# WIP: Focusing on Programmer Literacy in the Time of AI-Aided Code Generation

Joel A. Rosiene
*Department of Computer Science*
*Eastern Connecticut State University*
Windham, CT USA
rosienej@easternct.edu

Carolyn Pe Rosiene
*Department of Computing Sciences*
*University of Hartford*
W. Hartford, CT USA
rosiene@hartford.edu

***Abstract*—This work-in-progress, innovative-practice, full paper describes an approach to the training of computer scientists and engineers focused on the reading of programs generated by AI-based Large Language Models (LLM). In contrast to the practice of Literate Programming, in which the coder crafts a solution which reads as English, the proposed approach treats an existing programming as a "shorthand", which, when articulated, is explanatory. LLMs amount to enhanced search, having digested numerous examples with attention to context in order to provide a most "desired" result for a given prompt. How well the LLM generates a solution for a particular language varies substantially.**

**The paper gives numerous LLM-generated code examples of how the proper reading of programming statements conveys their function and requires little or no commentary. The Sieve of Eratosthenes will be used to contrast the LLM-generated code against code which adheres to the language's idiom.**

***Index Terms*—Programmer literacy, code generation, Large Language Models.**

## I. Introduction

It is proposed that literate programming approach which "weaves" the documentation with the "tangled" code to highlight the programmer's thinking that leads to the code can be replaced by fully stating a language statement correctly [1]. When modern programming constructs are treated as shorthand in which, when articulated, the code's intent becomes clear, and when coupled with unit tests, they can greatly reduce the need for commenting code [2].

When using an AI-based Large Language Model (LLM, e.g., Copilot), programs based on an English language prompt are generated which provide the most likely "correct" response to the prompt. The LLMs harvest examples from online resources which are common to the context provided by the prompt. In this way, LLMs extend the literate programming approach with the *prompt* acting as the programmer's intent expressed in English to be "tangled" with the resulting code.

The resulting code must be evaluated for correctness, which requires a solid understanding of the target language. This has led to the authors' approach to teaching the implementation of an algorithm in a particular programming language while focusing on how the language's constructs are correctly interpreted in each programming paradigm. We proceed with a background of how a variety of statements in C (C++), Java and Python should be interpreted, followed by the interpretation of LLM-generated code from the included prompts.

## II. Background

There is support for traditional literate programming [3] which consists of integrated development environments that embed natural language with the source code. Adherence to the idiom (e.g. Pythonic) facilitates language as shorthand for literate programming. Coding a routine in C (imperative), C++ or Java (object-oriented) or Python (multi-paradigm) often leads to distinct implementations. As an illustration of using programming constructs as shorthand for literate programming, we will discuss assignments and iteration below.

### A. Assignments

In C, the code fragment int x = 5; should be spoken as "the labelled memory location x is assigned the value of 5." While in Java, the same code fragment, int x = 5;, is best to consider (speak) as "the integer variable x is set to a 5", since autoboxing (depending on the version of Java) based on context leads to confusion; the JVM manages memory and does not directly relate to a virtual (or physical) memory location.

In the case of Python, x = 5 is spoken as "x takes a reference to 5." This is fundamental in understanding how Python manages its storage, which is allocated on assignment.

It is proposed that by treating the statement as shorthand, and articulating ("speaking" the full language's specific meaning of) the statement are beneficial in understanding the function of the algorithm's implementation.

### B. Iteration

Iteration in an imperative language (e.g. while, until, and for loops in C) are focused on the repetition opposed to any enclosed container. For example, the code fragment

```
for(sum=0, offset = 0; offset <=
                last_element; offset++)
     sum += a_list[offset];
```

would be read:

> "Initialize the sum to zero and starting the offset at zero, accumulate the values in a list into sum until the offset reaches the last element."

Modern languages support for iterable collections support an implementation as:

```
current_sum=0
for each_element in a_list:
     current_sum += each_element
```

which reads:

> "After referencing the current sum to zero for each element in a list, set the current sum to reference the old reference plus the value of each element."

Python provides a built-in *sum* keyword which sums a collection. Granted, the Python statement:

```
current_sum = sum(a_list)
```

is simply read as:

> "The current sum is set to reference the sum of a list."

### C. Programming Language Dialects

The above examples hint at how different languages lead to distinct approaches to coding. Programmers skilled in a particular language utilize the idioms to express the algorithm constructs in an efficient manner. It has been the authors' experience that the LLMs are useful in generating code interactively as an enhanced version of code completion (MS VS IntelliSense or Github Copilot) and will generate appropriate constructs. While code generation tools [4], [5] provide a translation of an imperative algorithm into the language of choice, they do not necessarily adhere to the language's idiom.

## III. AI Code-Generated Example

The Sieve of Eratosthenes is used as an example of LLM-generated code in multiple languages compared to code which adheres to the language's idiom. The sieve simply *generates a list of primes from 2 to N by elimination of composite numbers from the list; the loop invariant at iteration k is a list of numbers which do not have the numbers 2 to k as a factor*. How this procedure is expressed in a specific language will vary is subject to the language constructs as below. While adherence to a language's idiom is not necessary, code which does not adhere to the languages' best practices can lead to unneeded constructs or duplicated effort.

### A. Python

In Fig. 1, the Pythonic version of the Sieve of Eratosthenes, once the prime candidates are set to the range of interest, the numbers in half the range are used to remove elements from the prime candidates which they divide.

Using Python as a shorthand, the statement starting on the third line in Fig. 1 would be read as:

> "For each iteration the prime candidates reference, the list of numbers which are possibly primes are defined to be the numbers which are the current divisor or have a non-zero remainder."

The loop invariant in the above code only contains elements which have not been eliminated and is immediately halved in size. This approach is very similar to implementations in other languages which would maintain a data structure (linked list) as the loop invariant [6].

```
# ellipses added for paper, code is single line
N = 100
prime_candidates = list(range(2,N))
for divisor in range(2,N//2):
    prime_candidates = ...
       [possible_prime for possible_prime in prime_candidates ...
         if possible_prime == divisor or possible_prime % divisor != 0]
print(prime_candidates)
```

Fig. 1. Hand-coded Pythonic version of the Sieve of Eratosthenes

```
def sieve_of_eratosthenes(n):
    primes = [True] * (n + 1)
    p = 2
    while p**2 <= n:
        if primes[p] == True:
            for i in range(p**2, n + 1, p):
                primes[i] = False
        p += 1
    return [p for p in range(2, n) if primes[p]]
```

Fig. 2. AI code-generated Sieve of Eratosthenes

A free AI code-generator [4] produced the code in Fig. 2. This code is very similar to imperative implementations. This loop invariant would be read as:

> For the first element in the remaining list of possible primes, mark all multiples of the element as false starting at the square of the prime."

Note that the AI-generated code constructs the list of primes at the end by scanning the array of flags. Almost identical code in Fig. 3 is generated by a different AI Python code generator [7].

Smart Code Writer [5] provides explanations of the code as in Fig. 4 (shown without the popup in Fig. 3). The explanations for the sieve closely match the loop invariant described for the code generated from the previous AI.

### B. C and Java

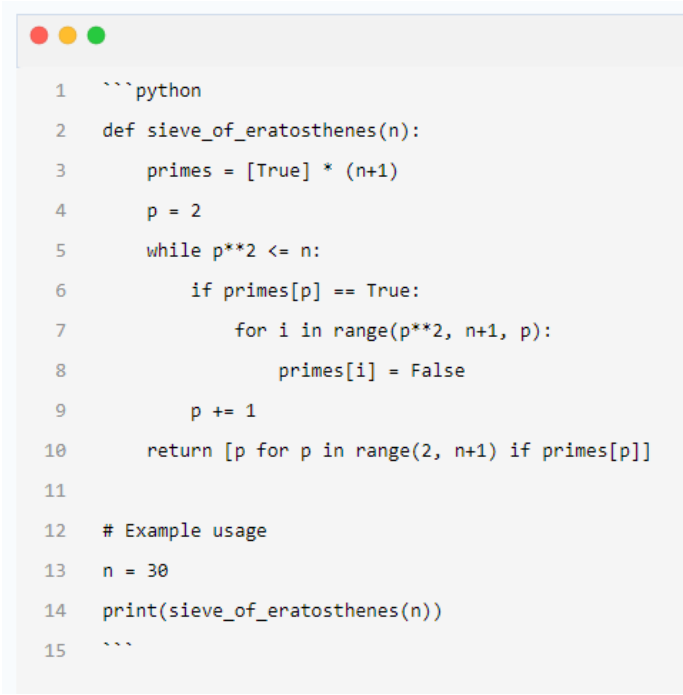
Using the same AI used to generate the Python code for C, the generated code in Fig. 5 for the sieve for C (or Java which is not shown but is left as an exercise to the reader) is very similar to the Python code [5].

Examining the code in Fig. 5, the AI-generated code reads:

> "The loop invariant checks the element at location p elements away from the labelled memory location prime and if it has not been eliminated from consideration (non-zero), sets the flag for all values in the list of primes to 0 if it is a multiple of the element."

## IV. AI Code-Generation as Enhanced Search

Given a target application which requires a particular algorithm, traditionally, a search of known code repositories for



```python
def sieve_of_eratosthenes(n):
    primes = [True] * (n+1)
    p = 2
    while p**2 <= n:
        if primes[p] == True:
            for i in range(p**2, n+1, p):
                primes[i] = False
        p += 1
    return [p for p in range(2, n+1) if primes[p]]

# Example usage
n = 30
print(sieve_of_eratosthenes(n))
```

Fig. 3. The Sieve provided by a secondary AI

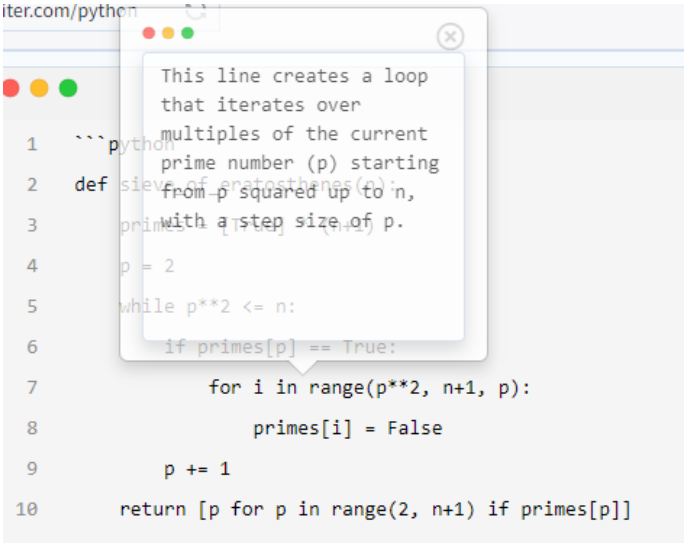
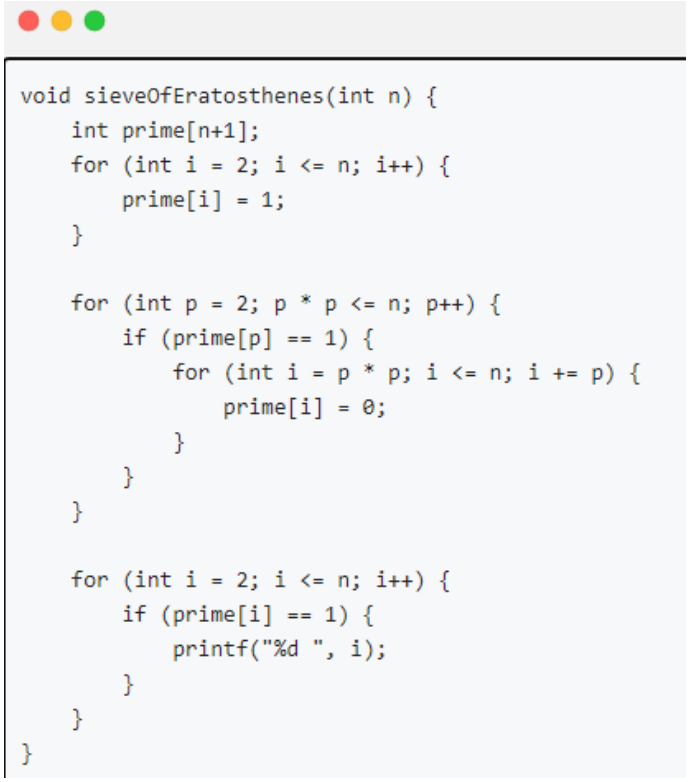



Fig. 4. Smart Code Writer explanations of code

reference code might be the first step of many students. To evaluate the appropriateness of the suggested code, the student must have a minimal ability to read code in the language of choice.

Alternatively, students may choose to use an LLM and a prompt to generate code. To generate reasonable responses to prompts, the LLMs have harvested existent code from online resources. The LLMs have been trained in publicly available references such as *The National Institute of Standards and Technology's Dictionary of Algorithms and Data Structures (DADS)* [8] and *Rosetta Code* [6]. The LLMs will be biased



```c
void sieveOfEratosthenes(int n) {
    int prime[n+1];
    for (int i = 2; i <= n; i++) {
        prime[i] = 1;
    }

    for (int p = 2; p * p <= n; p++) {
        if (prime[p] == 1) {
            for (int i = p * p; i <= n; i += p) {
                prime[i] = 0;
            }
        }
    }

    for (int i = 2; i <= n; i++) {
        if (prime[i] == 1) {
            printf("%d ", i);
        }
    }
}
```

Fig. 5. C code for the Sieve of Eratosthenes

to the most common approaches for a given prompt or the approach which is readily generated by direct translation from one language to another. Code which results from translation may not match the idiom (e.g. counted-for vs ranged-for) of the target language as shown above.

For the student to be able to evaluate generated code, the student needs to be able to read the code and assess if it matches the ideocracies of the desired language. This is a slight shift of focus from emphasizing the generation of control structures, which has been automated, to the interpretation of the presented code.

## V. Conclusion

Treating all code as a "shorthand" to be fully expanded when read is akin to literate programming provided the programmer is knowledgeable of the language's idiom. As the reader articulates statements, they are affectively untangling the code [1]. Furthermore, knowing what is idiomatic for the target language and what is "well written" will aid in the evaluation of AI-generated code. One can view the prompt as being "woven" [1] in with the AI-generated "tangled" code; however, if the tangled code is treated as a shorthand, a literate programmer articulates the shorthand into a complete description. Code generated from AI tools for a given prompt may provide the most common (and likely) implementation which may not be idiomatic of the target language. Using the AI interactively from enhanced code completion or by providing and fine-tuning a prompt benefits from a literate

programmer's ability to write code that reads well and adheres to the idiom.

## REFERENCES


[1] Donald E. Knuth (1984). "Literate Programming" (PDF). The Computer Journal. 27 (2). British Computer Society: 97–111. doi:10.1093/comjnl/27.2.97. Retrieved January 4, 2009.

[2] Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Pearson Education, 2009.

[3] "Literate Programming." Wikipedia, Wikimedia Foundation, 20 Apr. 2024, en.wikipedia.org/wiki/Literate_programming.

[4] "ZZZ Code AI." Zzz Projects, zzzcode.ai/. Accessed 13 May 2024.

[5] "AI-Powered Python Code Generator: The Ultimate Tool for Efficient Code Writing." SmartCodeWriter, 21 Apr. 2023, smartcodewriter.com/python/.

[6] "Sieve of Eratosthenes." Rosetta Code, Rosetta Code, 1 May 2024, rosettacode.org/wiki/Siev_of_Eratosthenes.

[7] Thomas H. Cormen, "Introduction to Algorithms", Fourth edition, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, The MIT Press, April 5, 2022.

[8] Sieve of Eratosthenes, xlinux.nist.gov/dads/HTML/sieve.html. Accessed 10 May 2024.