Java "Virtual Machine" vs. Python "Interpreter" parlance?

Asked 15 years, 11 months ago Modified 3 years, 8 months ago

Viewed 82k times

259

It seems rare to read of a Python "virtual machine" while in Java "virtual machine" is used all the time.

Both interpret byte codes; why call one a virtual machine and the other an interpreter?

java python jvm

Share

Improve this question

Follow

edited Jan 24, 2018 at 17:33

Matthias Braun

34.1k • 27 • 152 • 176

asked Jan 14, 2009 at 3:39

twils

2,754 • 3 • 18 • 7

14 Answers

Sorted by: Highest score (default)

212

In this post, "virtual machine" refers to process virtual machines, not to system virtual machines like Qemu or Virtualbox. A process virtual machine is simply a program which provides a general programming environment -- a program which can be programmed.

Java has an interpreter as well as a virtual machine, and Python has a virtual machine as well as an interpreter. The reason "virtual machine" is a more common term in Java and "interpreter" is a more common term in Python has a lot to do with the major difference between the two languages: static typing (Java) vs dynamic typing (Python). In this context, "type" refers to primitive data types -- types which suggest the in-memory storage size of the data. The Java virtual machine has it easy. It requires the programmer to specify the primitive data type of each variable. This provides sufficient information for Java bytecode not only to be interpreted and executed by the Java virtual machine, but even to be compiled into machine instructions. The Python virtual machine is more complex in the sense that it takes on the additional task of pausing before the execution of each operation to determine the primitive data types for each variable or data structure involved in the operation. Python frees the programmer from thinking in terms of primitive data types, and allows operations to be expressed at a higher level. The price of this freedom is performance. "Interpreter" is the preferred term for Python because it has to pause to inspect data types, and also because the comparatively concise syntax of dynamically-typed languages is a good

fit for interactive interfaces. There's no technical barrier to building an interactive Java interface, but trying to write any statically-typed code interactively would be tedious, so it just isn't done that way.

In the Java world, the virtual machine steals the show because it runs programs written in a language which can actually be compiled into machine instructions, and the result is speed and resource efficiency. Java bytecode can be executed by the Java virtual machine with performance approaching that of compiled programs, relatively speaking. This is due to the presence of primitive data type information in the bytecode. The Java virtual machine puts Java in a category of its own:

portable interpreted statically-typed language

The next closest thing is LLVM, but LLVM operates at a different level:

portable interpreted assembly language

The term "bytecode" is used in both Java and Python, but not all bytecode is created equal. bytecode is just the generic term for intermediate languages used by compilers/interpreters. Even C compilers like gcc use an intermediate language (or several) to get the job done. Java bytecode contains information about primitive data types, whereas Python bytecode does not. In this respect, the Python (and Bash,Perl,Ruby, etc.) virtual machine truly is fundamentally slower than the Java virtual machine, or rather, it simply has more work to do.

It is useful to consider what information is contained in different bytecode formats:

- **llvm:** cpu registers
- **Java:** primitive data types
- **Python:** user-defined types

To draw a real-world analogy: LLVM works with atoms, the Java virtual machine works with molecules, and The Python virtual machine works with materials. Since everything must eventually decompose into subatomic particles (real machine operations), the Python virtual machine has the most complex task.

Intepreters/compilers of statically-typed languages just don't have the same baggage that interpreters/compilers of dynamically-typed languages have. Programmers of statically-typed languages have to take up the slack, for which the payoff is performance. However, just as all nondeterministic functions are secretly deterministic, so are all dynamically-typed languages secretly statically-typed. Performance differences between the two language families should therefore level out around the time Python changes its name to HAL 9000.

The virtual machines of dynamic languages like Python implement some idealized logical machine, and don't necessarily correspond very closely to any real physical hardware. The Java virtual machine, in contrast, is more similar in functionality to a classical C compiler, except that instead of emitting machine instructions, it executes

built-in routines. In Python, an integer is a Python object with a bunch of attributes and methods attached to it. In Java, an int is a designated number of bits, usually 32. It's not really a fair comparison. Python integers should really be compared to the Java Integer class. Java's "int" primitive data type can't be compared to anything in the Python language, because the Python language simply lacks this layer of primitives, and so does Python bytecode.

Because Java variables are explicitly typed, one can reasonably expect something like Jython performance to be in the same ballpark as cPython. On the other hand, a Java virtual machine implemented in Python is almost guaranteed to be slower than mud. And don't expect Ruby, Perl, etc., to fare any better. They weren't designed to do that. They were designed for "scripting", which is what programming in a dynamic language is called.

Every operation that takes place in a virtual machine eventually has to hit real hardware. Virtual machines contain pre-compiled routines which are general enough to to execute any combination of logical operations. A virtual machine may not be emitting new machine instructions, but it certainly is executing its own routines over and over in arbirtrarily complex sequences. The Java virtual machine, the Python virtual machine, and all the other general-purpose virtual machines out there are equal in the sense that they can be coaxed into performing any logic you can dream up, but they are

different in terms of what tasks they take on, and what tasks they leave to the programmer.

Psyco for Python is not a full Python virtual machine, but a just-in-time compiler that hijacks the regular Python virtual machine at points it thinks it can compile a few lines of code -- mainly loops where it thinks the primitive type of some variable will remain constant even if the value is changing with each iteration. In that case, it can forego some of the incessent type determination of the regular virtual machine. You have to be a little careful, though, lest you pull the type out from under Psyco's feet. Pysco, however, usually knows to just fall back to the regular virtual machine if it isn't completely confident the type won't change.

The moral of the story is that primitive data type information is really helpful to a compiler/virtual machine.

Finally, to put it all in perspective consider this: a Python program executed by a Python interpreter/virtual machine implemented in Java running on a Java interpreter/virtual machine implemented in LLVM running in a qemu virtual machine running on an iPhone.

permalink

Share Improve this answer Follow

edited Nov 14, 2009 at 4:55

community wiki

2 revs
pooryorick

4 `trying to write any statically-typed code interactively would be tedious`. If you know OCaml and Haskell you'll see that that's not true since they are *very* concise statically-typed languages. – Matthias Braun Jan 24, 2018 at 17:41

@MatthiasBraun I can agree that those functional languages produce concise code but that doesn't necessarily mean that they are well suited for interactive mode. If OCaml and Haskell were dynamically-typed like lisp, they would work better for interactive mode I assume. – user5538922 May 9, 2018 at 1:20

I think this is one of the BEST answers I've ever read in SO. Extremely clarifying and helpful. Well explained and with real world analogies. Thanks for your answer! – NickS1 Mar 29, 2023 at 19:36

Jython isn't slow? Python is just a language, not a runtime. There is no reason a good Jython interpreter would not have similar performance to pure Java. This is because it would eventually HAVE TO emit `.CLASS` files which would run on the JVM. This means that python code would be parsed into an AST and then the Jython interpreter would use that AST to emit JVM bytecode. Unless the interpreter was just emitting horribly inefficient bytecode, there would be very little performance overhead. – ThisGuyCantEven May 31, 2023 at 18:07

IMHO the term interpreter is not very useful in this context, as technically speaking all languages, even assembly and binary are interpreted by someone. In this case, Java or Python gets interpreted by a VM, and C/assembly/binary gets interpreted by the CPU (At the end of the cascade, everything gets interpreted by the CPU). – redigaffi Oct 16, 2023 at 9:14

149

A virtual machine is a virtual computing environment with a specific set of atomic well defined instructions that are supported independent of any specific language and it is generally thought of as a sandbox unto itself. The VM is analogous to an instruction set of a specific CPU and tends to work at a more fundamental level with very basic building blocks of such instructions (or byte codes) that are independent of the next. An instruction executes deterministically based only on the current state of the virtual machine and does not depend on information elsewhere in the instruction stream at that point in time.

An interpreter on the other hand is more sophisticated in that it is tailored to parse a stream of some syntax that is of a specific language and of a specific grammer that must be decoded in the context of the surrounding tokens. You can't look at each byte or even each line in isolation and know exactly what to do next. The tokens in the language can't be taken in isolation like they can relative to the instructions (byte codes) of a VM.

A Java compiler converts Java language into a byte-code stream no different than a C compiler converts C

Language programs into assembly code. An interpreter on the other hand doesn't really convert the program into any well defined intermediate form, it just takes the program actions as a matter of the process of interpreting the source.

Another test of the difference between a VM and an interpreter is whether you think of it as being language independent. What we know as the Java VM is not really Java specific. You could make a compiler from other languages that result in byte codes that can be run on the JVM. On the other hand, I don't think we would really think of "compiling" some other language other than Python into Python for interpretation by the Python interpreter.

Because of the sophistication of the interpretation process, this can be a relatively slow process....specifically parsing and identifying the language tokens, etc. and understanding the context of the source to be able to undertake the execution process within the interpreter. To help accelerate such interpreted languages, this is where we can define intermediate forms of pre-parsed, pre-tokenized source code that is more readily directly interpreted. This sort of binary form is still interpreted at execution time, it is just starting from a much less human readable form to improve performance. However, the logic executing that form is not a virtual machine, because those codes still can't be taken in isolation - the context of the surrounding tokens

still matter, they are just now in a different more computer efficient form.

Share Improve this answer Follow

edited Jan 14, 2009 at 17:16

answered Jan 14, 2009 at 5:06

Tall Jeff

9,984 ● 7 ● 46 ● 61

9 I was under the impression that python did generate byte code, pyc, or is that what you are referring by "help accelerate such interpreted languages, this is where we can define intermediate forms of pre-parsed, pre-tokenized source code that is more readily directly interpreted." – James McMahon Jan 14, 2009 at 20:08

42 @InSciTek Jeff: From your answer it's not clear whether you do know that Python uses a virtual machine too. – tzot Jan 15, 2009 at 13:22

4 @TZ - The popular Python implementation is a Python compiler with a back side VM. In interactive mode, it is a bit of hybrid with both an interpreter front end, and a compiler back end. However those are implementation choices. I tried to describe difference between concept of VM and Interpreter – Tall Jeff Jan 15, 2009 at 14:55

10 `On the other hand, I don't think we would really think of "compiling" some other language other than Python into Python for interpretation by the Python interpreter.` It is possible to write a language that can be compiled into Python bytecode, just like Scala is compiled into Java bytecode. In interactive mode, Python's interactive shell compiles your typed command into bytecode and executes that bytecode. You

can write your own shell using eval and exec, and you can use compile() built-in function to turn a string into bytecode. – Lie Ryan Sep 27, 2010 at 13:53

4 @Lie Ryan yes but it's not officially supported like it is with the JVM. In Python, bytecode is an undocumented implementation detail. – Antimony Apr 11, 2014 at 14:05

77

Probably one reason for the different terminology is that one normally thinks of feeding the python interpreter raw human-readable source code and not worrying about bytecode and all that.

In Java, you have to explicitly compile to bytecode and then run just the bytecode, not source code on the VM.

Even though Python uses a virtual machine under the covers, from a user's perspective, one can ignore this detail most of the time.

Share Improve this answer

Follow

answered Jan 14, 2009 at 3:56

Mr Fooz

112k ● 7 ● 76 ● 103

4 I agree. This difference in terminology really comes down to the end-user (developer, that is) experience. It has nothing to do with real technical differences, as the technical line is so incredibly blurred as to be damn-near nonexistent. – Serafina Brocious Jan 14, 2009 at 4:46

2 +1: And -- more importantly -- what's the point? What program can't you write because of this distinction? What stack traceback is confusing you? What library doesn't appear to work correctly? – S.Lott Jan 14, 2009 at 11:43

@S.Lott Because it's always good to win arguments with colleagues. ;) – Qix - MONICA WAS MISTREATED Oct 6, 2014 at 3:50

21

To provide a deep answer to the question "*Why Java Virtual Machine, but Python interpreter?*" let's try to go back to the field of compilation theory as to the starting point of the discussion.

The typical process of program compilation includes next steps:

1. *Lexical analysis*. Splits program text into meaningful "words" called **tokens** (as part of the process all comments, spaces, new-lines etc. are removed, because they do not affect program behavior). The result is an ordered stream of tokens.
2. *Syntax analysis*. Builds the so-called **Abstract Syntax Tree (AST)** from the stream of tokens. AST establish relations between tokens and, as a consequence, defines an order of evaluation of the program.
3. *Semantic analysis*. Verifies semantical correctness of the AST using information about types and a set of semantical rules of the programming language. (For example, `a = b + c` is a correct statement from the

syntaxis point of view, but completely incorrect from the semantic point of view if `a` was declared as a constant object)

4. *Intermediate code generation*. Serializes AST into the linearly ordered stream of machine independent "primitive" operations. In fact, code generator traverses AST and logs the order of evaluation steps. As a result, from the tree-like representation of the program, we achieve much more simple list-like representation in which order of program evaluation is preserved.
5. *Machine code generation*. The program in the form of machine independent "primitive" bytecode is translated into machine code of particular processor architecture.

Ok. Lets now define the terms.

Interpreter, in the classical meaning of that word, assumes execution based on the program evaluation based on AST produced directly from the program **text**. In that case, a program is distributed in the form of **source code** and the interpreter is fed by program text, frequently in a dynamic way (statement-by-statement or line-by-line). For each input statement, interpreter builds its AST and immediately evaluates it changing the "state" of the program. This is a typical behavior demonstrated by scripting languages. Consider for example Bash, Windows CMD etc. Conceptually, Python takes this way too.

If we replace the AST-based execution step on the generation of intermediate machine-independent binary bytecode step in the interpreter we will split the entire process of program execution into two separate phases: compilation and execution. In that case what previously was an interpreter will become a bytecode compiler, which will transform the program from the form of the **text** into some **binary** form. Then the program is distributed in that binary form, but not in the form of source code. On the user machine, that bytecode is fed into a new entity -- **virtual machine**, which in fact interpret that bytecode. Due to this, virtual machines are also called **bytecode interpreter**. But put your attention here! A classical interpreter is a **text interpreter**, but a virtual machine is a **binary interpreter**! This is an approach taken by Java and C#.

Finally, if we add the machine code generation to the bytecode compiler we achieve in result what we call a classical **compiler**. A classical compiler converts the program source code into the **machine code** of a particular processor. That machine code then can be **directly executed** on the target processor without any additional mediation (without any kind of interpreter neither text interpreter nor binary interpreter).

Lets now go back to the original question and consider Java vs Python.

Java was initially designed to have as few implementation dependencies as possible. Its design is based on the

principle "write once, run anywhere" (WORA). To implement it, *Java* was initially designed as a programming language that compiles into machine-independent **binary bytecode**, which then can be executed on all platforms that support *Java* without the need for its recompilation. You can think about *Java* like about WORA-based *C++*. Actually, *Java* is closer to *C++* than to the scripting languages like *Python*. But in contrast to *C++*, *Java* was designed to be compiled into **binary bytecode** which then is executed in the environment of the **virtual machine**, while *C++* was designed to be compiled in machine code and then directly executed by the target processor.

Python was initially designed as a kind of scripting programing language which interprets scripts (programs in the form of the **text** written in accordance with the programming language rules). Due to this, Python has initially supported a dynamic interpretation of one-line commands or statements, as the Bash or Windows CMD do. For the same reason, initial implementations of Python had not any kind of bytecode compilers and virtual machines for execution of such bytecode inside, but from the start *Python* had required **interpreter** which is capable to understand and evaluate Python program **text**.

Due to this, historically, *Java* developers tended to talk about **Java Virtual Machine** (because initially, *Java* has come as package of *Java* bytecode compiler and **bytecode interpreter** -- **JVM**), and *Python* developers

tended to talk about *Python* interpreter (because initially *Python* has not any virtual machine and was a kind of classical **text interpreter** that executes program **text** directly without any sort of compilation or transformation into any form of binary code).

Currently, Python also has the virtual machine under the hood and can compile and interpret Python bytecode. And that fact makes an additional investment into the confusion "*Why Java Virtual Machine, but Python interpreter?*", because it seems that implementations of both languages contain virtual machines. But! Even in the current moment interpretation of program text is a primary way of Python programs execution. Python implementations exploit virtual machines under the hood exclusively as an optimization technique. Interpretation of binary bytecode in the virtual machine is much more efficient than a direct interpretation of the original program text. At the same time, the presence of the virtual machine in the Python is absolutely transparent for both Python language designers and Python programs developers. The same language can be implemented in interpreters with and without the virtual machine. In the same way, the same programs can be executed in interpreters with and without the virtual machine, and that programs will demonstrate exactly the same behavior and produce equally the same output from the equal input. The only observable difference will be the speed of program execution and the amount of memory consumed by the interpreter. Thus, the virtual machine in Python is

not an unavoidable part of the language design, but just an optional extension of the major Python interpreter.

Java can be considered in a similar way. Java under the hood has a JIT compiler and can selectively compile methods of Java class into machine code of the target platform and then directly execute it. But! Java still uses bytecode interpretation as a primary way of Java program execution. Like Python implementations which exploit virtual machines under the hood exclusively as an optimization technique, the Java virtual machines use Just-In-Time compilers exclusively for optimization purposes. Similarly, just because of the fact that direct execution of the machine code at least ten times faster than the interpretation of Java bytecode. And like in the case of Python, the presence of JIT compiler under the hood of JVM is absolutely transparent for both Java language designers and Java program developers. The same Java programming language can be implemented by JVM with and without JIT compiler. And in the same way, the same programs can be executed in JVMs with and without JIT inside, and the same programs will demonstrate exactly the same behavior and produce equally the same output from the equal input on both JVMs (with and without JIT). And like in the case of Python, the only observable difference between them, will be in the speed of execution and in the amount of memory consumed by JVM. And finally, like in the case of Python, JIT in Java also is not an unavoidable part of the language design, but just an optional extension of the major JVM implementations.

From the point of view of design and implementation of virtual machines of Java and Python, they differ significantly, while (attention!) both still stay virtual machines. JVM is an example of a low-level virtual machine with simple basic operations and high instruction dispatch cost. Python in its turn is a high-level virtual machine, for which instructions demonstrate complex behavior, and instruction dispatch cost is not so significant. Java operates with very low abstraction level. JVM operates on the small well-defined set of primitive types and has very tight correspondence (typically one to one) between bytecode instructions and native machine code instructions. In contrary, Python virtual machine operates at high abstraction level, it operates with complex data types (objects) and supports ad-hoc polymorphism, while bytecode instructions expose complex behavior, which can be represented by a series of multiple native machine code instructions. For example, Python supports unbounded range mathematics. Thus Python VM is forced to exploit long arithmetics for potentially big integers for which result of the operation can overflow the machine word. Hence, one bytecode instruction for arithmetics in Python can expose into the function call inside Python VM, while in JVM arithmetic operation will expose into simple operation expressed by one or few native machine instructions.

As a result, we can draw the next conclusions. Java Virtual Machine but Python interpreter is because:

1. The term of virtual machine assumes binary bytecode interpretation, while the term interpreter assumes program text interpretation.
2. Historically, Java was designed and implemented for binary bytecode interpretation and Python was initially designed and implemented for program text interpretation. Thus, the term "Java Virtual Machine" is historical and well established in the Java community. And similarly, the term "Python Interpreter" is historical and well established in the Python community. Peoples tend to prolong the tradition and use the same terms that were used long before.
3. Finally, currently, for Java, binary bytecode interpretation is a primary way of programs execution, while JIT-compilation is just an optional and transparent optimization. And for Python, currently, program text interpretation is a primary way of Python programs execution, while compilation into Python VM bytecode is just an optional and transparent optimization.

Therefore, both Java and Python have virtual machines are binary bytecode interpreters, which can lead to confusion such as "*Why Java Virtual Machine, but Python interpreter?*". The key point here is that for Python, a virtual machine is not a primary or necessary means of program execution; it is just an optional extension of the classical text interpreter. On the other hand, a virtual

machine is a core and unavoidable part of Java program execution ecosystem. Static or dynamic typing choice for the programming language design affects mainly the virtual machine abstraction level only, but does not dictate whether or not a virtual machine is needed. Languages using both typing systems can be designed to be compiled, interpreted, or executed within the environment of virtual machine, depending on their desired execution model.

Share Improve this answer

Follow

edited Dec 18, 2018 at 23:49

Farhad Abdolhosseini

486 ● 5 ● 8

answered Nov 17, 2018 at 23:10

ZarathustrA

3,620 ● 33 ● 28

8 This should be chosen as the official answer IMHO. – Ravikanth Andhavarapu Nov 29, 2018 at 22:58

The official answer should by YES since " both Java and Python have virtual machines are binary bytecode interpreters." Period. – stuartw Jul 4, 2019 at 4:33

20

Interpreter, translates source code into some efficient intermediate representation (code) and immediately executes this.

Virtual Machine, explicitly executes stored pre-compiled code built by a compiler which is part of the interpreter system.

A very important characteristic of a virtual machine is that the software running inside, is limited to the resources provided by the virtual machine. Precisely, it cannot break out of its virtual world. Think of secure execution of remote code, Java Applets.

In case of python, if we are keeping *pyc* files, as mentioned in the comment of this post, then the mechanism would become more like a VM, and this bytecode executes faster -- it would still be interpreted but from a much computer friendlier form. If we look at this as a whole, PVM is a last step of Python Interpreter.

The bottomline is, when refer Python Interpreter, it means we are referring it as a whole, and when we say PVM, that means we are just talking about a part of Python Interpreter, a runtime-environment. Similar to that of Java, we refer different parts differentyl, JRE, JVM, JDK, etc.

For more, Wikipedia Entry: Interpreter, and Virtual Machine. Yet another one here. Here you can find the Comparison of application virtual machines. It helps in understanding the difference between, Compilers, Interpreters, and VMs.

edited May 29, 2015 at 11:15

answered Jan 14, 2009 at 7:44

Adeel Ansari
39.9k ● 12 ● 96 ● 135

There's no real difference between them, people just follow the conventions the creators have chosen.

15

Share Improve this answer

Follow

answered Jan 14, 2009 at 3:41

Serafina Brocious

30.6k ● 12 ● 91 ● 115

3 I'll throw you a bone here since I think this is probably the real answer and you got down voted for lack of bits .
– vikingben Jan 12, 2016 at 2:21

13

The term interpreter is a legacy term dating back to earlier shell scripting languages. As "scripting languages" have evolved into full featured languages and their corresponding platforms have become more sophisticated and sandboxed, the distinction between a virtual machine and an interpreter (in the Python sense), is very small or non-existent.

The Python interpreter still functions in the same way as a shell script, in the sense that it can be executed without a separate compile step. Beyond that, the differences between Python's interpreter (or Perl or Ruby's) and Java's virtual machine are mostly implementation details. (One could argue that Java is more fully sandboxed than Python, but both ultimately provide access to the underlying architecture via a native C interface.)

Share Improve this answer

Follow

answered Jan 14, 2009 at 4:27

Daniel Naab

23k ● 8 ● 56 ● 55

1 there are java shells that can run java code without separate (user visible) compile steps. – Lie Ryan Oct 27, 2013 at 16:21

1 gimme the name :D – Maciej Nowicki Oct 18, 2017 at 10:20

3

Don't forget that Python has JIT compilers available for x86, further confusing the issue. (See psyco).

A more strict interpretation of an 'interpreted language' only becomes useful when discussing performance issues of the VM, for example, compared with Python, Ruby was (is?) considered to be slower because it is an interpreted language, unlike Python - in other words, context is everything.

Share Improve this answer

Follow

answered Jan 14, 2009 at 4:36

Arafangion

11.9k ● 1 ● 43 ● 74

2 That's wrong. First, there is no such thing as an "interpreted language". Whether an implementation uses a compiler or an interpreter is not a trait of the language but of the implementation. Second, of the 13 or so Ruby implementations, exactly 1 is an interpreter, all the others have compilers. – Jörg W Mittag Jan 14, 2009 at 18:03

3 Third, Ruby is not slow. No language is slow, because speed is not a trait of the language, but the language implementation. Of the 13 or so Ruby implementations, some are slower than some of the 7 Python implementations, some are faster. – Jörg W Mittag Jan 14, 2009 at 18:05

I think he is comparing the standard implementations here Jörg. CPython and Ruby (I think the official implementation is just named Ruby). – James McMahon Jan 14, 2009 at 20:16

While Arafangion might be referring to the "standard" implementations, he should have said so. I'm a Pythonista but I hate *any* statement of the form "Language X is slow", since I agree with Jörg on the matter of implementations. – tzot Jan 15, 2009 at 13:25

1 This is exactly why I said "was (is?)", and particularly the term "slower". Nowhere did I say that Ruby is in itself slow. – Arafangion Feb 17, 2009 at 0:02

3

Python can interpret code without compiling it to bytecode. **Java can't**.

> Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run.

(from the documentation).

In java, every single file *has to* be compiled to a `.class` file, which then runs on the JVM. On the contrary, python does that are imported by your main script, to help speed up subsequent uses of those files.

However, in the typical case, most of the python (at least, CPython) code runs in an emulated stack machine, which has nearly identical instructions to those of the JVM, so there's no great difference.

The real reason for the distiction however is because, from the beginning, java branded itself as "portable, executable bytecode" and python branded itself as dynamic, interpreted language with a REPL. Names stick!

Share Improve this answer Follow

answered Sep 15, 2019 at 9:33

aris

29k ● 10 ● 82 ● 107

Just because the bytecode is not dumped to a `.pyc` file, does not mean it was not generated by Cpython. it is in the `__code__` attribute of whatever object it is tied to. CPython's VM ONLY EXECUTES BYTECODE. The reason `.pyc` files exist is to save the Cpython interpreter the time of re-emitting the same bytecode repeatedly.
– ThisGuyCantEven May 31, 2023 at 18:01

2

There actually might be a reason why the HotSpot runtime is called a Virtual Machine while CPython is merely referred to as an interpreter

Firstly, CPython is just your average run of the mill, stack based, bytecode interpreter. You input Python opcode into it, and the software stack machine inside CPython evaluates your code, just like a normal Interpreter would.

The Java HotSpot runtime is different. First and foremost, Java has 3 Just-in Time Compilers, C1, C2, and an experimental one that isn't in use yet. But that's not the main reason. The Interpreter inside the JVM is a very special kind of Interpreter called a Template Interpreter. Instead of just executing bytecode directly in a massive opcode switch case statement like CPython (And really almost every other interpreter does) does, the Template Interpreter inside the JVM contains an enormous arraylist. What does it contain? Key-value pairs of bytecodes and native CPU instructions! The arraylist is empty on startup and is filled with mappings of bytecodes pointing to native machine language to be directly run on the hardware just before your application starts up, what this means is that the "Interpreter" inside the JVM isn't actually an Interpreter at all- It's actually a discount Compiler! When Java bytecode is run, the "Interpreter" simply maps the input bytecode directly to native machine language and executes the native mapping directly, rather than implementing it in software. I'm not exactly sure why the JVM was made this way, but I suspect it was to easily execute "Interpreted" Code together with JIT Compiled Code seamlessly, and for speed/performance. If you pitted the JVM without JIT against CPython or most other interpreters it would still probably come out ahead of

them, in virtue of its ingenious design which to my knowledge no other language has used before.

Share Improve this answer Follow

answered Apr 2, 2021 at 9:45

vertig0

41 ● 1

1 Your answer concentrates more on the difference between just-in-time compilation vs. interpretation, and does not seem to answer the question asking about ‘virtual machine’ vs. ‘interpreter’. Consider modifying or expanding your answer. – Johan Bezem Apr 2, 2021 at 11:28

1 CPython is absolutely a VM, regardless of whether or not it is "reffered to" as such. If you run your python code in CPython, it is parsed into an AST, which is then used to emit "bytecode" (think `.pyc` files), which is native to the Cpython VM. The only real difference is that Java `.CLASS` files are not compiled at runtime (Unless you are JITing). The Cpython VM also is not purely stack-based, it maintains its own private heap (which is dynamically allocated), where it stores ALL objects. – ThisGuyCantEven May 31, 2023 at 17:51

Also worth mentioning that not all bytecode is emitted to files, often it is just dumped into the `__code__` attribute of the object it is tied to. The reason `.pyc` files exist is if the same large bytecode will be used multiple times, the interpreter will try to save the bytecode to a file to avoid generating it over and over. – ThisGuyCantEven May 31, 2023 at 17:55

1

I think the lines between both are blurred, people mostly argue around meaning of word "interpreter" and how close the language stands to each side of "interpreter...compiler" spectrum. None makes 100%



however. I think it is easy to write Java or Python implementation which be of any value of the spectrum.

Currently both Java and Python have virtual machines and bytecode, though one operates by concrete value sizes (like 32-bit integer) while other has to determine the size for each call, which in my opinion doesn't define the border between the terms.

The argument that Python doesn't have officially defined bytecode and it exists only in memory also doesn't convince me, just because I am planning to develop devices which will recognize only Python bytecode and the compilation part will be done in browser JS machine.

Performance is only about the concrete implementation. We don't need to know the size of the object to be able to work with it, and finally, in most cases, we work with structures, not basic types. It is possible to optimize Python VM in the way that it will eliminate the need of creating new object each time during expression calculation, by reusing existing one. Once it is done, there is no global performance difference between calculating sum of two integers, which is where Java shines.

There is no killer difference between the two, only some implementation nuances and lack of optimization which are irrelevant to the end user, maybe up at the point where she starts to notice performance lags, but again it is implementation and not architecture issue.

Follow

answered Oct 2, 2019 at 10:41

Oleg Gordeev

582 ● 4 ● 6

I think it's important to note that "Python" does not have virtual machines or bytecode. It is just a language specification. A specific python runtime(IE Cpython or Jython) might have those things, though. – ThisGuyCantEven May 31, 2023 at 17:57

0

First of all you should understand that programming or computer science in general is not mathematics and we don't have rigorous definitions for most of the terms we use often.

now to your question :

what is an Interpreter (in computer science)

It translates source code by smallest executable unit and then executes that unit.

what is a virtual machine

in case of JVM the virtual machine is a software which contains an Interpreter, class loaders, garbage collector, thread scheduler , JIT compiler and many other things.

as you can see interpreter is a part or JVM and whole JVM can not be called an interpreter because it contains many other components.

why use word "Interpreter" when talking about python

with java the compilation part is explicit. python on the other hand is not explicit as java about its compilation and interpretation process, from end user's perspective interpretation is the only mechanism used to execute python programs

Share Improve this answer

Follow

answered Oct 1, 2016 at 9:06

mightyWOZ

8,275 ● 3 ● 33 ● 50

0

No, they don't both interpret byte code.

Python only interprets bytecode if you are running with pypy. Otherwise it is compiled into C and interpreted at that level.

Java compiles to bytecode.

Share Improve this answer

Follow

answered Mar 12, 2018 at 17:46

Michael Tamillow

425 ● 4 ● 11

Can you give any resources to your answer? – Isuru Dilshan Apr 8, 2018 at 12:46

and this is on Java: en.wikipedia.org/wiki/Java_virtual_machine – Michael Tamillow Apr 12, 2018 at 20:51

This is what's wrong with Stack Overflow. Someone has a pissy fit because they are called out and expresses it with downvotes. – Michael Tamillow Jan 30, 2020 at 21:04

This is utterly false. CPython has ALWAYS ONLY EXECUTED BYTECODE. All bytecode is either dumped into a `.pyc` file by the interpreter, or stored in the `__code__` attribute of the object it is tied to. Pypy simply uses its own different intermediate representation (IR), which is actually then used to emit LLVM bytecode on the fly. I must repeat... CPYTHON ONLY EXECUTES BYTECODE AND IS A VM. – ThisGuyCantEven May 31, 2023 at 18:03

0

for posts that mention that python does not need to generate byte code, I'm not sure that's true. it seems that all callables in Python must have a `.__code__.co_code` attribute which contains the byte code. I don't see a meaningful reason to call python "not compiled" just because the compiled artifacts may not be saved; and often aren't saved by design in Python, for example all comprehension compile new bytecode for it's input, this is the reason comprehension variable scope is not consistent between `compile(mode='exec, ...)` and compile `compile(mode='single', ...)` such as between running a python script and using pdb

Share Improve this answer Follow

answered Jan 13, 2020 at 23:52

community wiki
ThorSummoner