iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 13 **Issue:** IX **Month of publication:** September 2025

DOI: https://doi.org/10.22214/ijraset.2025.74374

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue IX Sep 2025- Available at www.ijraset.com

A Deep Dive into Python Execution: CPython Bytecode, PVM, and Online Platforms

Gajanan Santosh Koleshwar
Assistant Professor, Department of Master of Computer Application, Maharashtra Institute of Technology, Chhatrapati Sambhaji Nagar -431007 (M.S,), India.

Abstract: Python's popularity in artificial intelligence, education, and software development is driven by its intuitive syntax and flexible execution model. Unlike traditional compiled languages, Python code is first transformed into bytecode and then interpreted bythe Python Virtual Machine (PVM), making it highly portable and adaptable. This paper explores the internal workings of CPython the reference implementation of Python, by examining how source code is compiled into bytecode,how the PVM executes these instructions, and how recent changes in Python's bytecode enhance performance. Additionally, the paper discusses the growth of online Python interpreters, which make coding accessible from any device without complex setup. By analyzing these technical processes, the paper aims to help learners and practitioners better understand how Python operates internally and how modern tools are shaping its use in education and development.
Keywords: Artificial Intelligence (AI), Operating System (OS), Python Virtual Machine (PVM), CPython, bytecode.

I. INTRODUCTION

The exponential growth of artificial intelligence (AI) driven software has increased the demand for programming languages that combine high-level abstraction with flexible execution. Python has become aleadinglanguagein this spacedueto its clearsyntax, comprehensivestandard library, and seamless integration with widely used AI and data science frameworks such as TensorFlow and PyTorch[1][6][7]. Its simplicity supports rapid prototyping, while its maturity enables stable, large-scale deployments, making it well suited for both research and production environments[2][3][4].

In addition to these technical strengths, Python has demonstrated remarkable popularity and sustainability over time. It has consistently ranked among the top three programming languages worldwide, as reported by TIOBE, IEEE Spectrum, and RedMonk figures broadly supported in the literature on Python's community and adoption[2][3][6]. These rankings reflect Python's strong developer community, academic relevance, and widespread use across domains[6][9]. Its active ecosystem and platform independence contribute to its long-term viability[1][2][9]. As AI systems grow in complexity, a deeper understanding of Python's internal execution model particularly the interpretation of bytecode and the role of the Python Virtual Machine (PVM) becomes essential for optimizing performance and runtime behavior [4][10].

II. TRADITIONAL COMPILATION PROCESS OVERVIEW

In conventional software development, source code is typically written in high-level languages such as C or C++ [2][12]. These languages must undergo a sequence of processingstages before execution. Once the code is written and verified to be free of errors, a compiler isused to translate the human-readable instructions into machine code a set of low-level instructions composed of binary digits (0s and 1s) that can be directly executed by the CPU. The output of this compilation process is known as the object code or executable program, and it is saved as an executable file on the system's secondary storage [11][13].

During compilation, the compiler also determines the entry-point address, i.e., the location of the first instruction to be executed. This address is embedded in the executable file. When the user initiates the program, the Operating System (OS) retrieves the executable file from disk, loads it into memory, and begins execution by transferring control to the address of the first instruction [11]. This model, while efficient and optimized for speed, results in platform-specific binaries and lacks the flexibility required by modern cross-platform and dynamic applications [11][13].

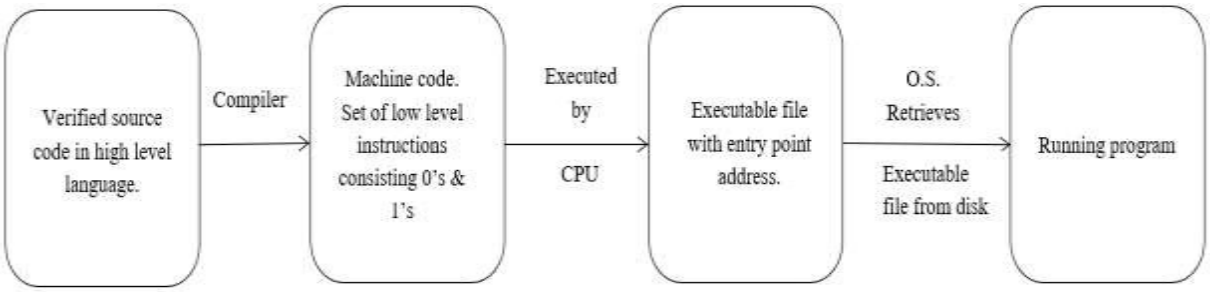


Figure1:TraditionalCompilationProcess,fromsourcecodetorunningprogram.

III. CPYTHON EXECUTION PROCESS

Python differs significantly in its execution process and might seem tricky. Python uses CPython which is the most widely used implementation of Python [2][4]. When we download Python from python.org we are getting CPython[2]. CPython is the default and most widely used implementation of Python, and it is designed to be cross-platform. This means we can run CPython onmajoroperatingsystemslikeWindows,macOS,andLinux,andevenonsomemobileplatforms, provided you have the appropriate interpreter installed [2][3]. CPython can be thought of as a system with two main components: a compiler and the Python Virtual Machine (PVM) [4][10]. Together, these components form what is known as the Python interpreter not just a compiler [10].

A. Bytecode Generation

Rather than compiling the source code into platform-dependent machine code, the Python interpreter first compiles it into an intermediate representation known as bytecode [2]. A common misconception about bytecode is that it consists of binary digits (0s and 1s), similar to machine code. However, in Python, bytecode is a set of instructions for the PVM and is typicallyrepresented in a more abstract, platform-independent format [2][12]. Python bytecode is a low- level, platform-independent intermediate representation of source code, specifically designed for execution by the PVM. The interpreter automatically generates this bytecode at runtime or during module importation, which helps optimize subsequent executions. These compiled instructions are typically stored in .pyc files (standard bytecode) or .pyo files (optimized bytecode), located within the pycachedirectory[2][10][12].

Internally, Python bytecode consists of a sequence of numeric opcodes (operation codes), with each instruction usually represented in a single byte[10]. Although all digital files are ultimately stored as binary data on disk, Python bytecode should not be confused with machine code [2][12]. Machine code is architecture-specific (e.g., x86, ARM) and executed directly by hardware, whereas Python bytecode is interpreted by the PVM [2][5][12].

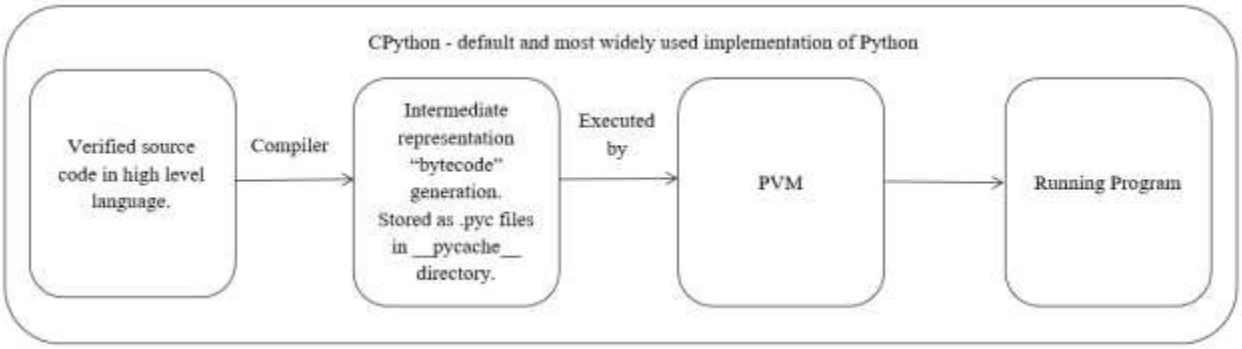


Figure2:ExecutionProcessinCPython-fromsourcecodetorunningprogramviabytecodeand PVM.

B. BytecodeStructure andInterpretation

Python provides several mechanisms to inspect its compiled bytecode, primarily for educational, debugging, and analytical purposes[4][10][14]. The most accessible and widely used among these is the built-in dis module, which disassembles code objects and presents a human readable sequence of low-level instructions representing the PVM operations [10][14]. In the following sections, we examine this bytecode through three different approaches to gain deeper insights into Python's execution process [2][10].

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue IX Sep 2025- Available at www.ijraset.com

IV. BYTECODE INSPECTION ANDANALYSIS

1) Using dis.dis()

The dis.dis() function from Python's dis module disassembles code objects into a human readable sequence of bytecode instructions [10]. It reveals opcodes, their arguments, andinstruction offsets. This aids in understanding how Python source code is translated and executedby the interpreter. It is especially useful for analyzing performance and debugging low-level behavior [2][4][10].

```
# 1. Disassembled Bytecode - Human-readable Bytecode using dis.dis()
import dis

def greet():
    print("Simple Example to see the Python's bytecode")

dis.dis(greet)
```

```
  4           0 RESUME                   0

  5           2 LOAD_GLOBAL              1 (NULL + print)
             14 LOAD_CONST               1 ("Simple Example to see the Python's bytecode")
             16 PRECALL                  1
             20 CALL                     1
             30 POP_TOP
             32 LOAD_CONST               0 (None)
             34 RETURN_VALUE
```

Figure3:Disassembledbytecodehumanreadablesequenceofbytecode.

2) Inspectingcode.co_code

The code.co_code attribute of a function object exposes the raw bytecode stream asabytestring,representingopcode–argumentpairsinbinaryform(e.g., \x..)[10].Thisbyte streamiswhatthePVMdirectlyinterpretsduringexecution.Unlikedis.dis(),itdoesnot disassemble the code but reveals the exact compiled form internally used by the interpreter[2][10].

```
# 2. Raw Bytecode Stream – Raw Bytecode (co_code as binary \x.. format)
def greet():
    print("Simple Example to see the Python's bytecode")
print(greet.__code__.co_code)
```

```
b'\x97\x00t\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00d\x01\xa6\x01\x00\x00\xab\x01\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00d\x00S\x00'
```

Figure4:RawBytecodeStream -PVMretrievesandinterpretsthesebytecodeinstructionsfrommemoryorfrom.pyc files.

3) Using dis.get_instructions()

The dis.get_instructions() function provides a structured and iterable view of a code object's bytecode [10]. Each instruction is represented as a named tuple containing the opcode name, numeric code, argument, resolved argument value, and byte offset. This detailed breakdown allows for precise inspection and programmatic analysis of Python'slow-level execution steps [8][10].

```
# 3. Structured Opcode - Inspection Structured Opcode Inspection using dis.get_instructions()
import dis
for instr in dis.get_instructions(greet):
    print(instr.opname, instr.opcode, instr.arg, instr.argval)
```

```
RESUME 151 0 0
LOAD_GLOBAL 116 1 print
LOAD_CONST 100 1 Simple Example to see the Python's bytecode
PRECALL 166 1 1
CALL 171 1 1
POP_TOP 1 None None
LOAD_CONST 100 0 None
RETURN_VALUE 83 None None
```

Figure5:StructureOpcodeInspection-detailedviewofeachinstruction.

4) KeyBytecodeInstructionsandTheirRole

In Python 3.11 and later, the CPython interpreter introduced a new set of bytecode instructions aimed at optimizing performance and improving the internal execution model. One notable addition is the RESUME instruction, which appears at the beginning of code objects. This instruction is used internally by the PVM to mark the start or resumption point of a function or frame, particularly in contexts such as coroutines, debuggers, or advanced control flow scenarios[14]. It helps manage the execution state of a frame more efficiently and is part of Python's adaptive interpreter improvements. Other essential bytecode instructions include LOAD_GLOBAL, LOAD_CONST, CALL, and RETURN_VALUE. In Python bytecode, the LOAD_GLOBAL instruction is responsible for retrieving a global variable such as a built-in function like print or a user defined global name and pushing it onto the evaluation stack. Itsearches the global and built-in namespaces and raises a NameError if the variable is not found[10]. The LOAD_CONST instruction is used to push constant values, such as strings or numbers, from the constant pool of a code object onto the operand stack. Together, these prepare the data required for function execution [10].

The CALL instruction, updated in Python 3.11, is now split into PRECALL and CALL. PRECALL sets up the function call with the appropriate number of arguments, while CALL executes the function using those arguments. This separation allows for better optimization and clearer handling of function invocation. Finally, the RETURN_VALUE instruction signifies theend of a function and returns the result to the caller by popping the top value from the stack. If no explicit return value is provided, it defaults to None[10]. These instructions, while low-level, form the backbone of Python's runtime execution model. Their evolution, particularly in recent Python versions, reflects ongoing efforts to enhance interpreter efficiency and provide better support for modern programming constructs[14].

5) Working ofPVM

In contrast to traditional compiled languages, Python bytecode is executed by the PVM, making it inherently portable and independent of specific hardware architectures[2][5][12]. Thetask of executing bytecode is handled entirely by the PVM, which is a core component of the Python runtime environment [5][10][14]. The PVM interprets bytecode instructions sequentially and performs the corresponding operations through software-based execution[10]. Effectively, the PVM serves as Python's execution engine, supporting critical features such as automatic memory management, dynamic typing, and exception handling [2][4]. This architecture enables Python to retain cross-platform and highly adaptable attributes that are especially beneficial in AI researchand deployment environments, where flexibility, rapid prototyping, and platform independence are often essential [1][6][7].

The PVM executes bytecode in the form of a sequential stream of opcode–argument pairs. Thisbytecodeistypicallystoredinthecode.co_codeattributeoffunctionormodule-level code objects [2][10]. During program execution or module importation, the PVM retrieves and interprets these bytecode instructions from memory or from .pyc files located in the pycachedirectory [2][10]. Internally, the PVM follows a stack-based architecture, using a runtimeevaluation stack to manage operands and intermediate values. Each bytecode instructioncorresponds to a predefined operation (opcode) such as loading a variable, performing arithmetic,or invoking a function. These instructions are processed one at a time through an internal loop,often referred to as the evaluation loop or main execution loop [5][14].

At runtime, the PVM reads each opcode from the bytecode stream and determines the corresponding operation handler, which is implemented in C in the CPython interpreter [10]. These handlers execute specific actions, such as pushing values onto the stack, accessing variables from local or global namespaces, or calling functions [10]. Each instruction modifies either the stack or the broader execution environment. The PVM, as mostly implemented in CPython, functions as a purely interpreted engine. Unlike Just-In-Time (JIT) compilers that convert bytecode into native machine code at runtime, the CPython PVM interprets platform-independent bytecode directly through software. This avoids the need for CPU-specific code generation and favors simplicity and portability over raw performance [2][4][14].

To maintain execution state, the PVM dynamically allocates memory for function call frames, object references, and namespace mappings (locals, globals, and built-ins) [10]. Since Python is a high-level, dynamically typed language, the interpreter performs real-time lookups and memorymanipulationsduringexecution[2][4].Consequently,uninterruptedaccesstosystem memory is essential for consistent and reliable performance. Once the bytecode stream has been fully interpreted, the result of execution depends on the nature of the program. If the code contains expressions or functions with return values, the result is returned to the caller or printed to the console if specified [10]. For statements with side effects such as I/O operations, variable assignments, or data manipulations, the PVM ensures that the intended changes take place as encoded in the bytecode [10].

In the case of function execution, the final result (if any) remains on top of the evaluation stack and is returned to the calling context [10].

For scripts or modules, the result is typically a series of side effects that affect the interpreter's global or local state, such as modified data structures or displayed outputs [10]. Importantly, the PVM does not produce standalone executable binaries or native machine code. Instead, the results of execution are ephemeral and context dependent, theymayexist as in-memorydata structures, printed output, returned values, or changes to the program's runtime state. The lifecycle of these results is governed by Python's memory management system, which relies on reference counting and garbage collection [2][4]. That's why, the result of PVM execution is not a compiled artifact, but rather the real-time behavior and observable effect of interpreting and executing Python bytecode within the runtime environment.

V. ONLINE PYTHON INTERPRETERS

Withthegrowingdemandforaccessible,platform-independentprogrammingenvironments, online compilers have emerged as essential tools for modern software development and education. These web-based interpreters and compilers allow users to write, compile, and execute codedirectly in a browser, eliminating the need for local development environment setup [6][10][15]. With the advancement of web technologies and the growing demand for accessible programming environments, Python development is no longer confined to local installations. A range of online platforms such as Replit, Google Colab, Jupyter Notebook (via JupyterHub), Trinket, and OnlineGDB allow users to write and execute Python code directlyin the browser without requiring anysetup. These platforms simulate Python's runtime behavior using cloud infrastructure, enabling real-time code execution on virtually any device [6][10][15]. In parallel, mobile applications suchas Pydroid3, QPython, and Coding Python bring Python development to smartphones and tablets. These apps either embed a native Python interpreter or connect to remote servers, offering portable and responsive coding environments. Together, web and mobile interpreters significantly extend Python's reach, particularly in education, rapid prototyping, and collaborative development scenarios [1][6][15][16].

1) TheNeedforOnlinePython Interpreters

Online Python interpreters have emerged in response to the increasing need for fast,flexible, and platform-independent development tools. Traditional setups often require installation of an interpreter, dependencies, and an IDE, causing barriers that can hinder beginners or users on restricted systems[2]. By providing ready-to-use environments accessible through a browser or mobile device, online interpreters eliminate these hurdles. Theyare especiallyuseful in educational settings such as MOOCs, coding bootcamps, and live classroom sessions, where learners can begin coding immediately[15]. Developers also use them to test small code snippets or collaborate on short term tasks. Moreover, in low resource contexts such as public computers, mobile devices, or locked-downsystems,onlineinterpretersprovideaviablealternativetolocaldevelopment.Theyarealsoincreasinglyusedincodinginterviewsandassessmentplatformstofacilitatereal-timecode evaluation[16]. In essence, online Python interpreters bridge the gap between accessibility and functionality, broadening participation in programming by minimizing technical entry barriers.

2) HowOnlinePythonInterpretersWork

Online Python interpreters allow users to interact with code through a web or mobile interface, typically following a client-server model. When code is entered in a browser, it is transmittedtoaremoteserverwhereabackendinterpreter(e.g.,CPythonor PyPy) executesit.This server operates within sandboxed environments such as containers or virtual machines to ensure isolation, security, and consistent resource allocation[17][18][19]. On mobile platforms, execution may be local or remote. Apps like Pydroid3 and QPython embed native Python interpreters, allowing offline use, whereas others connect to cloud servers for remote execution, similar to their web-based counterparts[20][21][22][23].

Some interpreters, such as Pyodide or Skulpt, execute code directly within the browserusing technologies like WebAssembly or JavaScript-based transpilation. This model eliminates server dependencies and enables offline functionality while still supporting real-time execution [23][24]. Increasingly, cloud-based Python environments also provide access to hardware accelerators such as Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs) to support high-performance computing workloads[25]. GPUs are well-suited for general-purpose parallel processing and are commonly used to accelerate libraries such as TensorFlow, PyTorch, and NumPy, particularly for tasks involving matrix computations, image processing, and machine learning[1][7][8][26]. TPUs, by contrast, are specialized for tensor-based operations and deliver optimal performance for large-scale deep learning training and inference, especially when using TensorFlow[27]. These accelerators are often integrated into online platforms such as GoogleColab and Kaggle, where users can request GPU, TPU-backed runtime environments on demand. This capability democratizes access to computational power, enabling advanced AI and datascience tasks to be performed from lightweight browser-based tools[19][20][28].

Regardless of the backend execution model local, remote, or in-browser the core behavior remains consistent: Python source code is compiled to bytecode and executed by the PVM[2][4][5]. Outputs such as print statements, error messages, and visualizations are rendered within the user interface, typically built using HTML, CSS, and JavaScript. Many platforms also offer enhanced functionality such as code saving, sharing, syntax highlighting, and basic debugging, making online Python interpreters effective tools for education, prototyping, and lightweight development[6][10][15][19].

3) UsageTrendsComparedtoTraditionalInterpreters

While professional developers continue to favor local interpreters and IDEs such as VS Code, PyCharm, or terminal environments for full-scale software development[2][10] the adoption of online Python interpreters has grown rapidly in educational and collaborative contexts. According to the GitHub Education Survey 2023, over 60% of coding bootcamps and online courses use browser-based interpreters to onboard beginners. These tools offer a seamless, installation-free experience that is particularly attractive for students and self-learners. Platforms like Replit and Google Colab serve millions of users daily. Replit alone reported over 20 million usersasof2024,underscoringthewidespreaduseofbrowser-basedcodingenvironments.Online interpreters arenow commonlyemployed in educational assignments, competitive coding, machine learning notebooks, and collaborative development tasks [7][15]. Globally, online Python interpreters are estimated to contribute to around 10–20% of overall Python usage. This estimate is supported by evidence from the growing adoption of scalable, cloud based Python IDEs especially in educational settings where these platforms support large numbers of students and enable automated assessments, real-time code execution, and features like syntax highlighting [29]. Their appeal lies in theiraccessibility, deviceindependence, and suitabilityforteaching, experimentation, and short-term development tasks.

4) AdvantagesandLimitationsofOnlinePythonInterpreters

OnlineinterpretersprovideimmediateaccesstoPythonprogrammingenvironmentswithout the need for installation or configuration. They can be accessed from any internet-enabled device, making them highly suitable for learning, quick testing, remote collaboration, and use on restricted systems [16][17][19]. Their simplicity and flexibility have made them especially popular in education,prototypingand mobile development [7][20].

However, these platforms come with certain constraints. Access to system-level features, external files, and long-running processes is typically restricted. Execution performance may be affected by network latency, and many platforms require constant internet access [19].Furthermore, some interpreters may not support all third-party libraries or advanced development workflows, limiting their use for complex or production-grade applications [10][17].

5) FutureScopeofOnlinePythonInterpreters

The future of online Python interpreters is promising, driven by advancements in web technologies, containerization and cloud computing [17][23]. We can expect more powerful,secure, and feature-rich platforms, including AI-assisted coding, enhanced debugging capabilities, and support for multi-language and full-stack development workflows [19][20]. Technologies like WebAssembly(e.g., Pyodide)mayfurther bridge the gap between local and online development by enabling browser-based execution even in offline scenarios [23]. These tools will be particularly impactful in under resourced or remote educational settings, where ease of access is crucial [16][17]. Continued improvements in containerized execution and user customization are likely to expand the functionality of online platforms. However, for complex projects, high-performance computing, or production environments, local interpreters and fully-featured IDEs will remain essential due to their superior control, performance, and debugging features [2][10].

VI. CONCLUSION

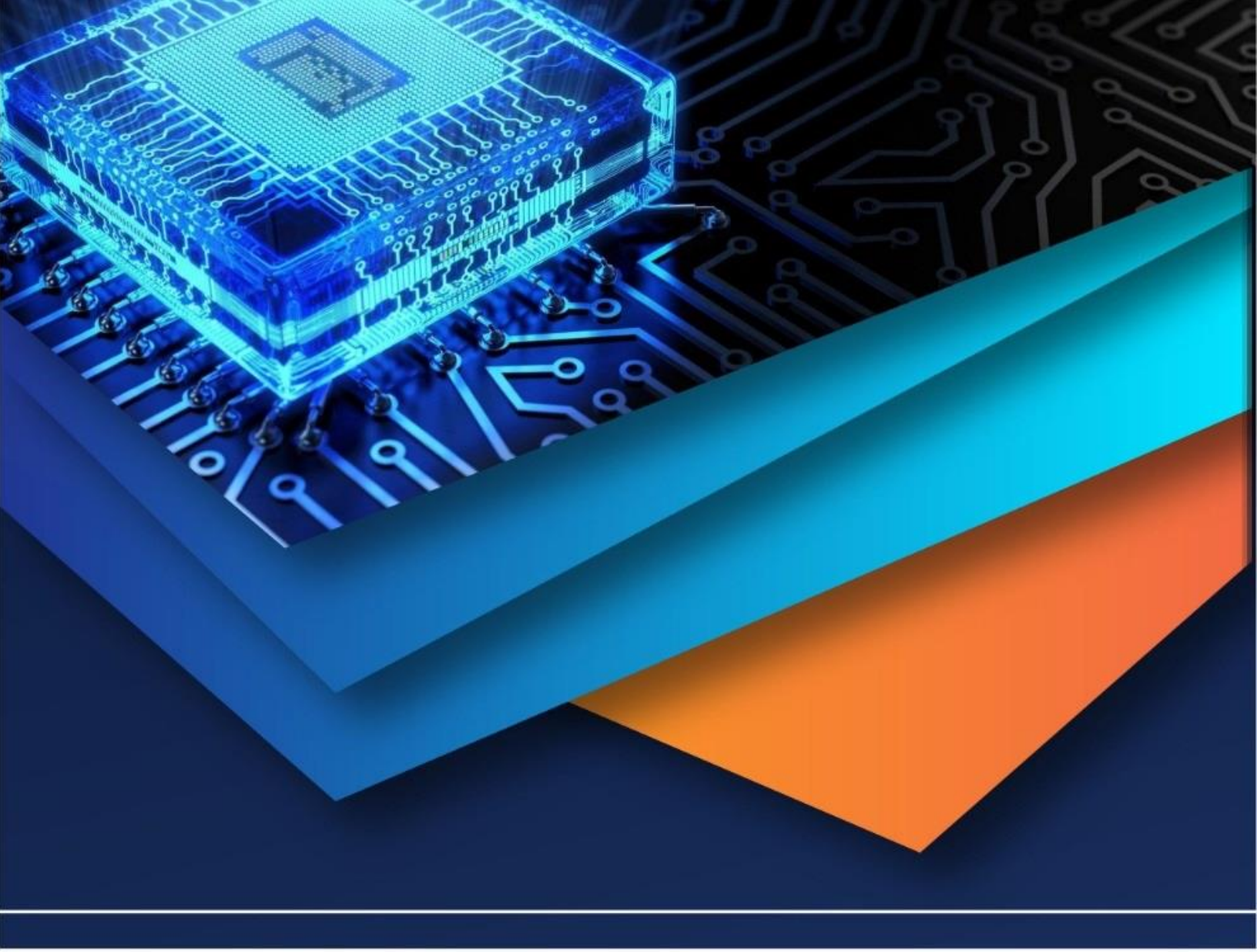
The execution model of Python, particularly through its CPython implementation,highlights the language's focus on portability, dynamism, and runtime flexibility. By compiling high-level source code into platform-independent bytecode and executing it via the PVM, Python enables rapid development and cross-platform compatibility. The disassembly and bytecode inspection tools discussed in this study such as dis, co_code, and get_instructions() offer valuable insight into Python's internal workings, bridging the gap between abstraction and execution.

In parallel, the growth of online Python interpreters has expanded the accessibility of programming environments, removing traditional barriers like installation and setup.

Theseplatformsareespeciallybeneficialineducationalcontexts,smallscalesoftwaredevelopment,and collaborative workflows. While they face limitations related to system access, performance, and extensibility, their role in democratizing programming remains significant. Looking ahead, advances in web technologies, cloud infrastructure, and AI integration are likely to enhance the functionality of online interpreters, making them an increasingly practical option for broader development scenarios. This paper thus connects Python's internal execution mechanisms with the evolving landscape of its runtime platforms, offering both technical insight and practical relevance for learners, educators, and developers alike.

REFERENCES

[1] T.E.Oliphant,"PythonforScientificComputing,"ComputinginScience&Engineering,vol. 9, no. 3, pp. 10–20, May/Jun. 2007. DOI: 10.1109/MCSE.2007.58.
[2] M.Lutz, LearningPython,5th ed.,Sebastopol,CA,USA:O'ReillyMedia,2013.
[3] L.Hetland,PythonAlgorithms:MasteringBasicAlgorithmsin thePythonLanguage,2nded., Berkeley, CA, USA: Apress, 2014.
[4] B.Slatkin,EffectivePython:90SpecificWaystoWriteBetterPython,2nded.,Boston,MA, USA: Addison-Wesley, 2019.
[5] G.vanRossumandJ.deBoer,"InteractivelyTestingRemoteServersUsingthePython Programming Language," CWI Quarterly, vol. 4, no. 4, pp. 283–303, 1991.
[6] J.VanderPlas,PythonDataScienceHandbook:EssentialToolsforWorkingwithData, Sebastopol, CA, USA: O'Reilly Media, 2016.
[7] I.Goodfellow,Y.Bengio,andA.Courville,DeepLearning,Cambridge,MA,USA:MIT Press, 2016.
[8] S.vanderWalt,S.C.Colbert,andG.Varoquaux,"TheNumPyArray:AStructurefor
[9] EfficientNumericalComputation,"ComputinginScience&Engineering,vol.13,no.2,pp.22–30, Mar./Apr. 2011. DOI: 10.1109/MCSE.2011.37.
[10] W.McKinney,PythonforDataAnalysis:DataWranglingwithPandas,NumPy,andIPython, 2nd ed., Sebastopol, CA, USA: O'Reilly Media, 2017.
[11] D.BeazleyandB.K.Jones,PythonCookbook:RecipesforMasteringPython3,3rded., Sebastopol, CA, USA: O'Reilly Media, 2013.
[12] A.Godbole,DemystifyingComputers,NewDelhi,India:McGrawHillEducation,2013,ch. 9.1.
[13] A.V.Aho,M.S.Lam,R.Sethi,andJ.D.Ullman,Compilers:Principles,Techniques,and Tools, 2nd ed., Boston, MA, USA: Addison-Wesley, 2006.
[14] C.GhezziandM.Jazayeri,ProgrammingLanguageConcepts,3rded.,Wiley,2002.
[15] J.Wiedemeieretal.,"PYLINGUAL:TowardPerfectDecompilationofEvolvingHigh- Level Languages," unpublished.
[16] C.Cooketal.,"BenefitsandPitfallsofJupyterNotebooksintheClassroom,"ACM Transactions on Computing Education, 2020.
[17] M.Arayaetal.,"JOVIAL:Notebook-basedAstronomicalDataAnalysisintheCloud," arXiv, 2018. arXiv:1805.00000.
[18] S.Ghoshetal.,"EnhancingPython-basedCloudComputingEducationUsingJupyterHub and BinderHub," in Proc. 2020 ACM Conf. Innovation and Technology in Computer Science Education, 2020. DOI: 10.1145/3341525.3387397.
[19] A.Rule,A.Tabard,andJ.D.Hollan,"ExplorationandExplanationinComputational Notebooks," in Proc. 2018 CHI Conf. Human Factors in Computing Systems, 2018. DOI: 10.1145/3173574.3173606.
[20] B.E.GrangerandF.Pérez,"Jupyter:ThinkingandStorytellingWithCodeandData," Journal of Open Source Software, vol. 6, no. 62, 2021. DOI: 10.21105/joss.03014.
[21] J.M.Perkel,"WhyJupyterisDataScientists'ComputationalNotebookofChoice,"Nature, vol. 563, pp. 145–147, 2018. DOI: 10.1038/d41586-018-07196-1.
[22] A.Headetal.,"ManagingMessesinComputationalNotebooks,"inProc.2019CHIConf. Human Factors in Computing Systems, 2019. DOI: 10.1145/3290605.3300500.
[23] F. Pérez, B. E. Granger, et al., "The Jupyter Notebook as a Tool for Open Science," in Positioningand Powerin AcademicPublishing:Players,AgentsandAgendas,Springer,2016.
[24] M.Meieretal.,"pyodide:BringingthePythonRuntimetotheBrowserviaWebAssembly," arXiv, 2021. arXiv:2107.11408.
[25] M.Abadietal.,"WebAssemblyforCloud-NativePython:ExperienceswithPyodideand Beyond," arXiv, 2023. arXiv:2304.06789.
[26] B.Karreretal.,"AcceleratedPythonintheCloud:UseofGPUsandTPUsinJupyter Environments," in Proc. 2022 IEEE Int. Conf. Cloud Engineering, 2022. DOI: 10.1109/IC2E53769.2022.00019.
[27] A.Paszkeetal.,"PyTorch:AnImperativeStyle,High-PerformanceDeepLearningLibrary," in Advances in Neural Information Processing Systems, vol. 32, pp. 8024–8035, 2019.
[28] N.P.Jouppietal.,"In-DatacenterPerformanceAnalysisofaTensorProcessingUnit,"in Proc. 44th Annual Int. Symp. Computer Architecture, pp. 1–12, 2017. DOI: 10.1145/3079856.3080246.
[29] A.Bisong,BuildingMachineLearningandDeepLearningModelsonGoogleCloud Platform: A Comprehensive Guide for Beginners, Berkeley, CA, USA: Apress, 2019.
[30] E.Frankfordetal.,"AnOnlineIntegratedDevelopmentEnvironmentforAutomated Programming Assessment Systems," arXiv, 2025. arXiv:2503.13127.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY