

# How Does Python Interpreter Actually Work?

February 26, 2023

## Contents

<b>1 Overview</b>	<b>1</b>
<b>2 But, What is a Virtual Machine?</b>	<b>2</b>
2.1 General Virtual Machine . . . . .	2
2.2 Virtual Machine Related to Programming Languages . . . . .	3
2.3 Python Virtual Machine . . . . .	3
2.4 Python as A Stack Based VM . . . . .	4
<b>3 Example of A Disassembly process</b>	<b>5</b>
<b>4 An Optimization</b>	<b>8</b>
<b>5 Some Terminology</b>	<b>8</b>
5.1 Run time & Runtime Environment . . . . .	8
5.2 Constant pool . . . . .	9
<b>6 Resources and further reading</b>	<b>9</b>

## 1 Overview

When you run a Python program, the interpreter goes through several stages to execute the code. These stages are:

1. **Tokenization/Lexing:** The interpreter reads the source code and breaks it up into individual *tokens* or *lexemes*, which are the smallest units of meaning in the language. For example, an identifier like `my_variable` or an operator like `+` is a token.
2. **Parsing:** The interpreter takes the stream of tokens and parses it according to the rules of the Python grammar. This involves constructing an *abstract syntax tree (AST)* that represents the *structure of the code*.
3. **Compilation:** The interpreter generates *bytecode* from the AST. Bytecode

is a lower-level representation of the code that can be executed by the *Python virtual machine (VM)*.

4. **Execution:** The interpreter executes the bytecode on the *Python VM*. This involves *loading* the bytecode into memory, *setting up* the execution environment, and *executing* each instruction in the bytecode *in sequence*.

During the execution stage, the interpreter may also perform additional tasks, such as *garbage collection* to reclaim memory that is no longer being used, or *importing modules* that are required by the code.

Overall, these stages allow the Python interpreter to take source code written in a high-level language and execute it efficiently on a virtual machine.

## 2 But, What is a Virtual Machine?

### 2.1 General Virtual Machine

A virtual machine (VM) is a software emulation of a physical machine or computer system. VMs allow multiple operating systems or environments to run on a single physical machine, without requiring separate hardware for each system. VMs achieve this by providing a layer of abstraction between the physical hardware and the operating system or software running on top of it.

VMs can be useful for a variety of reasons. For example:

They allow developers and IT professionals to test and run different operating systems and applications without needing separate hardware for each system. They can simplify software deployment by allowing applications and services to be run in a consistent, isolated environment that is independent of the underlying hardware and operating system. They can provide a way to sandbox potentially dangerous or untrusted software, by running it in an isolated environment that can be easily discarded or reset. VMs work by using a combination of software and hardware to create a virtualized environment. The software component, known as a hypervisor or virtual machine monitor (VMM), runs on the physical hardware and creates one or more virtual machines, each of which appears to the operating system or software running on it as if it were a physical machine.

The hypervisor provides virtualized versions of hardware components, such as CPUs, memory, storage devices, and network adapters, which the virtual machines can use to run their operating systems and applications. The hypervisor also manages access to the physical hardware, ensuring that each virtual machine has access to the resources it needs, while preventing conflicts or interference between the different virtual machines.

When software or an operating system runs on a virtual machine, it interacts with the virtualized hardware as if it were running on a physical machine. The virtual machine's operating system sends commands to the hypervisor, which translates them into commands that can be executed by the physical hardware.

Similarly, data from the physical hardware is intercepted by the hypervisor and redirected to the appropriate virtual machine.

Overall, virtual machines provide a powerful way to abstract and isolate different software environments, allowing them to run independently of the underlying hardware and operating system. This can be useful for development, testing, deployment, and security, among other applications.

## 2.2 Virtual Machine Related to Programming Languages

In the context of programming languages like Java and Python, a VM (Virtual Machine) is a software implementation of a computer that executes code compiled or interpreted from the respective programming languages.

For example, the Java Virtual Machine (JVM) is an abstract machine that provides a runtime environment in which Java bytecode can be executed. The JVM is responsible for translating the bytecode into machine code that can be executed by the underlying hardware.

Similarly, Python code is typically executed using the Python Virtual Machine (PVM), which is responsible for interpreting and executing Python bytecode. The PVM provides a runtime environment for Python code, which includes support for dynamic typing, garbage collection, and other features of the Python language.

In both cases, the VM provides a layer of abstraction between the code written in the respective programming language and the underlying hardware, allowing the code to be executed on a wide range of systems without requiring modifications to the code itself. The VM also provides various runtime services, such as memory management, security, and exception handling, that are essential for running complex software systems.

## 2.3 Python Virtual Machine

The **Python Virtual Machine (VM)** is a software implementation of the Python programming language. It is responsible for executing the bytecode generated by the Python compiler and translating it into machine code that can be run on the *host system*.

The Python VM is designed to be **platform-independent**, meaning that it can run on any system that has a compatible implementation of the Python runtime. This is possible because the bytecode generated by the compiler is a *standardized format* that can be interpreted by any Python VM.

The Python VM provides several key features that make it a powerful and flexible runtime environment for Python code. These include:

1. **Automatic memory management:** The VM automatically manages memory allocation and deallocation, so developers don't have to worry

about manual memory management like in languages such as C++.

2. **Dynamic typing:** Python is a dynamically typed language, which means that the type of a variable can change at runtime. The Python VM is responsible for managing this dynamic type information and ensuring that operations are performed correctly.
3. **Garbage collection:** The Python VM includes a garbage collector that automatically frees up memory that is no longer being used, so developers don't have to manually manage memory deallocation.
4. **Interpreted execution:** Python code is typically executed in an interpreted fashion, meaning that it is executed *line-by-line* by the VM. This allows for **fast feedback during development** and makes it easy to write *scripts* and quick *prototypes*<sup>1</sup>.

Overall, the Python VM is a powerful and flexible runtime environment that provides many features that make Python a popular language for a wide range of applications.

## 2.4 Python as A Stack Based VM

A stack-based virtual machine is a type of virtual machine that uses a stack to store and manipulate data. In a stack-based VM, operands are pushed onto the stack, and operations are performed by popping operands off the stack, performing the operation, and pushing the result back onto the stack.

Python's virtual machine, known as the "Python VM" or the "PVM," is a stack-based virtual machine. When Python code is executed, the interpreter uses a stack to keep track of the execution state, including the values of variables and the results of operations.

For example, when a Python function is called, the interpreter pushes the arguments onto the stack, along with a reference to the function itself. The function's code is then executed, with the interpreter popping arguments and local variables off the stack as needed, and pushing the results of operations back onto the stack. When the function returns, the result is left on top of the stack for the caller to access.

---

<sup>1</sup>the term "prototypes" refers to quick and simple implementations of a script or application that are used to test out an idea or demonstrate functionality, without necessarily implementing the full range of features or optimizations that would be required in a production-ready version. In other words, a "prototype" is a preliminary or early version of a script or application that is created for the purpose of testing and experimentation, rather than for final deployment or distribution. *Prototyping* is often used in software development as a way to *rapidly iterate on ideas and gather feedback*, before investing significant time and resources into a more complete implementation. In Python, the ease of writing and executing code makes it well-suited to prototyping, since developers can quickly write and test small snippets of code to try out different ideas and approaches. Once a working prototype is in place, it can be refined and optimized over time to create a more robust and polished final product.

The use of a stack-based VM has several advantages, including simplicity, flexibility, and efficiency. Because the VM operates on a stack of values, rather than on individual registers or memory locations, it can be more easily optimized and adapted to different architectures and platforms. Additionally, stack-based VMs are often more compact and efficient than other types of VMs, since they require fewer instructions to perform the same operations.

### 3 Example of A Disassembly process

src: the great Anthony Sotttile

```
# t.py
def f():
    thing = 'world'
    print('hello hello ' + thing)

if __name__ == "__main__":
    f()
```

If you run the script by passing it to the interpreter on the commandline, Python doesn't bother to "compile" that, because it assumes that you're gonna run that script once, or it might be an extensionless script and wouldn't really know where to put the compiled file.

```
python t.py
```

However, if you import the code,

```
python -c 'import t'
```

it will create a `__pycache__` directory (if you're in python2 you get a `.pyc` file) that contains `t.cpython-310.pyc`, a file with `.pyc` extension, which is a "compiled" python file. This file contains the bytecode that python runs.

To get the bytecode, we use the python's builtin `dis` module:

```
python -m dis t.py
```

the output is:

1	0 LOAD_CONST	0 (<code object f at 0x1048526b0, file "t.py", line 1>)
	2 LOAD_CONST	1 ('f')
	4 MAKE_FUNCTION	0
	6 STORE_NAME	0 (f)
5	8 LOAD_NAME	1 (__name__)
	10 LOAD_CONST	2 ('__main__')
	12 COMPARE_OP	2 (==)
	14 POP_JUMP_IF_FALSE	13 (to 26)

```

6          16 LOAD_NAME          0 (f)
          18 CALL_FUNCTION      0
          20 POP_TOP
          22 LOAD_CONST          3 (None)
          24 RETURN_VALUE

5      >> 26 LOAD_CONST          3 (None)
          28 RETURN_VALUE

```

Disassembly of <code object f at 0x1048526b0, file "t.py", line 1>:

```

2          0 LOAD_CONST          1 ('world')
          2 STORE_FAST           0 (thing)

3          4 LOAD_GLOBAL          0 (print)
          6 LOAD_CONST          2 ('hello hello ')
          8 LOAD_FAST            0 (thing)
         10 BINARY_ADD
         12 CALL_FUNCTION          1
         14 POP_TOP
         16 LOAD_CONST          0 (None)
         18 RETURN_VALUE

```

chatgpt: The disassembly consists of two parts: the code for the `f()` function (starting from **line 1**) and the code for the script that calls `f()` (starting from **line 5**). Here’s a line-by-line explanation of the disassembly:

**Lines 1-6:** The code for the `f()` function. **Line 1** loads a code object for `f()` and stores it in the global namespace as the name `f`. **Line 2** loads the string `'f'` and stores it in the **constant pool**. **Line 4** creates a function object from the code object and pushes it onto the stack. **Line 6** stores the function object in the global namespace as the name `f`.

In the disassembly line “6 STORE\_NAME 0 (f)”, the words and numbers have the following meanings:

- 6: the line number in the disassembly where this bytecode instruction appears.
- STORE\_NAME: the *opcode* or bytecode instruction being executed. This opcode stores the top of the stack value in a variable with the given name.
- 0: the index of the name in the constant pool that represents the variable name being stored. In this case, 0 refers to the name `f`.
- (f): the name of the variable being stored. In this case, the constant pool value at index 0 is the string `'f'`, which represents the name of the variable being stored.

So, the instruction 6 STORE\_NAME 0 (f) stores the value at the top of the stack into the variable `f`. This is how the function object created on **line 4** is given

the name `f` in the global namespace.

**Lines 8-24:** The code for the script that calls `f()`. **Line 8** loads the current module's name and stores it in the stack. **Line 10** loads the string `'__main__'` and stores it in the constant pool. **Line 12** compares the top two stack values for equality (using the `==` operator) and jumps to the target at **line 13** if they are not equal. **Line 16** loads the function object `f` from the global namespace and calls it with no arguments. **Line 20** discards the function's return value. **Line 22** loads the constant `None` and returns it.

**Lines 26-28:** The target of the conditional jump at **line 12**. **Line 26** loads the constant `None` and returns it.

**Lines 2-18:** The disassembly of the `f()` function. **Line 2** loads the constant string `'world'` and stores it in the local variable `thing`. **Line 4** loads the `print` function from the global namespace and stores it on the stack. **Line 6** loads the constant string `'hello hello '` and stores it on the stack. **Line 8** loads the value of `thing` and stores it on the stack. **Line 10** concatenates the two strings using the `BINARY_ADD` bytecode, which pops the top two stack values, concatenates them, and pushes the result back onto the stack. **Line 12** calls the `print` function with the single argument on the top of the stack. **Line 14** pops the return value of the function (which is `None`) from the stack. **Line 16** loads the constant `None` and returns it.

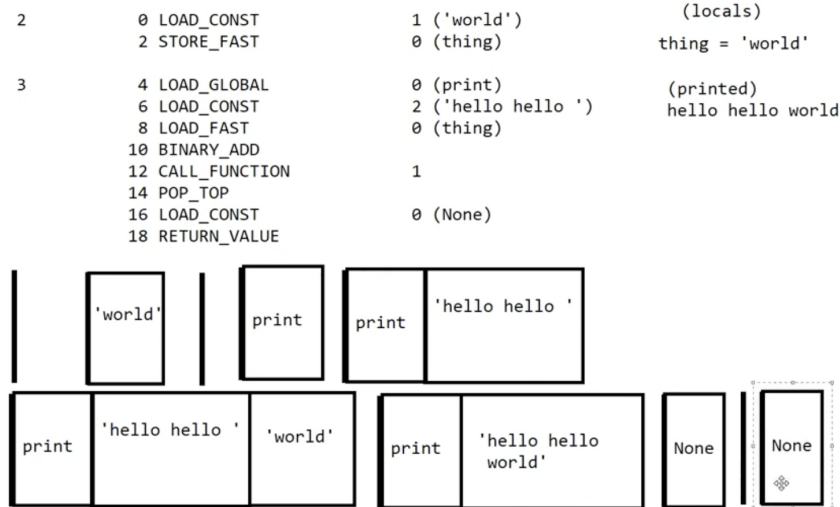


Figure 1: stack ([src](#))

## 4 An Optimization

If instead of storing `world` in a variable and passing it to `print` function, you pass the string `world` like following,

```
# t.py
def f():
    print('hello hello ' + 'world')

if __name__ == "__main__":
    f()
```

Python will optimize it per below

...

Disassembly of <code object f at 0x102f526b0, file "t.py", line 1>:

3	0 LOAD_GLOBAL	0 (print)
	2 LOAD_CONST	1 ('hello hello world')
	4 CALL_FUNCTION	1
	6 POP_TOP	
	8 LOAD_CONST	0 (None)
10	RETURN_VALUE	

essentially when python sees two string literal it will optimize out the `BINARY_ADD` operation.

## 5 Some Terminology

### 5.1 Run time & Runtime Environment

The **Python runtime**, also known as the *Python interpreter*, is the software that executes Python code. It provides an environment where Python code can be executed, and includes a number of built-in features and functions that are available to Python programs.

The Python runtime is responsible for executing Python code and managing memory allocation, data types, and other low-level details. It also provides access to the Python standard library and other external libraries, which can be imported and used in Python programs.

The **Python runtime environment** typically includes the Python interpreter, standard library modules, and any external libraries that have been installed on the system. It can also include other tools and utilities that are used in the development and deployment of Python applications, such as development environments, debugging tools, and deployment frameworks.

In general, the Python runtime environment provides everything that is needed to run Python code, from the basic language features and syntax to the rich



ecosystem of libraries and tools that make Python a popular choice for a wide range of applications.

## 5.2 Constant pool

In Python, the **constant pool** is a table of constants that are used in a *compiled code object*.

When a Python module is imported or a script is executed, the Python interpreter first compiles the source code into bytecode instructions, which are then executed by the interpreter. During the compilation process, the Python compiler builds a constant pool that contains *all of the constant values* used in the code, such as strings, integers, floating-point numbers, and other immutable objects.

The constant pool is a separate data structure that stores the constant values as bytecode instructions that can be *efficiently* loaded onto the stack during execution. The bytecode instructions that reference the constant pool use integer indexes to refer to specific constants in the pool.

For example, consider the following Python code:

```
x = 123
y = 'hello'
z = [1, 2, 3]
```

The compiled bytecode instructions for this code will include a constant pool that contains the integer 123, the string 'hello', and a reference to a new list object [1, 2, 3]. The instructions that assign these values to variables will use integer indexes to reference the corresponding constant values in the pool.

The constant pool is an implementation detail of the Python interpreter and is not directly accessible from Python code. However, it is an important part of the compiled code object and can affect the performance and memory usage of the code at runtime.

## 6 Resources and further reading

- [chatgpt](#)
- [Python behind the scenes #1: how the CPython VM works](#)
- [python is compiled? \(+ disassembler\) \(intermediate\) anthony explains #203](#)
- [Python's dis module](#)