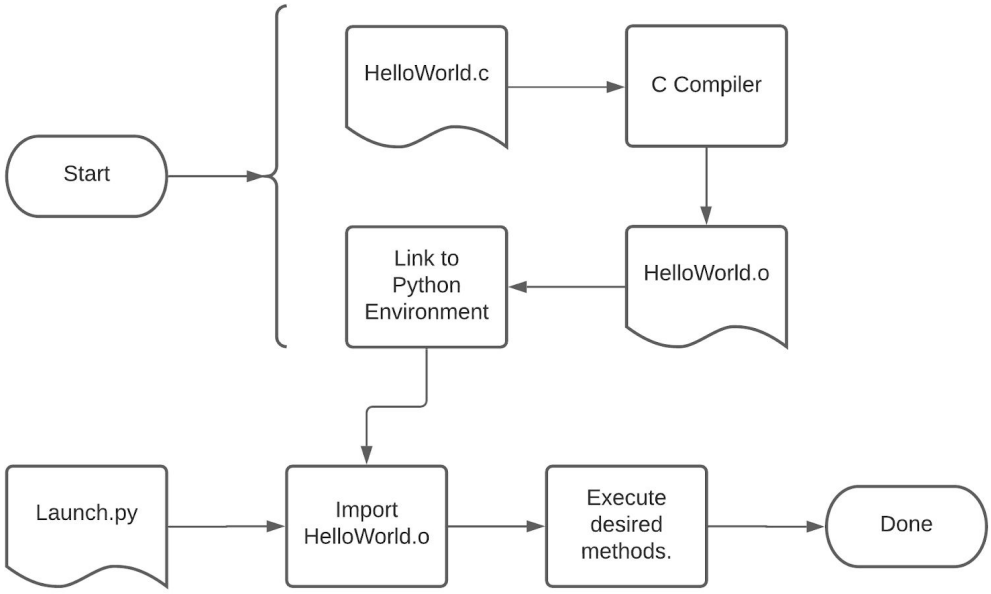

Figure by Author

Cython

Another third-party tool for improving Python run times, through association with C, is the Cython compiler; Not to be confused with CPython. The Cython compiler can compile Python source code into C source code. The generated C code can then be compiled and linked back into a Python environment [2]. This method allows a programmer to write and debug source code in pure Python. Then, when the program is ready, it can be compiled to C in order to achieve run time optimizations. This process is similar to extension programming except that the extension is implemented in Python as opposed to C. An advantage that Cython brings to extension programming is that extensions for Python can be written in Python. This can be beneficial when one does not want to manually implement parts of their application in C. Using Cython is also considered to be more safe than writing one's own extension from scratch because Cython automatically handles all extension library references, de-references, wrapping, and linking. Using Cython to convert Python into C can lower the barrier to entry for extension programming, possibly leading to more widespread adoption.

Figure 6: Cython Process Diagram

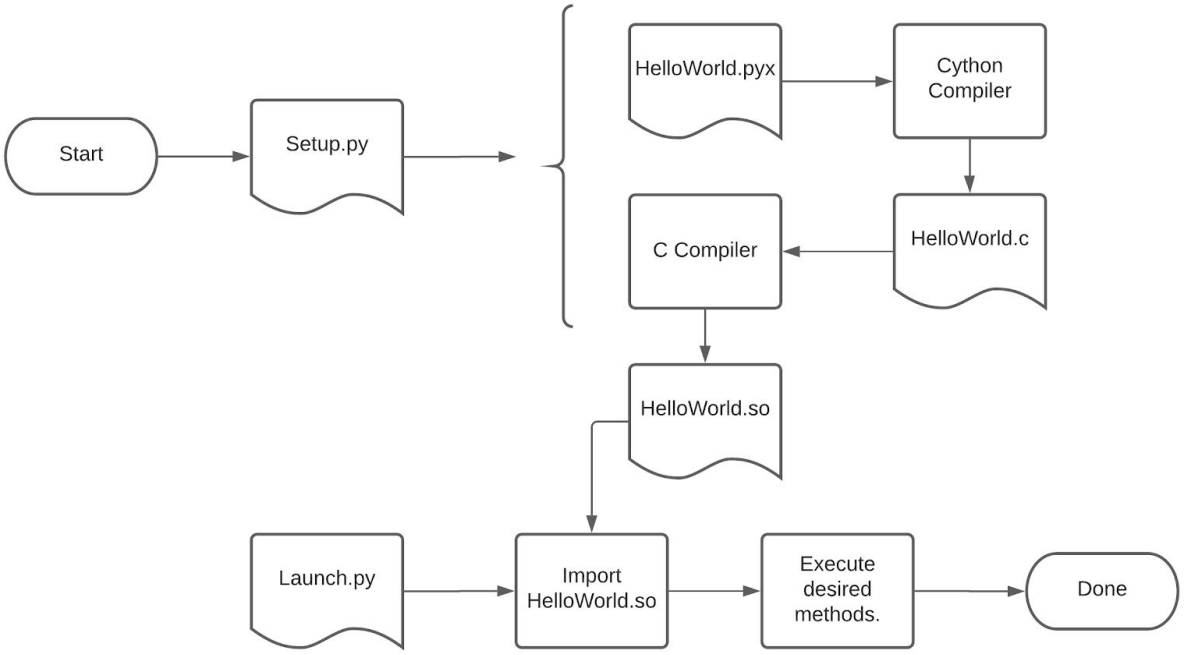

Figure by Author

Cython is not only a tool for converting Python source code to C. Cython also has its own implementation of a programming language, again, known as *Cython*. *Cython*, the programming language, can be thought of as Python with C types. The Cython implementation refers to these types as *extension types*. Normally, Python classes are implemented as a Python dictionary object [2]. This dictionary contains all of the class's attributes and functions. When a function is invoked from a class, the class's dictionary is searched for that function's signature [1]. By giving explicit types to Python classes, the Cython compiler can define the class using a C-struct, which is accessible at the C level [2]. This removes the need to perform a Python dictionary lookup when invoking some function from the class [2]. Eliminating this overhead means that Cython programs can do the same task as their pure Python counterpart, in less time.

The ability to extend Python, using C, is reliant on *CPython,* Python's default implementation. CPython is responsible for compiling Python code into an intermediary language known as bytecode [1]. After generating a bytecode from a particular source, CPython interprets the bytecode using a virtual machine. This process is notorious for being slow and inefficient [1], however, speed gains are achievable. One study of Python programs, which were

interpreted with CPython, found that “more than half of the total execution time” can be attributed to Python’s dynamic typing and number boxing [1]. Number boxing and unboxing occurs, for example, when arithmetic operations are performed on Python’s numerical objects. Python’s integers, floats, and complex numbers are implemented as objects. This allows Python’s numbers to have methods declared within them. Take the following Python script, which demonstrates Python’s *float class* function, *is_integer.*

```
X = 1.0  # Declare X as a float.
if X.is_integer():  # Demonstrate Python's methods within float objects.
    print(F"Even though X is of type {type(X)}, it is still considered an integer.")
```

Code by Author

The function, *is_integer,* returns True if the float object, X, holds an integer value, in the mathematical sense of the word integer. When executed, the above script outputs the following; *Even though X is of type <class 'float'>, it is still considered an integer.* While this feature may make it convenient to program in Python, it also leads to extra steps needing to be carried out before any arithmetic operations can be done on these numerical objects. For example, when adding a constant number to a variable, *X*, we are actually only interested in adding to the *real* attribute of *X*, not the entire object representing *X*. In fact, the entire float object contains fifty seven combined attributes and methods, each serving a different purpose. Many of these attributes exist to provide convenience functions to the Python programmer. For example, another function within the *float* object is *as_integer_ratio*, which returns a float value as numerator and denominator components. While this may prove to be a convenient function, it introduces a constant overhead to every single float object that is instantiated in a Python program. By default, there exists no way to *turn off* these convenience functions. However, with the advent of supplemental tools, *turning off* these convenience functions is becoming more of a reality.

Numba

Several runtime optimizations can be achieved by supplementing the Python interpreter used to run a program. One such tool to achieve this is *Numba*, a “Just-in-Time compiler for CPython” [5]. Numba is primarily aimed at speeding up heavy, matrix- or vector-based operations in Python [5]. To accomplish this, Numba takes advantage of container attributes to generate efficient looping sequences, in machine code [5]. When a matrix has a determinable

type, as NumPy arrays and for-loops have, Numba can generate code that is entirely independent of the Python interpreter. Numba refers to this as "*nopython mode* because all operations can be lowered into efficient machine code without relying on the Python runtime for any operation" [5]. Functions that can be processed in *nopython mode* can be run in parallel threads, further optimizing the program's total run time [5]. The difficulty in implementing this functionality comes from Python's lack of explicit types. Without explicit type information, the Numba compiler must infer upon or rely on the developer's annotation for information regarding an object's type.

As stated before, Numba is a *supplement* to CPython, not its replacement. Numba requires users to annotate their source code functions with a Python construct known as *decorators* [5]. Decorators are specialized lines of code that come just before a function's declaration. Numba uses these decorators to activate separate compilation routines for the associated function [5]. Since Numba is known as a *Just-in-Time (JIT)* compiler, the decorator used to signal Numba is *@jit.* Placing this decorator above a Python function will signal Numba to compile the function into an intermediate representation (IR) [5]. An IR of a Python function can be thought of as a branch of the function which contains type information. If the type information can be properly inferred by Numba, then the function will be compiled in *nopython* mode, as previously explained. When Numba cannot infer a variable type, the IR is compiled in a different mode, known as *object mode* [5]. In object mode, the compiled output is fed back into the Python runtime when the program is instantiated. Due to the requirement upon the Python runtime, functions compiled in *object mode* will not achieve run time optimizations as great as *nopython mode* [5]. However, compilation in *object mode* does remove the overhead imposed by the CPython interpreter's compilation step, thus improving program runtime speeds [5]. For a demonstration of Numba's contributions, a very slight variation of *python_find_rms* will be implemented once again. The only difference between the pure Python implementation and Numba's implementation of the RMS calculator is that Numba's @jit decorator prepends the latter.

```
def python_find_rms(samples):
    """ Python implementation of the Root Mean Squared function.

        Stated again, for reference.
    """
    square_sum = 0 # Initialize counter for sum of squares.
    for x in samples: # Loop over every sample in list.
```

```
        square_sum += (x ** 2) # Add squared sample to counter.
    count = len(samples)  # Get total number of samples for average calculation.
    mean_square = square_sum / count # Compute average of squares.
    rms = mean_square ** 0.5 # Take square root, raise to 1/2 power.
    return rms


@jit  # Numba's J-I-T decorator. This makes all the difference.
def numba_find_rms(samples):
    """ Numba supplemented implementation of the Root Mean Squared function."""
    square_sum = 0 # Initialize counter for sum of squares.
    for x in samples: # Loop over every sample in list.
        square_sum += (x ** 2) # Add squared sample to counter.
    count = len(samples)  # Get total number of samples for average calculation.
    mean_square = square_sum / count # Compute average of squares.
    rms = mean_square ** 0.5 # Take square root, raise to 1/2 power.
    return rms
```

Code by Author

Figure 7

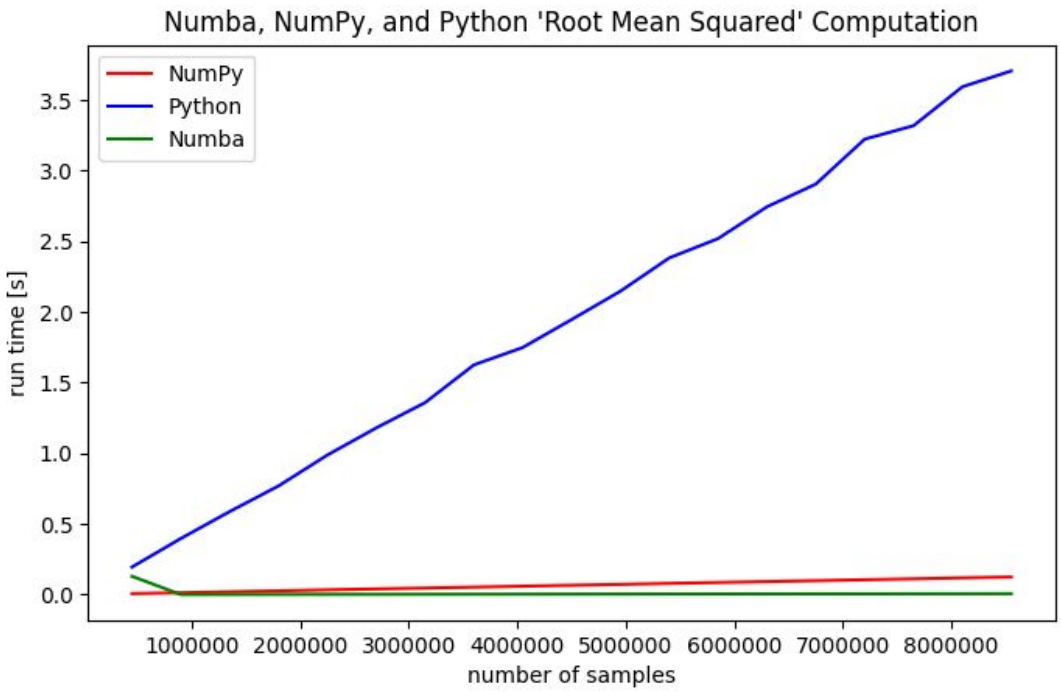

Figure by Author

In this demonstration, the Numba compiler was left to its own devices in discovering the types of each variable in the RMS calculator. Surprisingly, Numba outperformed NumPy in this demonstration of a naive RMS calculator implementation. Although better use of NumPy's functions could be made in order to further optimize the NumPy RMS calculator, that is outside

the scope of this demonstration. To be shown was the freedom that can be attained by implementing Numba as opposed to NumPy. Numba allows a Python programmer to have direct access to NumPy-like runtime speeds within their custom functions. This allows programmers to break away from the strictly mathematical structure of NumPy, and expand fast, C-extended functions to other parts of Python. This freedom does come at some cost, however. Numba postpones function compilation until the function "is first called," then Numba "just-in-time compiles the function" for use throughout the rest of the program's operation [5]. The overhead imposed by Numba's function compilation, at runtime, results in Numba's initially decreasing run time, as seen in *Figure 7.* Due to this imposed overhead, some consideration must go into the program that is incorporating Numba's functionality. If the program is meant to be run once, or only a few times, the constant overhead imposed by Numba may lead to poorer performance than using NumPy. However, using Numba is still considerably faster than using pure Python.

To ensure Numba's type inference happens properly each time, Numba's @jit decorator can be overloaded with object type information. A function's type information can be conveyed to Numba using a function signature [7]. A function's signature defines its expected input types as well as its output type. This not only allows Numba to unambiguously declare types, it also allows the Python programmer to declare types within a Python program [7]. Currently, Numba supports declaration of most C primitive types, arrays, and even other functions inside of a function signature. Although declaring Numba function signatures is not always necessary, it can be helpful for placing some restrictions onto Python function definitions. Normally, pure Python functions rely on *assertions* and manual type checking when an input is required to be of a specific type. An assertion is "a convenient way to insert debugging" information into a Python program [10]. As such, assertions do not actually prohibit incorrect use of a function, instead, they notify the programmer of function misuse when it actually occurs, at runtime. Since Python is an interpreted language, assertions cannot be checked before the program is actually executed. This is in contrast to C, where type checking is done well before a program is initialized. The issue of implementing types into Python is *double edged*. On the one hand, Python's lack of explicit types can make programs very difficult to debug. On the other hand, requiring developers to specify every object's type can be time consuming and counterproductive to rapid prototyping needs [8].

Conclusion

Python, in its pure form, tends to be too slow for heavy and continuous computational tasks [8]. However, despite this fact, Python has continued to rise in popularity [13]. Part of Python's hype has originated from highly efficient third-party libraries, such as NumPy [8]. The NumPy library allows Python programs to escape the slow CPython implementation, and enter into the domain of machine code, through extension programming [9]. Said machine code is usually in the form of precompiled, ready-to-go function calls. However, with the advent of tools such as Cython and Numba, the need to precompile an extension library is disappearing. In slimming down the steps required to create a Python extension, Cython and Numba have made extension creation more accessible to the Python programmer. With third-party Python compiling tools, Python extensions can now be written in Python, as opposed to C. Even better, extensions can be implemented directly into a Python source file, along with pure Python source code, removing the need to create complex directory structures. These simplifications to extension programming will allow for a more widespread adoption of custom extensions in Python. With the slowing of Moore's Law, programmers cannot rely on next year's new hardware to carry the performance of their application [6]. Instead, increases in computer capabilities will rely more and more on the implementation of optimal software [6]. As extension programming is the primary method to optimizing Python's runtime performance, it will prove to be increasingly useful, so long as CPython remains the default Python implementation.

Bibliography

[1] Barany, G. 2014. Python Interpreter Performance Deconstructed. In Proceedings of the Workshop on Dynamic Languages and Applications (Dyla'14). Association for Computing Machinery, New York, NY, USA, 1–9. DOI:https://doi-org.libproxy.csustan.edu/10.1145/2617548.2617552

[2] Stefan Behnel, Robert Bradshaw, Dag Sverre Seljebotn, Greg Ewing, William Stein, Gabriel Gellner, et al. 2020. Cython Documentation: Extension types (aka. cdef classes). Retrieved from http://docs.cython.org/en/latest/src/tutorial/cdef_classes.html#extension-types-aka-cdef-classes

[3] GitHub. 2020. NumPy GitHub Repository. Retrieved from https://github.com/numpy/numpy

[4] LLVM. 2020. The LLVM Compiler Infrastructure. Retrieved from https://llvm.org/

[5] Lam S. K., Antoine Pitrou, and Stanley Seibert. 2015. Numba: a LLVM-based Python JIT compiler. In Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM '15). Association for Computing Machinery, New York, NY, USA, Article 7, 1–6. https://dl.acm.org/doi/10.1145/2833157.2833162

[6] Leiserson, C. E., Thompson, N. C., Emer, J. S., Kuszmaul, B. C., Lampson, B. W., Sanchez, D., & Schardl, T. B. (2020). There's plenty of room at the Top: What will drive computer performance after Moore's law?. Science (New York, N.Y.), 368(6495), eaam9744. https://doi.org/10.1126/science.aam9744

[7] Numba. 2020. Numba Documentation. Retrieved from https://numba.readthedocs.io/en/stable/index.html#numba-documentation

[8] Oliphant, T. E., "Python for Scientific Computing," in Computing in Science & Engineering, vol. 9, no. 3, pp. 10-20, May-June 2007, doi: 10.1109/MCSE.2007.58.

[9] Python Software Foundation. 2020. Python Documentation: Extending Python with C or C++. Retrieved from https://docs.python.org/3/extending/extending.html#extending-python-with-c-or-c

[10] Python Software Foundation. 2020. Python Documentation: Python Language Reference. Retrieved from https://docs.python.org/3/reference/index.html#the-python-language-reference

[11] Python Software Foundation. 2020. Python Documentation: Python/C API Reference Manual. Retrieved from https://docs.python.org/3/c-api/index.html#python-c-api-reference-manual

[12] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy efficiency across programming languages: how do energy, time, and memory relate? In Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2017). Association for Computing Machinery, New York, NY, USA, 256–267. Retrieved from https://doi.org/10.1145/3136014.3136031

[13] TIOBE. 2020. TIOBE Index for October 2020. Retrieved from https://www.tiobe.com/tiobe-index/