Theory, Language Choice, and Design to Solve Programming Problems

Anthony J. Coots

National University

TIM-8102: Principles of Computer Science

Dr. *******

May 11, 2025

Theory, Language Choice, and Design to Solve Programming Problems

Programming problems are often centered around three fundamental concepts: theoretical foundations, the choice of programming language, and applying both to develop solutions that meet system requirements for a given use case. Computer science theories, such as but not limited to Automata Theory and Type Theory, play an important role in how developers or programmers approach code and problem-solving.

This paper explores the meshing between theory and practice, and how these concepts inform language selection and program/software development. It lends a high-level overview of three programming languages used today, outlining additional criteria for choosing an apt language, and applies such principles to a theoretical approach to a taxable order calculator sample.

Theoretical Foundations and Relationship to Programming and Applications

Automata Theory

Automata theory is a mathematical discipline concerned with abstract machines, automata, and the types of problems they can solve. These machines serve as symbolic models of information processing systems, such as computers, rather than physical robots or mechanical devices (Reilly, Ralston, & Hemmendinger, 2003). At its core, an automaton reads an input string from a finite alphabet, typically including a set containing 0 and 1. It either computes a result or recognizes whether the input belongs to a particular set of valid strings.

A core application of Automata Theory in computer science is in lexical analysis, which is a stage of compilation in modern programming languages. Lexical analyzers use finite automata, which operate without memory beyond their current state, to scan and tokenize source code to identify elements known as keywords and operators.

Another example closer to everyday use is the automata as safety-critical systems in aviation or medical software, where state machines must avoid dangerous or undefined states, such as total failure involving a commercial airliner or malfunction in an insulin pump. Modeling allowable transitions between states explicitly prevents unintended behaviors.

Type Theory

Type theory is a formal system developed to resolve existing logical contradictions, such as Russell's Paradox, which states "the set of all sets that do not contain themselves" (Type Theory, 2009). In computing terms, this resembles writing a function that calls itself in a manner that may cause infinite recursion or a system crash. The theory structures how to prevent this by assigning every entity to a specific type while restricting operations to only valid combinations of the kinds to create a structured hierarchy where higher-order constructs do not interact with lower-order constructs in paradoxical ways.

Explicitly related to programming, type checkers utilize the principles of this theory at compile time to establish correctness and consistency. This process verifies that variables, functions, and expressions conform to expected types, preventing common errors such as type mismatches and null reference exceptions. In strongly typed languages, or languages significantly abstracted from the machine, such as Java or Rust, compile-time checks significantly reduce runtime errors; thus, this theory plays a critical role in modern software reliability and language design.

Programming Languages Overview

Python

Python is a third-generation language designed to be closer to natural human language than earlier-generation languages such as assembly or machine code. Unlike lower-level

languages, Python offers a higher degree of abstraction from hardware, enabling programmers and developers to focus on logical structure rather than low-level implementation. It is widely used for automation, data science, and artificial intelligence tasks.

Geo et al. (2023) demonstrated Python's utility in addressing computationally intensive problems by applying the Strassen Algorithm, an older yet robust mathematical method for matrix multiplication. The study evaluates the performance of this algorithm across three Python environments: IDLE, Jupyter Notebook, and Google Colab, since the best means of performance for linear algebra in computer science remains an open-ended question. The findings illustrate Python's ability to support advanced mathematical operations relevant to fields such as computer graphics, physics, and engineering, where linear algebra and matrix operations are fundamental. Additionally, they show how the language is effective in performance benchmarking and algorithmic analysis across different platforms.

C# (C-Sharp)

C# is a high-level language developed by Microsoft and is commonly used for creating enterprise and desktop applications, particularly in networked environments where security, scalability, and maintainability are priorities. Pashynskykh et al. (2022) explored C#'s capabilities by building cybersecurity analysis software for local computer networks and computer-integrated systems. Their application, developed in C# and .NET (a common tool used with C#, also developed by Microsoft), performs tasks such as scanning device ports, retrieving MAC addresses, and identifying vulnerabilities, all of which are routine for network auditing. Ultimately, their study demonstrates C#'s ability to address security-oriented programming in environments that require reliable access to system-level resources and multithreading capabilities.

Assembly

Though Assembly is a low-level programming language and more challenging to write and maintain due to its proximity to hardware, meaning it is less abstracted from the hardware, it offers greater efficiency and execution speed than other languages, such as C or Rust, overall. This is valuable in contexts where performance is paramount, like systems programming, embedded systems, and algorithm optimization. Unlike higher-level languages, Assembly provides near-immediate access to processor instructions.

Mafteiu-Scai et al. (2024) conducted a performance comparison of Assembly with C and Rust in implementing machine learning algorithms such as Linear Regression and k-Nearest Neighbors. Using public datasets for cancer and diabetes detection, they found that Assembly outperformed C by an average factor of 4 and Rust by a factor of 20 in execution time. While Assembly sacrifices development ease due to the exchange in interpretability, the study shows that its value lies in high-performance computing, where runtime efficiency is paramount.

Criteria for Language Selection

Selecting the programming language fit for a project involves balancing many factors, sometimes highlighting a few while disregarding others. These factors include scalability, performance, security, use case complexity, development lifecycle, exception handling, readability, writability, and portability. These factors often compete; for example, higher performance might compromise readability, or a language's scalability features may lead to longer development cycles.

Assembly language, a language close to the hardware, generally delivers superior performance but requires more effort to write and debug, which can affect scalability and

maintainability. Conversely, high-level languages like Python may suffer in execution speed in exchange for rapid development, readability, and access to extensive libraries.

Scalability is program-critical when applications must accommodate a growing user base or expanding data volumes. In enterprise settings, languages like C#, usually in tandem with the .NET ecosystem (both are Microsoft developments), support scalable architecture through memory management, multithreading, and other robust tooling. Low-level languages like Assembly can be optimized for speed. Still, it is often ill-advised for scalable program development due to the complexity and lack of abstraction from the hardware level.

Performance remains a concern in programs where speed is a priority. Mafteiu-Scai et al. (2024) show that Assembly can significantly outperform higher-level languages like Rust and C in machine learning tasks; a specific example involves using linear regression and k-nearest neighbors algorithms. However, the time and expertise required to write efficient Assembly code outweigh the benefits in many use cases, considering that higher-level alternatives, like Python, have optimized libraries for such.

Security considerations often influence programming language selections in domains involving airspace or healthcare, where the integrity of a system is non-negotiable. Yu and Duan (2022) illustrate how middle and low-level languages like C++ and Assembly can be exploited to reverse engineer and manipulate real-time applications such as 3D video games. In languages that offer memory safety features, such as Rust or Java, it may be preferable to mitigate any vulnerabilities.

Use case complexity also plays a decisive role, as some languages are purpose-built for specific domains (e.g., R, SPSS for statistical computing). In contrast, others are general-purpose

(e.g., Python, C-based). A mismatch between language capabilities and problem complexity can lead to inefficiencies or unmanageable codebases.

If long-term maintenance is necessary and community support is available, these languages may be considered for the development lifecycle. Exception handling is needed to build robust software in Java and Python to offer more fault-tolerant code. In contrast, languages like C may require more manual error checking, increasing oversight risk.

Readability and writability describe how easily a programmer or developer can read and write code. For instance, Python is known for its cleaner and more concise syntax. In contrast, Assembly is significantly harder to read and write due to its proximity to the hardware. Portability refers to the ease with which code can be executed across multiple platforms. Thanks to its Java Virtual Machine platform, languages like Java are designed for this purpose. Conversely, low-level languages require significant adaptation because they rely on specific hardware. No single language meets all the criteria mentioned. The choice depends on understanding the system or program's priorities and expectations.

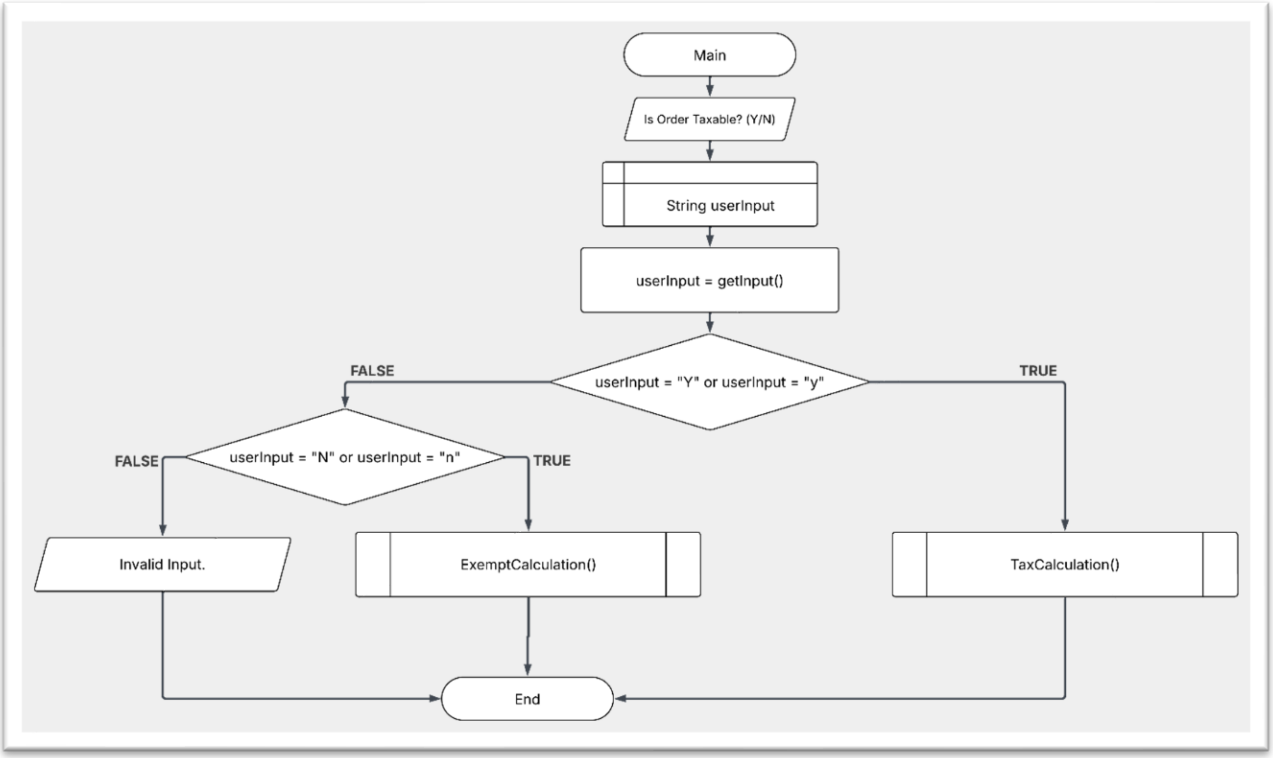
Application to Use Case

The following diagrams demonstrate the use case of a programming language to depict algorithms and logic as pseudocode to address the following:

- A main method for a taxable order based on user input, calling the appropriate function.
- A function to calculate the dollar amount of the order with tax and return the result to the main method.
- A function to calculate the dollar amount of the order without tax and return the result to the main method.
- A function to display the result using natural language in a displayed message.

Diagram 1

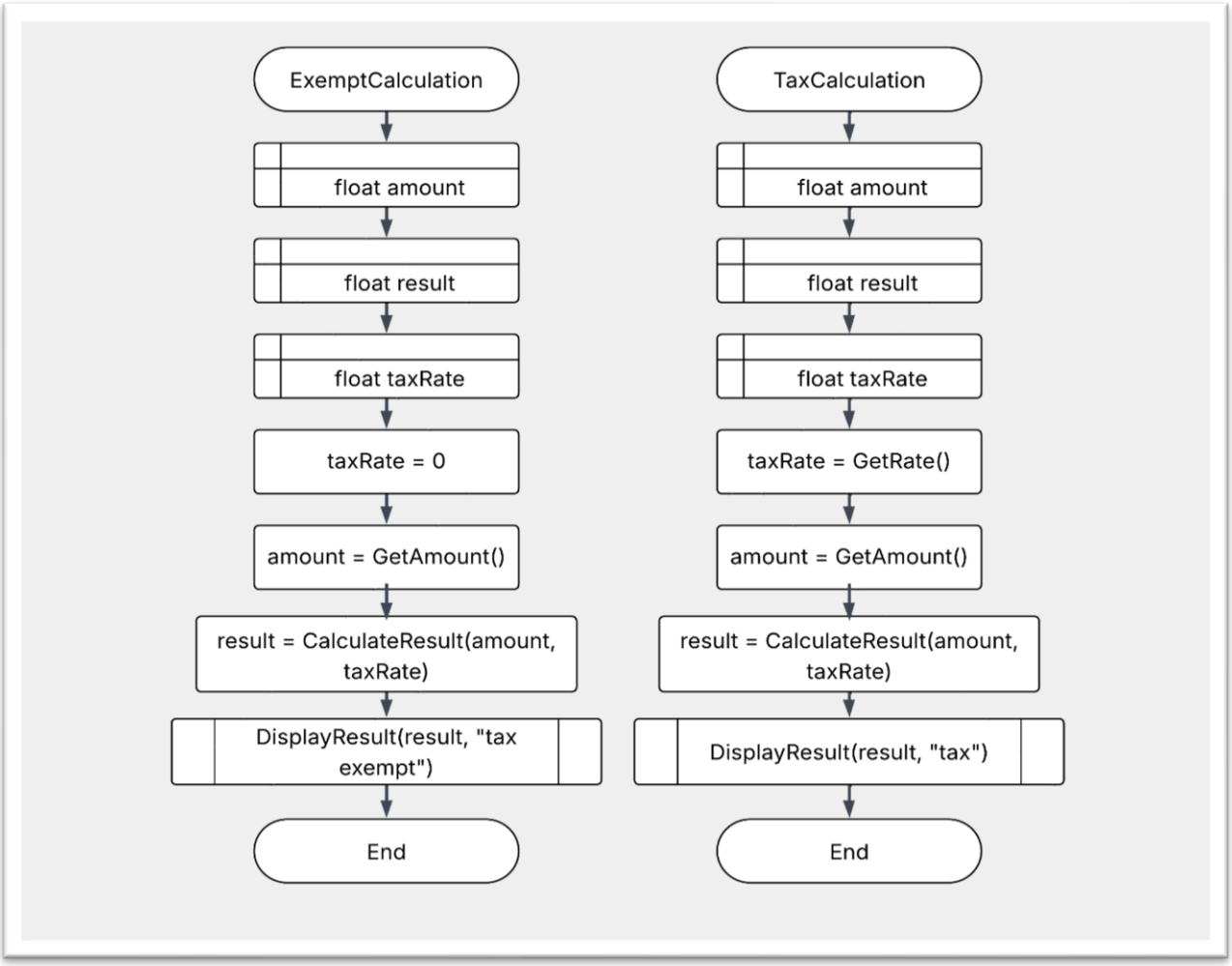
Main Method.



Note. The main method starts by printing a scenario-appropriate question, internally stores the user's input, and a dedicated function will be called if the input is appropriate. The result will be displayed via another function called in the calculation functions. Braunschweig and Busbee (n.d.) inspire this example with their temperature example.

Diagram 2

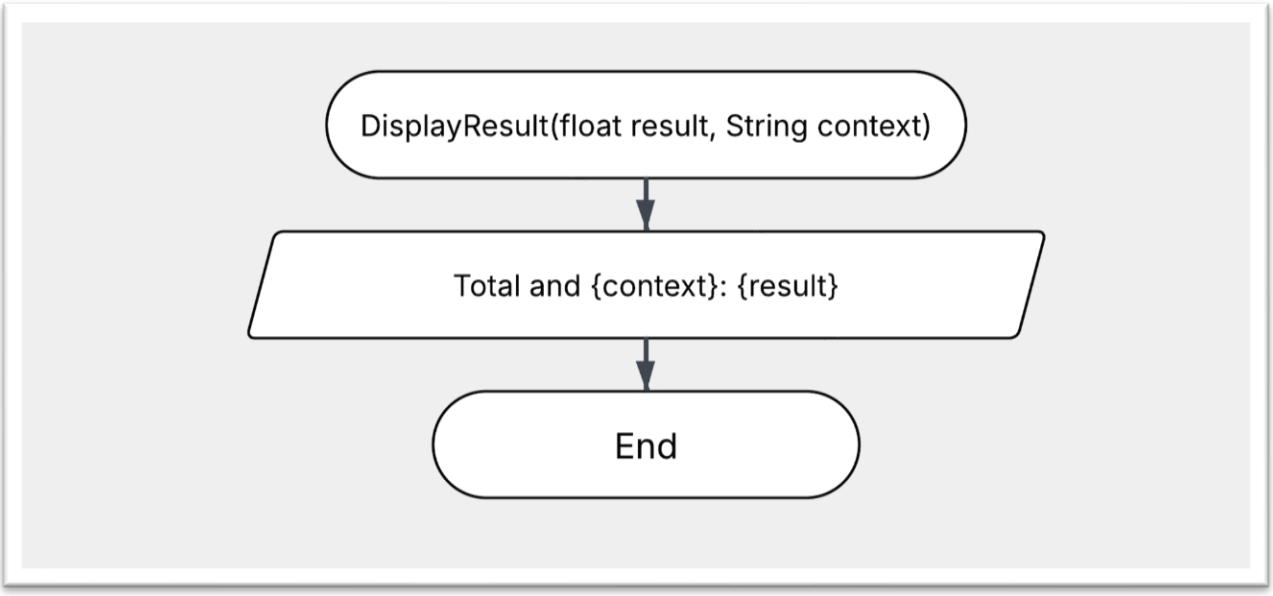
Order with/without Tax Functions.



Note. ExemptCalculation and TaxCalculation are similar functions, defining a dollar variable, a result, and a precise tax rate. If exempt, the result uses a tax rate of 0; otherwise, it is retrieved in another function not defined in this paper. The result is set in another function that is not defined in this paper. The result is then displayed in natural language and shows the value in the message (see Diagram 3). Braunschweig and Busbee (n.d.) inspire this example with their temperature example.

Diagram 3

Display Result Function.



Note. The function requires the result of the calling function and the context. The display omits a symbol such as '$' as this is not the only currency. The display will output "Total and tax: ##.##" or "Total and tax exempt: ##.##". Braunschweig and Busbee (n.d.) inspire this example with their temperature example.

Conclusion

Understanding the interconnectedness of computer science theory, programming languages, and real-world applications encompasses effective development. No single language meets all criteria, and choosing the appropriate programming language, when guided by theory and practical needs, allows developers to create systems as intended. The example of the taxable order calculator illustrates how to produce precise and reliable solutions.

References

Braunschweig, D., & Busbee, K. L. (n.d.). Programming fundamentals – A modular structured approach (2nd ed.). https://press.rebus.community/programmingfundamentals/

Braunschweig, D., & Busbee, K. L. (n.d.). Programming fundamentals – A modular structured approach (2nd ed.). https://press.rebus.community/programmingfundamentals/chapter/condition-examples/

Gao, S., Gao, W., Malomo, O., Allagan, J. D., Eyob, E., & Su, J. (2023). Comparison and Applications of Multiplying 2 by 2 Matrices Using the Strassen Algorithm in Python IDLE, Jupyter Notebook, and Colab. *2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE), Computer Science, Computer Engineering, & Applied Computing (CSCE), 2023 Congress in, CSCE*, 750–755. https://doi.org/10.1109/CSCE60160.2023.00128

Mafteiu-Scai, L. O., Balutoiu, V. D., & Mafteiu-Scai, R. T. (2024). Improving the Performance of Machine Learning Algorithms by Using Assembly Language. *2024 26th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2024 26th International Symposium on, SYNASC*, 251–255. https://doi.org/10.1109/SYNASC65383.2024.00049

Reilly, E. D., Ralston, A., & Hemmendinger, D. (Eds.). (2003). Automata theory. In *Encyclopedia of Computer Science* (4th ed.). Wiley.

https://search.credoreference.com/articles/Qm9va0FydGljbGU6MTY2NTExMg==?aid=102577

Type theory. (2009). In R. T. Cook, *A Dictionary of Philosophical Logic* (1st ed.). Edinburgh University Press. https://search.credoreference.com/articles/Qm9va0FydGljbGU6MjQ5OTk3NA==?aid=102577

Vladyslav Pashynskykh, Yelyzaveta Meleshko, Mykola Yakymenko, Dmytro Bashchenko, & Roman Tkachuk. (2022). Research the Possibilities of the C# Programming Language for Creating Cybersecurity Analysis Software in Computer Networks and Computer-Integrated Systems. *Сучасні Інформаційні Системи*, *6*(2). https://doi.org/10.20998/2522-9052.2022.2.09

Yu, L., & Duan, Y. (2022). A Reverse Modification Method for Binary Code and Data. *Sensors (Basel, Switzerland)*, *22*(20). https://doi.org/10.3390/s22207714