# python is a compiled language

**Understanding Why Python Is a Compiled Language: A Deep Dive**

**python is a compiled language** might sound surprising to many developers and enthusiasts who have long heard about Python as an interpreted language. This common perception has led to some confusion around how Python actually executes code under the hood. In this article, we will unravel the mystery behind Python's execution model, explore what it means to be a compiled language, and discuss how Python blends the worlds of compilation and interpretation. Along the way, we'll also touch on related concepts such as bytecode, just-in-time (JIT) compilation, and the role of Python interpreters.

## What Does It Mean for a Language to Be Compiled?

Before diving into Python specifically, it’s essential to clarify what "compiled language" means in the programming world. Traditionally, compiled languages are those where source code is transformed into machine code before execution. This machine code is a low-level, highly optimized set of instructions that the CPU can run directly. Examples include C, C++, and Rust.

On the other hand, interpreted languages execute instructions directly, without a separate compilation step. The interpreter reads the source code line-by-line and runs it on the fly. Classic examples of interpreted languages include JavaScript and early versions of BASIC.

However, this clear-cut distinction has blurred over time, especially with languages like Python that use intermediate steps involving compilation and interpretation together.

## Python Is a Compiled Language: Breaking Down the Process

When we say **python is a compiled language**, we're referring to the fact that Python source code is first compiled into an intermediate form called bytecode before execution. This bytecode is a low-level set of instructions designed for the Python Virtual Machine (PVM), a key component of Python’s runtime environment.

### From Source Code to Bytecode

Here’s what happens when you run a Python script:

1. **Parsing:** The Python interpreter reads the source code and parses it to check for syntax errors.

2. **Compilation to Bytecode:** If the code is syntactically correct, Python compiles it into bytecode, which is a platform-independent, intermediate representation.
3. **Execution by the Python Virtual Machine:** The PVM interprets this bytecode, executing it step-by-step.

This compilation to bytecode happens automatically and behind the scenes. You can actually see the compiled bytecode files (.pyc files) in Python's `__pycache__` directory after running a script.

## Why Compile to Bytecode?

The compilation step is crucial because it allows Python to optimize execution and reuse compiled code. Bytecode is more compact and faster to execute than source code, and since it's platform-independent, Python programs can run on any system with a compatible PVM.

This intermediate compilation stage differentiates Python from purely interpreted languages and justifies the claim that Python is a compiled language in a broader sense.

# The Role of the Python Interpreter and Virtual Machine

Understanding the Python interpreter's role helps clarify the hybrid nature of Python's execution model.

## Interpreter vs. Compiler: Python's Hybrid Approach

Python's interpreter is responsible for both compiling source code to bytecode and interpreting that bytecode. This contrasts with languages like C, where compilation and execution are strictly separate phases.

Because Python's interpreter handles both steps, the language offers great flexibility and ease of use. Developers can execute code interactively, modify it on the fly, and run scripts without waiting for a separate compilation step.

## The Python Virtual Machine (PVM)

The PVM is the runtime engine that executes the bytecode. Think of it as a specialized interpreter designed to understand Python bytecode instructions. The PVM handles memory management, method calls, and other runtime details.

Since the PVM interprets the bytecode, Python is sometimes described as an interpreted

language. But remember, this interpretation happens after an initial compilation step — hence, Python is both compiled and interpreted.

# Advanced Compilation Techniques in Python

Python’s standard execution model involves bytecode compilation, but there are other ways to compile Python code that affect performance and deployment.

## Just-In-Time (JIT) Compilation

Some Python implementations, like PyPy, incorporate Just-In-Time compilation. JIT compilers translate bytecode into machine code at runtime, optimizing frequently executed code paths to speed up execution dramatically.

This approach means Python code is compiled to machine code on the fly, blending the benefits of interpretation (flexibility) with those of compilation (speed). PyPy’s success illustrates how Python’s compiled nature can be leveraged for better performance.

## Static Compilation and Tools

Tools like Cython allow Python developers to compile Python code into C extensions. This process translates Python code into C, which is then compiled into machine code. This not only improves execution speed but also enables integration with low-level libraries.

Other tools, such as Nuitka and PyInstaller, compile Python code into standalone executables or optimized binaries, further demonstrating Python’s compiled capabilities in real-world scenarios.

# Why Understanding That Python Is a Compiled Language Matters

Recognizing that **python is a compiled language** in its own right can help developers write more efficient code and appreciate the language’s design choices.

## Performance Implications

Knowing that Python compiles code to bytecode means you can leverage techniques like pre-compilation to speed up script startups or reduce runtime overhead. For example, distributing `.pyc` files can save users from recompiling on their machines.

Moreover, understanding bytecode allows deeper debugging and optimization, as developers can use tools like `dis` to inspect bytecode instructions and optimize hotspots.

## Portability and Compatibility

Since Python bytecode is platform-independent, Python programs enjoy a high degree of portability. This feature is critical for cross-platform development and deployment, especially in diverse environments like servers, desktops, and embedded systems.

## Security and Distribution

Compiled bytecode files can be distributed without exposing raw source code, offering a basic level of code obfuscation. While not foolproof, this approach helps protect intellectual property and reduces the risk of accidental code modification.

# Common Misconceptions About Python’s Compilation

Despite the facts, many still consider Python purely interpreted, which is an oversimplification.

## “Python Is Slow Because It’s Interpreted”

While interpretation does add overhead, Python’s bytecode compilation and the existence of JIT compilers mean Python can be faster than many purely interpreted languages. Performance bottlenecks often arise from algorithmic inefficiencies rather than the compilation model.

## “Python Doesn’t Need Compilation”

Although Python’s compilation happens automatically and transparently, it is a vital step. Without it, code execution would be slower and less efficient.

# Final Thoughts on Python’s Compiled Nature

The statement **python is a compiled language** reflects the nuanced reality of how Python executes code. Python combines the best of both worlds by compiling code into bytecode and then interpreting that bytecode within a virtual machine. This hybrid model

offers a unique balance of speed, flexibility, and portability that has contributed to Python’s widespread popularity.

Whether you’re a beginner trying to grasp Python’s internals or an experienced developer optimizing your applications, understanding the compilation process can empower you to write better, more efficient Python code. As Python continues to evolve, with advances like JIT compilation and static analysis tools, its nature as a compiled language will only become more pronounced and significant.

# Frequently Asked Questions

## Is Python a compiled language?

Python is primarily an interpreted language, but it also involves a compilation step where the source code is compiled into bytecode before execution by the Python virtual machine.

## How does Python's compilation process work?

Python source code is first compiled into bytecode (.pyc files), which is a low-level set of instructions executed by the Python interpreter (PVM). This compilation is automatic and happens behind the scenes.

## Does Python compile to machine code like C or C++?

No, Python does not compile directly to machine code. Instead, it compiles to bytecode, which is interpreted by the Python virtual machine, making it different from fully compiled languages like C or C++.

## What is the difference between compiled and interpreted languages in the context of Python?

Compiled languages translate source code directly into machine code before execution, while interpreted languages execute code line-by-line. Python is a hybrid: it compiles code to bytecode, then interprets that bytecode at runtime.

## Can Python be compiled to an executable binary?

Yes, tools like PyInstaller, cx_Freeze, and Nuitka can compile Python code into standalone executable binaries, but under the hood, they bundle the Python interpreter and bytecode rather than compiling to native machine code.

## What is bytecode in Python?

Bytecode is an intermediate, platform-independent representation of Python source code. It is generated by compiling the source code and executed by the Python virtual machine.

## Does compiling Python code improve performance?

Compiling Python to bytecode improves startup time compared to interpreting source code directly, but it does not significantly improve runtime performance, which depends on the interpreter and runtime optimizations.

## Are there versions of Python that are fully compiled?

Yes, implementations like Cython and PyPy can compile Python code to machine code or optimize execution, providing performance improvements over the standard CPython interpreter.

## Why do people sometimes say Python is an interpreted language if it involves compilation?

Because the compilation step in Python is automatic, transparent, and compiles to bytecode rather than machine code, Python is commonly described as interpreted, emphasizing that execution happens via a virtual machine rather than natively compiling to machine code.

# Additional Resources

Python is a Compiled Language: An In-Depth Examination of Its Execution Model

**python is a compiled language** — a statement that often sparks debate among developers, educators, and programming language enthusiasts. At first glance, this assertion may seem counterintuitive given Python's reputation as a quintessential interpreted language. However, the truth about Python's execution process is more nuanced and merits a thorough exploration. Understanding whether Python is compiled, interpreted, or somewhere in between is essential for grasping its performance characteristics, development workflow, and the underlying mechanisms that enable its flexibility.

# Understanding the Nature of Python's Execution

To dissect the claim that python is a compiled language, it is necessary to first clarify what compilation entails in contrast to interpretation. Traditionally, compiled languages like C or C++ transform source code into machine code via a compiler before execution, resulting in standalone executables optimized for a specific platform. Interpreted languages, such as JavaScript or early versions of BASIC, process source code line-by-line at runtime, without a prior conversion step, which can impact performance but enhances portability and dynamism.

Python occupies a unique space in this spectrum. When Python code runs, it undergoes a compilation phase, but not to native machine code. Instead, Python source code (.py files) is compiled into an intermediate form known as bytecode (.pyc files). This bytecode is a

low-level, platform-independent representation of the code, which the Python Virtual Machine (PVM) subsequently interprets and executes. This two-step process blurs the lines between traditional compiled and interpreted paradigms, making the blanket statement that python is a compiled language partially accurate but incomplete without context.

## The Role of Bytecode in Python’s Execution Model

Python’s compilation to bytecode is an automatic and integral process that happens behind the scenes whenever a script runs. This bytecode compilation is a performance optimization; by translating human-readable source into a more efficient form, Python reduces the overhead of parsing and interpreting code repeatedly. Bytecode files are stored in the __pycache__ directory, enabling faster startup times on subsequent executions.

Unlike machine code generated by traditional compilers, bytecode is not directly executed by the hardware. Instead, it is executed by the PVM, a software-based interpreter designed to read and execute bytecode instructions. This setup allows Python to maintain its high-level abstractions, cross-platform compatibility, and dynamic features such as runtime type checking and dynamic binding.

## Comparing Python with Fully Compiled Languages

The distinction between Python and fully compiled languages is significant when considering performance and deployment. Languages like C++ compile source code directly into machine code tailored for a specific operating system and processor architecture. This compilation results in highly optimized binaries, which often run faster and consume fewer resources.

Python’s bytecode compilation does not yield such native executables. Instead, the PVM adds an interpretation layer that introduces runtime overhead. This difference explains why Python generally runs slower compared to compiled languages but gains advantages in terms of flexibility, ease of debugging, and rapid prototyping.

# Is Python a Compiled Language? Exploring the Spectrum

The statement python is a compiled language can be explored through the lens of different Python implementations and their execution strategies. CPython, the standard and most widely used implementation, follows the bytecode compilation and interpretation model described above. However, alternative Python interpreters adopt different approaches that influence whether Python behaves more like a compiled or interpreted language.

## Alternative Python Implementations and Their Compilation Strategies

- **PyPy:** An implementation that includes a Just-In-Time (JIT) compiler, translating Python bytecode into machine code at runtime, which significantly improves execution speed.
- **Cython:** A superset of Python designed to generate C code from Python-like syntax. Cython compiles this C code into native binaries, bridging the gap between interpreted Python and compiled languages.
- **Jython:** A Python implementation for the Java Virtual Machine (JVM) that compiles Python code into Java bytecode, which is then executed by the JVM.
- **IronPython:** Targets the .NET framework, compiling Python code to Intermediate Language (IL), allowing integration with .NET assemblies and runtime.

These variations demonstrate that python is a compiled language in some contexts, depending on the implementation and target platform. The diversity of Python interpreters reflects the language’s adaptability and broad ecosystem.

## Performance Implications of Python’s Compilation Model

The compilation to bytecode and subsequent interpretation by the PVM influence Python’s runtime performance. While bytecode compilation reduces the cost of repeated parsing, the interpretation layer remains a bottleneck compared to native machine code execution. This trade-off affects applications that demand high computational efficiency, such as scientific computing, real-time systems, or game development.

To mitigate these limitations, developers often employ tools and techniques such as:

1. **Using Cython:** Compiling performance-critical modules to C to achieve near-native speeds.
2. **JIT Compilers:** Leveraging PyPy’s JIT to dynamically translate hot code paths into machine code at runtime.
3. **Extension Modules:** Integrating native extensions written in C or C++ to offload intensive computations.
4. **Multiprocessing and Asynchronous Programming:** Improving efficiency by parallelizing or optimizing I/O-bound tasks.

These strategies highlight that while python is a compiled language in the sense of producing bytecode, achieving high performance often requires transcending the base execution model.

# Implications for Developers and the Python Ecosystem

The hybrid nature of Python’s compilation and interpretation affects various aspects of development, from debugging to distribution.

## Debugging and Development Workflow

Because Python source code is compiled to bytecode but remains highly readable and dynamically typed, debugging remains straightforward. Developers can inspect source code directly, set breakpoints, and modify code on the fly without recompiling entire applications. This agility accelerates development cycles and supports Python’s popularity in prototyping, data science, and educational environments.

## Distribution and Deployment Considerations

Unlike fully compiled languages that produce standalone executables, Python programs typically require the Python interpreter and associated runtime environment for execution. The bytecode files (.pyc) improve load times but do not replace the need for the interpreter. Tools like PyInstaller and cx_Freeze exist to package Python applications into distributable bundles, bundling the interpreter and dependencies into a single executable for user convenience.

This packaging approach underscores that python is a compiled language only up to the bytecode stage, with runtime interpretation remaining necessary unless additional compilation steps are taken.

## Security and Obfuscation

Since Python’s bytecode is easier to decompile back into readable source code compared to native binaries, concerns arise regarding code confidentiality and intellectual property protection. Compiled languages generate machine code that is inherently more difficult to reverse-engineer. Developers seeking to protect their Python source often rely on obfuscation tools, encryption, or distributing compiled extensions to mitigate risks.

# Reconciling the Terminology: Python’s Place in the Programming Language Landscape

The debate over whether python is a compiled language reflects broader challenges in categorizing modern programming languages. Python’s design philosophy prioritizes readability, simplicity, and developer productivity, leveraging a hybrid execution strategy that balances performance and flexibility.

In essence, Python is a language that is compiled to bytecode, which is then interpreted. This model defies the binary classification of compiled versus interpreted, positioning Python as a bytecode-compiled interpreted language. Understanding this distinction is crucial for developers, educators, and decision-makers when evaluating Python’s suitability for specific projects or comparisons with other languages.

By embracing this nuanced perspective, the programming community can better appreciate Python’s unique strengths and limitations, fostering informed choices and innovative applications across diverse domains.

## Python Is A Compiled Language

**python is a compiled language:** *Recent Advances of the Fragment Molecular Orbital Method* Yuji Mochizuki, Shigenori Tanaka, Kaori Fukuzawa, 2021-01-04 This book covers recent advances of the fragment molecular orbital (FMO) method, consisting of 5 parts and a total of 30 chapters written by FMO experts. The FMO method is a promising way to calculate large-scale molecular systems such as proteins in a quantum mechanical framework. The highly efficient parallelism deserves being considered the principal advantage of FMO calculations. Additionally, the FMO method can be employed as an analysis tool by using the inter-fragment (pairwise) interaction energies, among others, and this feature has been utilized well in biophysical and pharmaceutical chemistry. In recent years, the methodological developments of FMO have been remarkable, and both reliability and applicability have been enhanced, in particular, for non-bio problems. The current trend of the parallel computing facility is of the many-core type, and adaptation to modern computer environments has been explored as well. In this book, a historical review of FMO and comparison to other methods are provided in Part I (two chapters) and major FMO programs (GAMESS-US, ABINIT-MP, PAICS and OpenFMO) are described in Part II (four chapters). dedicated to pharmaceutical activities (twelve chapters). A variety of new applications with methodological breakthroughs are introduced in Part IV (six chapters). Finally, computer and information science-oriented topics including massively parallel computation and machine learning are addressed in Part V (six chapters). Many color figures and illustrations are included. Readers can refer to this book in its entirety as a practical textbook of the FMO method or read only the chapters of greatest interest to them.

**python is a compiled language: Modeling, Simulation and Optimization of Complex Processes** Hans Georg Bock, Ekaterina Kostina, Xuan Phu Hoang, Rolf Rannacher, 2008-06-19 This proceedings volume covers the broad interdisciplinary spectrum of scientific computing and presents recent advances in theory, development of methods, and applications in practice.

**python is a compiled language:** *Python for Bioinformatics* Sebastian Bassi, 2016-04-19 Programming knowledge is often necessary for finding a solution to a biological problem. Based on

the author's experience working for an agricultural biotechnology company, Python for Bioinformatics helps scientists solve their biological problems by helping them understand the basics of programming. Requiring no prior knowledge of programming-related concepts, the book focuses on the easy-to-use, yet powerful, Python computer language. The book begins with a very basic introduction that teaches the principles of programming. It then introduces the Biopython package, which can be useful in solving life science problems. The next section covers sophisticated tools for bioinformatics, including relational database management systems and XML. The last part illustrates applications with source code, such as sequence manipulation, filtering vector contamination, calculating DNA melting temperature, parsing a genbank file, inferring splicing sites, and more. The appendices provide a wealth of supplementary information, including instructions for installing Python and Biopython and a Python language and style guide. By incorporating examples in biology as well as code fragments throughout, the author places a special emphasis on practice, encouraging readers to experiment with the code. He shows how to use Python and the Biopython package for building web applications, genomic annotation, data manipulation, and countless other applications.

**python is a compiled language: The Python-Based Laboratory** John Essick, 2024-12-26 The Python-Based Laboratory: A Hands-On Guide for Scientists and Engineers provides a learn-by-doing approach to acquiring the Python programming skills needed to implement computer-controlled experimental work. The book leads its readers to mastery of the popular, open-source Python computer language in its role as a powerful laboratory tool by carrying out interesting and relevant projects that explore the acquisition, production, analysis, and presentation of digitized waveforms. Readers, who are assumed to have no prior computer programming or Python background, begin writing meaningful programs in the first few pages. The Python-Based Laboratory can be used as a textbook for science and engineering instructional laboratory students who are being taught up-to-date Python-based experimental skills. The book also works well as a self-study guide for professional laboratory researchers, industrial engineers, hobbyists, and electronics enthusiasts seeking to automate tasks using Python. Topics covered include the control of data acquisition devices (including multifunction data acquisition hardware and IEEE-interfaced stand-alone instruments), data file storage and presentation, digitized data concepts (such as resolution, sampling frequency, and aliasing), and data analysis techniques (curve fitting and fast Fourier transform). As readers work their way through the book, they build several computer-based instruments, including a DC voltmeter, digital oscilloscope, DC voltage source, waveform generator, blinking LED array, digital thermometer, and spectrum analyzer. Each chapter concludes with a Do-It-Yourself project and a Use It! example as well as a healthy selection of homework-style problems, allowing readers to test their understanding and further develop their Python-based experimentation skills.

**python is a compiled language: Theoretical and Mathematical Foundations of Computer Science** Qihai Zhou, 2011-10-29 This book constitutes the refereed post-proceedings of the Second International Conference on Theoretical and Mathematical Foundations of Computer Science, ICTMF 2011, held in Singapore in May 2011. The conference was held together with the Second International Conference on High Performance Networking, Computing, and Communication systems, ICHCC 2011, which proceedings are published in CCIS 163. The 84 revised selected papers presented were carefully reviewed and selected for inclusion in the book. The topics covered range from computational science, engineering and technology to digital signal processing, and computational biology to game theory, and other related topices.

**python is a compiled language: Python for Scientists** John M. Stewart, 2017-07-20 Scientific Python is taught from scratch in this book via copious, downloadable, useful and adaptable code snippets. Everything the working scientist needs to know is covered, quickly providing researchers and research students with the skills to start using Python effectively.

**python is a compiled language:** A Practical Handbook of Corpus Linguistics Magali Paquot, Stefan Th. Gries, 2021-05-04 This handbook is a comprehensive practical resource on corpus linguistics. It features a range of basic and advanced approaches, methods and techniques in corpus

linguistics, from corpus compilation principles to quantitative data analyses. The Handbook is organized in six Parts. Parts I to III feature chapters that discuss key issues and the know-how related to various topics around corpus design, methods and corpus types. Parts IV-V aim to offer a user-friendly introduction to the quantitative analysis of corpus data: for each statistical technique discussed, chapters provide a practical guide with R and come with supplementary online material. Part VI focuses on how to write a corpus linguistic paper and how to meta-analyze corpus linguistic research. The volume can serve as a course book as well as for individual study. It will be an essential reading for students of corpus linguistics as well as experienced researchers who want to expand their knowledge of the field.

**python is a compiled language: Implementing Reproducible Research** Victoria Stodden, Friedrich Leisch, Roger D. Peng, 2018-12-14 In computational science, reproducibility requires that researchers make code and data available to others so that the data can be analyzed in a similar manner as in the original publication. Code must be available to be distributed, data must be accessible in a readable format, and a platform must be available for widely distributing the data and code. In addition, both data and code need to be licensed permissively enough so that others can reproduce the work without a substantial legal burden. Implementing Reproducible Research covers many of the elements necessary for conducting and distributing reproducible research. It explains how to accurately reproduce a scientific result. Divided into three parts, the book discusses the tools, practices, and dissemination platforms for ensuring reproducibility in computational science. It describes: Computational tools, such as Sweave, knitr, VisTrails, Sumatra, CDE, and the Declaratron system Open source practices, good programming practices, trends in open science, and the role of cloud computing in reproducible research Software and methodological platforms, including open source software packages, RunMyCode platform, and open access journals Each part presents contributions from leaders who have developed software and other products that have advanced the field. Supplementary material is available at www.ImplementingRR.org.

**python is a compiled language:** *Learning Python* Mark Lutz, 2025-02-25 Get a comprehensive, in-depth introduction to the core Python language with this hands-on book. Based on author Mark Lutz's popular training course, this updated sixth edition will help you quickly write efficient, high-quality code with Python. It's an ideal way to begin, whether you're new to programming or a professional developer versed in other languages. Complete with quizzes, exercises, and helpful illustrations, this easy-to-follow self-paced tutorial gets you started with Python 3.12 and all other releases in use today. With a pragmatic focus on what you need to know, it also introduces some advanced language features that have become increasingly common in Python code. This book helps you: Explore Python's built-in object types such as strings, lists, dictionaries, and files Create and process objects with Python statements, and learn Python's syntax model Use functions and functional programming to avoid redundancy and maximize reuse Organize code into larger components with modules and packages Code robust programs with Python's exception handling and development tools Apply object-oriented programming and classes to make code customizable Survey advanced Python tools including decorators, descriptors, and metaclasses Write idiomatic Python code that runs portably across a wide variety of platforms

**python is a compiled language:** *Parallel Processing and Applied Mathematics* Roman Wyrzykowski, Jack Dongarra, Ewa Deelman, Konrad Karczewski, 2018-03-22 The two-volume set LNCS 10777 and 10778 constitutes revised selected papers from the 12th International Conference on Parallel Processing and Applied Mathematics, PPAM 2017, held in Lublin, Poland, in September 2017. The 49 regular papers presented in the proceedings were selected from 98 submissions. For the workshops and special sessions, that were held as integral parts of the PPAM 2017 conference, a total of 51 papers was accepted from 75 submissions. The papers were organized in topical sections named as follows: Part I: numerical algorithms and parallel scientific computing; particle methods in simulations; task-based paradigm of parallel computing; GPU computing; parallel non-numerical algorithms; performance evaluation of parallel algorithms and applications; environments and frameworks for parallel/distributed/cloud computing; applications of parallel computing; soft

computing with applications; and special session on parallel matrix factorizations. Part II: workshop on models, algorithms and methodologies for hybrid parallelism in new HPC systems; workshop power and energy aspects of computations (PEAC 2017); workshop on scheduling for parallel computing (SPC 2017); workshop on language-based parallel programming models (WLPP 2017); workshop on PGAS programming; minisymposium on HPC applications in physical sciences; minisymposium on high performance computing interval methods; workshop on complex collective systems.

**python is a compiled language:** Programming Basics with C# Svetlin Nakov, Nakov's Team, 2019-09-01 The free book Programming Basics with C# (https://csharp-book.softuni.org) is a comprehensive entry level computer programming tutorial for absolute beginners that teaches basics of coding (variables and data, conditional statements, loops and methods), logical thinking and problem solving using the C# language. The book comes with free video lessons for each chapter, 150+ practical exercises with an automated online evaluation system (online judge) and solution guidelines for the exercises. The book Programming Basics with C# introduces the readers with writing programming code at a beginners level (basic coding skills), working with development environment (IDE), using variables and data, operators and expressions, working with the console (reading input data and printing output), using conditional statements (if, if-else, switch-case), loops (for, while, do-while, foreach) and methods (declaring and calling methods, passing parameters and returning values), as well as algorithmic thinking and solving practical programming problems. This free coding book for beginners is written by a team of developers lead by Dr. Svetlin Nakov (https://nakov.com) who has 25+ years practical software development experience and 15+ years as software development trainer. The free book Programming Basics with C# is an official textbook for the Programming Basics classes at the Software University (SoftUni), used by tens of thousands of students at the start of their software development education. The book relies on the explain by examples and learn by doing approaches to learning the practical coding skills required to become a software engineer. Each chapter provides some concepts, explained as video lesson with lots of code examples, followed by practical exercises involving the use of the new concepts with online evaluation system (online judge). Learners watch the videos, try the sample code and solve the exercises, which come as part of each book chapter. Exercises are given in series with increasing complexity: from quite trivial, though little complicated to highly complicated, requiring more thinking and research in Internet. Most exercises come with detailed hints and guidelines about how to construct a correct solution. Download the free C# programming basics book (as PDF, ePub and Mobi formats), watch the video lessons and the live coding demos, solve the practical exercises and evaluate your solutions at the book official Web site: https://csharp-book.softuni.org. Tags: book, programming, free, computer programming, coding, writing code, programming basics, ebook, programming book, book programming, C#, CSharp, C# book, Visual Studio, .NET, tutorial, C# tutorial, video lessons, C# videos, programming videos, programming lessons, coding lessons, coding videos, programming concepts, data types, variables, operators, expressions, calculations, statements, console input and output, control-flow logic, program logic, conditional statements, nested conditions, loops, nested loops, methods, functions, method parameters, method return values, problem solving, practical exercises, practical coding, learn by examples, learn by doing, code examples, online judge system, Nakov, Svetlin Nakov, SoftUni, ISBN 978-619-00-0902-3, ISBN 9786190009023 Detailed Book Contents: Preface - about the book, scope, how to learn programming, how to become a developer, authors team, SoftUni, the online judge, forums and other resources Chapter 1. First Steps in Programming - writing simple commands, writing simple computer programs, runtime environments, the C# language, Visual Studio and other IDEs, creating a console program, writing computer programs in C# using Visual Studio, building a simple GUI and Web apps in Visual Studio Chapter 2.1. Simple Calculations - using the system console, reading and printing integers, using data types and variables, reading floating-point numbers, using arithmetic operations, concatenating text and numbers, using numerical expressions, exercises with simple calculations, creating a simple GUI app for converting currencies Chapter 2.2. Simple Calculations –

Exam Problems - practical problems with console input / output and simple calculations, with solution guidelines, from programming basics exams Chapter 3.1. Simple Conditions - using simple conditional statements, comparing numbers, simple if-else conditions, variable scope, sequence of if-else conditions, using the debugger, practical exercises with simple conditions with solution guidelines Chapter 3.2. Simple Conditions – Exam Problems - practical problems with simple if-else conditions, with solution guidelines, from programming basics exams Chapter 4.1. More Complex Conditions - nested if conditions (if-else inside if-else), using the logical OR, AND and NOT operators, using the switch-case conditional statements, building GUI app for visualizing a point in a rectangle, practical exercises with solution guidelines Chapter 4.2. More Complex Conditions – Exam Problems - practical problems with more complex if-else conditions and nested if conditions, with solution guidelines, from programming basics exams Chapter 5.1. Repetitions (Loops) - using simple for-loops, iterating over the numbers from 1 to n, reading and processing sequences of numbers from the console, using the for-loop code snipped in Visual Studio, many practical exercises with loops, with solution guidelines, summing numbers, finding min / max element, drawing with the turtle graphics in a GUI app Chapter 5.2. Loops – Exam Problems - practical problems with simple loops, with solution guidelines, from programming basics exams Chapter 6.1. Nested Loops - using nested loops (loops inside other loops), implementing more complex logic with loops and conditional statements, printing simple and more complex 2D figures on the console using nested loops, calculations and if conditions, practical exercises with nested loops with solution guidelines, building a simple Web app to draw ratings in Visual Studio using ASP.NET MVC Chapter 6.2. Nested Loops – Exam Problems - practical problems with nested loops and more complex logic, with solution guidelines, from programming basics exams Chapter 7.1. More Complex Loops - using for-loops with a step, loops with decreasing loop variable, using while loops, and do-while loops, solving non-trivial problems like calculating GCD (greatest common divisor) and finding the prime numbers in certain range, infinite loops with break inside, using simple try-catch statements to handle errors, building a simple Web based game using Visual Studio and ASP.NET MVC, practical exercises with more complex loops with solution guidelines Chapter 7.2. More Complex Loops – Exam Problems - practical problems with nested and more complex loops with non-trivial logic, with solution guidelines, from programming basics exams Chapter 8.1. Practical Exam Preparations – Part I - sample practical exam from the entrance exams at the Software University, with solution guidelines, covering 6 problems with simple calculations, with simple conditions, with more complex conditions, with a simple loop, with nested loops, with nested loops and more complex logic Chapter 8.2. Practical Exam Preparations – Part II - another sample practical exam from the entrance exams at the Software University, with solution guidelines, covering 6 problems with simple calculations, with simple conditions, with more complex conditions, with a simple loop, with nested loops, with nested loops and more complex logic Chapter 9.1. Problems for Champions – Part I - a sample set of more complex problems, requiring stronger algorithmic thinking and programming techniques, with solution guidelines Chapter 9.2. Problems for Champions – Part II - another set of more complex problems, requiring stronger algorithmic thinking and programming techniques, with solution guidelines Chapter 10. Methods - what is method, when to use methods, defining and calling methods (functions), passing parameters and returning values, returning multiple values, overloading methods, using nested methods (local functions), naming methods correctly, good practices for using methods Chapter 11. Tricks and Hacks - some special techniques, tricks and hacks for improving our performance with C# and Visual Studio: hints how to format the code, conventions an guidelines about naming the code elements, using keyboard shortcuts in VS, defining and using code snippets in VS, debugging code, using breakpoints and watches Conclusion - the skills of the software engineers, how to continue learning software development after this book (study software engineering in SoftUni, study in your own way), how to get learning resources and how many time it takes to become a skillful software engineer and start a job

**python is a compiled language: A Gentle Introduction to Effective Computing in Quantitative Research** Harry J. Paarsch, Konstantin Golyaev, 2016-05-06 A practical guide to using

modern software effectively in quantitative research in the social and natural sciences. This book offers a practical guide to the computational methods at the heart of most modern quantitative research. It will be essential reading for research assistants needing hands-on experience; students entering PhD programs in business, economics, and other social or natural sciences; and those seeking quantitative jobs in industry. No background in computer science is assumed; a learner need only have a computer with access to the Internet. Using the example as its principal pedagogical device, the book offers tried-and-true prototypes that illustrate many important computational tasks required in quantitative research. The best way to use the book is to read it at the computer keyboard and learn by doing. The book begins by introducing basic skills: how to use the operating system, how to organize data, and how to complete simple programming tasks. For its demonstrations, the book uses a UNIX-based operating system and a set of free software tools: the scripting language Python for programming tasks; the database management system SQLite; and the freely available R for statistical computing and graphics. The book goes on to describe particular tasks: analyzing data, implementing commonly used numerical and simulation methods, and creating extensions to Python to reduce cycle time. Finally, the book describes the use of LaTeX, a document markup language and preparation system.

**python is a compiled language: Computational Science — ICCS 2004** Marian Bubak, Geert D. van Albada, Peter M.A. Sloot, Jack Dongarra, 2004-05-25 The International Conference on Computational Science (ICCS 2004) held in Krak´ ow, Poland, June 6–9, 2004, was a follow-up to the highly successful ICCS 2003 held at two locations, in Melbourne, Australia and St. Petersburg, Russia; ICCS 2002 in Amsterdam, The Netherlands; and ICCS 2001 in San Francisco, USA. As computational science is still evolving in its quest for subjects of inves- gation and e?cient methods, ICCS 2004 was devised as a forum for scientists from mathematics and computer science, as the basic computing disciplines and application areas, interested in advanced computational methods for physics, chemistry, life sciences, engineering, arts and humanities, as well as computer system vendors and software developers. The main objective of this conference was to discuss problems and solutions in all areas, to identify new issues, to shape future directions of research, and to help users apply various advanced computational techniques. The event harvested recent developments in com- tationalgridsandnextgenerationcomputingsystems,tools,advancednumerical methods, data-driven systems, and novel application ?elds, such as complex - stems, ?nance, econo-physics and population evolution.

**python is a compiled language: Beginning Python** Magnus Lie Hetland, 2008-10-21 Gain a fundamental understanding of Python's syntax and features with the second edition of Beginning Python, an up–to–date introduction and practical reference. Covering a wide array of Python–related programming topics, including addressing language internals, database integration, network programming, and web services, you'll be guided by sound development principles. Ten accompanying projects will ensure you can get your hands dirty in no time. Updated to reflect the latest in Python programming paradigms and several of the most crucial features found in Python 3.0 (otherwise known as Python 3000), advanced topics, such as extending Python and packaging/distributing Python applications, are also covered.

**python is a compiled language: LPI Web Development Essentials Study Guide** Audrey O'Shea, 2023-10-17 Pass the LPI Web Development Essentials exam and set yourself up for success at a new web development job In LPI Linux Professional Institute Web Development Essentials Study Guide: Exam 030-100, accomplished IT educator and systems engineer, Audrey O’Shea delivers an easy-to-follow and hands-on roadmap to passing the LPI Web Development Essentials exam and hitting the ground running at a new job as a web developer. In the book, you’ll explore the software development skills, web technologies, HTML, CSS, Node.js, and JavaScript info you need to implement modern applications and solutions in a web environment. You will find: Introductory coverage of SQL, HTML, JavaScript, CSS, and MongoDB A heavy emphasis on real-world job skills, as well as the technologies used every day by web developers in the field Complimentary access to the Sybex interactive online learning environment and test bank, complete with hundreds of practice

questions, electronic flashcards, and a searchable glossary of important terms An essential and practical resource for anyone preparing for the Web Development Essentials certification exam, LPI Linux Professional Institute Web Development Essentials Study Guide: Exam 030-100 is also the ideal book for entry-level software developers seeking knowledge of web development tools and principles.

**python is a compiled language:** Essentials of Computer Organization and Architecture with Navigate Advantage Access Linda Null, 2023-04-13 Essentials of Computer Organization and Architecture focuses on the function and design of the various components necessary to process information digitally. This title presents computing systems as a series of layers, taking a bottom–up approach by starting with low-level hardware and progressing to higher-level software. Its focus on real-world examples and practical applications encourages students to develop a “big-picture” understanding of how essential organization and architecture concepts are applied in the computing world. In addition to direct correlation with the ACM/IEEE guidelines for computer organization and architecture, the text exposes readers to the inner workings of a modern digital computer through an integrated presentation of fundamental concepts and principles.

**python is a compiled language: MicroPython for the Internet of Things** Charles Bell, 2017-11-24 Quickly learn to program for microcontrollers and IoT devices without a lot of study and expense. MicroPython and controllers that support it eliminate the need for programming in a C-like language, making the creation of IoT applications and devices easier and more accessible than ever. MicroPython for the Internet of Things is ideal for readers new to electronics and the world of IoT. Specific examples are provided covering a range of supported devices, sensors, and MicroPython boards such as Pycom’s WiPy modules and MicroPython’s pyboard. Never has programming for microcontrollers been easier. The book takes a practical and hands-on approach without a lot of detours into the depths of theory. The book: Shows a faster and easier way to program microcontrollers and IoT devices Teaches MicroPython, a variant of one of the most widely used scripting languages Is friendly and accessible to those new to electronics, with fun example projects What You'll Learn Program in MicroPython Understand sensors and basic electronics Develop your own IoT projects Build applications for popular boards such as WiPy and pyboard Load MicroPython on the ESP8266 and similar boards Interface with hardware breakout boards Connect hardware to software through MicroPython Explore the easy-to-use Adafruit IO connecting your microcontroller to the cloud Who This Book Is For Anyone interested in building IoT solutions without the heavy burden of programming in C++ or C. The book also appeals to those wanting an easier way to work with hardware than is provided by the Arduino and the Raspberry Pi platforms.

**python is a compiled language: Learning Penetration Testing with Python** Christopher Duffy, 2015-09-30 Utilize Python scripting to execute effective and efficient penetration tests About This Book Understand how and where Python scripts meet the need for penetration testing Familiarise yourself with the process of highlighting a specific methodology to exploit an environment to fetch critical data Develop your Python and penetration testing skills with real-world examples Who This Book Is For If you are a security professional or researcher, with knowledge of different operating systems and a conceptual idea of penetration testing, and you would like to grow your knowledge in Python, then this book is ideal for you. What You Will Learn Familiarise yourself with the generation of Metasploit resource files Use the Metasploit Remote Procedure Call (MSFRPC) to automate exploit generation and execution Use Python's Scapy, network, socket, office, Nmap libraries, and custom modules Parse Microsoft Office spreadsheets and eXtensible Markup Language (XML) data files Write buffer overflows and reverse Metasploit modules to expand capabilities Exploit Remote File Inclusion (RFI) to gain administrative access to systems with Python and other scripting languages Crack an organization's Internet perimeter Chain exploits to gain deeper access to an organization's resources Interact with web services with Python In Detail Python is a powerful new-age scripting platform that allows you to build exploits, evaluate services, automate, and link solutions with ease. Python is a multi-paradigm programming language well suited to both object-oriented application development as well as functional design patterns.

Because of the power and flexibility offered by it, Python has become one of the most popular languages used for penetration testing. This book highlights how you can evaluate an organization methodically and realistically. Specific tradecraft and techniques are covered that show you exactly when and where industry tools can and should be used and when Python fits a need that proprietary and open source solutions do not. Initial methodology, and Python fundamentals are established and then built on. Specific examples are created with vulnerable system images, which are available to the community to test scripts, techniques, and exploits. This book walks you through real-world penetration testing challenges and how Python can help. From start to finish, the book takes you through how to create Python scripts that meet relative needs that can be adapted to particular situations. As chapters progress, the script examples explain new concepts to enhance your foundational knowledge, culminating with you being able to build multi-threaded security tools, link security tools together, automate reports, create custom exploits, and expand Metasploit modules. Style and approach This book is a practical guide that will help you become better penetration testers and/or Python security tool developers. Each chapter builds on concepts and tradecraft using detailed examples in test environments that you can simulate.

**python is a compiled language:** Working with Map Projections Fritz Kessler, Sarah Battersby, 2019-05-03 A map projection fundamentally impacts the mapmaking process. Working with Map Projections: A Guide to Their Selection explains why, for any given map, there isn't a single best map projection. Selecting a projection is a matter of understanding the compromises and consequences of showing a 3-D space in two dimensions. The book presents a clear understanding of the processes necessary to make logical decisions on selecting an appropriate map projection for a given data set. The authors discuss the logic needed in the selection process, describe why certain decisions should be made, and explain the consequences of any inappropriate decision made during the selection process. This book also explains how the map projection will impact the map's ability to fulfill its purpose, uses real-world data sets as the basis for the selection of an appropriate map projection, and provides illustrations of an appropriately and inappropriately selected map projection for a given data set. The authors take a novel approach to discussing map projections by avoiding an extensive inventory of mathematical formulae and using only the mathematics of map projections that matter for many mapping tasks. They also present information that is directly applicable to the process of selecting map projections and not tied to a specific software package. Written by two leading experts, this book is an invaluable resource for anyone studying or working with geospatial data, from students to experienced professionals, and will help readers successfully weigh the pros and cons of choosing one projection over another to suit a map's intended purpose.

**python is a compiled language:** *The Essentials of Computer Organization and Architecture* Linda Null, Julia Lobur, 2006 Computer Architecture/Software Engineering

## Related to python is a compiled language

**What does colon equal (:=) in Python mean? - Stack Overflow** In Python this is simply =. To translate this pseudocode into Python you would need to know the data structures being referenced, and a bit more of the algorithm

**python - What does the caret (^) operator do? - Stack Overflow** Side note, seeing as Python defines this as an xor operation and the method name has "xor" in it, I would consider it a poor design choice to make that method do something not related to xor

**syntax - Python integer incrementing with ++ - Stack Overflow** In Python, you deal with data in an abstract way and seldom increment through indices and such. The closest-in-spirit thing to ++ is the next method of iterators

**syntax - What do >> and << mean in Python? - Stack Overflow** The other case involving print >>obj, "Hello World" is the "print chevron" syntax for the print statement in Python 2 (removed in Python 3, replaced by the file argument of the

**operators - Python != operation vs "is not" - Stack Overflow** In a comment on this question, I saw a statement that recommended using result is not None vs result != None What is the

difference? And why might one be recommended over the other?

**Does Python have a ternary conditional operator?** Python is a syntax-rich language with lots of idiomatic tricks that aren't immediately apparent to the dabbler. But the more you learn and understand the mechanics of

**python - `from import` vs `import .` - Stack Overflow** I'm wondering if there's any difference between the code fragment from urllib import request and the fragment import urllib.request or if they are interchangeable. If they are

**Exponentials in python: x**y vs (x, y) - Stack Overflow** The dis module can be useful for checking what's happening in Python. E.g. try entering dis.dis(lambda x: -x**2) and seeing how the output changes as you parenthesise the

**python - Iterating over dictionaries using 'for' loops - Stack Overflow** Why is it 'better' to use my_dict.keys() over iterating directly over the dictionary? Iteration over a dictionary is clearly documented as yielding keys. It appears you had Python 2

**python - Errno 13 Permission denied - Stack Overflow** For future searchers, if none of the above worked, for me, python was trying to open a folder as a file. Check at the location where you try to open the file, if you have a folder with

**What does colon equal (:=) in Python mean? - Stack Overflow** In Python this is simply =. To translate this pseudocode into Python you would need to know the data structures being referenced, and a bit more of the algorithm

**python - What does the caret (^) operator do? - Stack Overflow** Side note, seeing as Python defines this as an xor operation and the method name has "xor" in it, I would consider it a poor design choice to make that method do something not related to xor

**syntax - Python integer incrementing with ++ - Stack Overflow** In Python, you deal with data in an abstract way and seldom increment through indices and such. The closest-in-spirit thing to ++ is the next method of iterators

**syntax - What do >> and << mean in Python? - Stack Overflow** The other case involving print >>obj, "Hello World" is the "print chevron" syntax for the print statement in Python 2 (removed in Python 3, replaced by the file argument of the

**operators - Python != operation vs "is not" - Stack Overflow** In a comment on this question, I saw a statement that recommended using result is not None vs result != None What is the difference? And why might one be recommended over the other?

**Does Python have a ternary conditional operator?** Python is a syntax-rich language with lots of idiomatic tricks that aren't immediately apparent to the dabbler. But the more you learn and understand the mechanics of

**python - `from import` vs `import .` - Stack Overflow** I'm wondering if there's any difference between the code fragment from urllib import request and the fragment import urllib.request or if they are interchangeable. If they are

**Exponentials in python: x**y vs (x, y) - Stack Overflow** The dis module can be useful for checking what's happening in Python. E.g. try entering dis.dis(lambda x: -x**2) and seeing how the output changes as you parenthesise the

**python - Iterating over dictionaries using 'for' loops - Stack Overflow** Why is it 'better' to use my_dict.keys() over iterating directly over the dictionary? Iteration over a dictionary is clearly documented as yielding keys. It appears you had Python 2

**python - Errno 13 Permission denied - Stack Overflow** For future searchers, if none of the above worked, for me, python was trying to open a folder as a file. Check at the location where you try to open the file, if you have a folder with

**What does colon equal (:=) in Python mean? - Stack Overflow** In Python this is simply =. To translate this pseudocode into Python you would need to know the data structures being referenced, and a bit more of the algorithm

**python - What does the caret (^) operator do? - Stack Overflow** Side note, seeing as Python defines this as an xor operation and the method name has "xor" in it, I would consider it a poor

design choice to make that method do something not related to xor

**syntax - Python integer incrementing with ++ - Stack Overflow** In Python, you deal with data in an abstract way and seldom increment through indices and such. The closest-in-spirit thing to ++ is the next method of iterators

**syntax - What do >> and << mean in Python? - Stack Overflow** The other case involving print >>obj, "Hello World" is the "print chevron" syntax for the print statement in Python 2 (removed in Python 3, replaced by the file argument of the

**operators - Python != operation vs "is not" - Stack Overflow** In a comment on this question, I saw a statement that recommended using result is not None vs result != None What is the difference? And why might one be recommended over the other?

**Does Python have a ternary conditional operator?** Python is a syntax-rich language with lots of idiomatic tricks that aren't immediately apparent to the dabbler. But the more you learn and understand the mechanics of

**python - `from import` vs `import .` - Stack Overflow** I'm wondering if there's any difference between the code fragment from urllib import request and the fragment import urllib.request or if they are interchangeable. If they are

**Exponentials in python: x**y vs (x, y) - Stack Overflow** The dis module can be useful for checking what's happening in Python. E.g. try entering dis.dis(lambda x: -x**2) and seeing how the output changes as you parenthesise the

**python - Iterating over dictionaries using 'for' loops - Stack Overflow** Why is it 'better' to use my_dict.keys() over iterating directly over the dictionary? Iteration over a dictionary is clearly documented as yielding keys. It appears you had Python 2

**python - Errno 13 Permission denied - Stack Overflow** For future searchers, if none of the above worked, for me, python was trying to open a folder as a file. Check at the location where you try to open the file, if you have a folder with

**What does colon equal (:=) in Python mean? - Stack Overflow** In Python this is simply =. To translate this pseudocode into Python you would need to know the data structures being referenced, and a bit more of the algorithm

**python - What does the caret (^) operator do? - Stack Overflow** Side note, seeing as Python defines this as an xor operation and the method name has "xor" in it, I would consider it a poor design choice to make that method do something not related to xor

**syntax - Python integer incrementing with ++ - Stack Overflow** In Python, you deal with data in an abstract way and seldom increment through indices and such. The closest-in-spirit thing to ++ is the next method of iterators

**syntax - What do >> and << mean in Python? - Stack Overflow** The other case involving print >>obj, "Hello World" is the "print chevron" syntax for the print statement in Python 2 (removed in Python 3, replaced by the file argument of the

**operators - Python != operation vs "is not" - Stack Overflow** In a comment on this question, I saw a statement that recommended using result is not None vs result != None What is the difference? And why might one be recommended over the other?

**Does Python have a ternary conditional operator?** Python is a syntax-rich language with lots of idiomatic tricks that aren't immediately apparent to the dabbler. But the more you learn and understand the mechanics of

**python - `from import` vs `import .` - Stack Overflow** I'm wondering if there's any difference between the code fragment from urllib import request and the fragment import urllib.request or if they are interchangeable. If they are

**Exponentials in python: x**y vs (x, y) - Stack Overflow** The dis module can be useful for checking what's happening in Python. E.g. try entering dis.dis(lambda x: -x**2) and seeing how the output changes as you parenthesise the

**python - Iterating over dictionaries using 'for' loops - Stack Overflow** Why is it 'better' to use my_dict.keys() over iterating directly over the dictionary? Iteration over a dictionary is clearly

documented as yielding keys. It appears you had Python 2

**python - Errno 13 Permission denied - Stack Overflow** For future searchers, if none of the above worked, for me, python was trying to open a folder as a file. Check at the location where you try to open the file, if you have a folder with

**What does colon equal (:=) in Python mean? - Stack Overflow** In Python this is simply =. To translate this pseudocode into Python you would need to know the data structures being referenced, and a bit more of the algorithm

**python - What does the caret (^) operator do? - Stack Overflow** Side note, seeing as Python defines this as an xor operation and the method name has "xor" in it, I would consider it a poor design choice to make that method do something not related to xor

**syntax - Python integer incrementing with ++ - Stack Overflow** In Python, you deal with data in an abstract way and seldom increment through indices and such. The closest-in-spirit thing to ++ is the next method of iterators

**syntax - What do >> and << mean in Python? - Stack Overflow** The other case involving print >>obj, "Hello World" is the "print chevron" syntax for the print statement in Python 2 (removed in Python 3, replaced by the file argument of the

**operators - Python != operation vs "is not" - Stack Overflow** In a comment on this question, I saw a statement that recommended using result is not None vs result != None What is the difference? And why might one be recommended over the other?

**Does Python have a ternary conditional operator?** Python is a syntax-rich language with lots of idiomatic tricks that aren't immediately apparent to the dabbler. But the more you learn and understand the mechanics of

**python - `from import` vs `import .` - Stack Overflow** I'm wondering if there's any difference between the code fragment from urllib import request and the fragment import urllib.request or if they are interchangeable. If they are

**Exponentials in python: x**y vs (x, y) - Stack Overflow** The dis module can be useful for checking what's happening in Python. E.g. try entering dis.dis(lambda x: -x**2) and seeing how the output changes as you parenthesise the

**python - Iterating over dictionaries using 'for' loops - Stack Overflow** Why is it 'better' to use my_dict.keys() over iterating directly over the dictionary? Iteration over a dictionary is clearly documented as yielding keys. It appears you had Python 2

**python - Errno 13 Permission denied - Stack Overflow** For future searchers, if none of the above worked, for me, python was trying to open a folder as a file. Check at the location where you try to open the file, if you have a folder with

# Related Articles


- study tips for pre med students
- how much more lisa harper bible study
- teaching transparency answers chapter 18

Back to Home