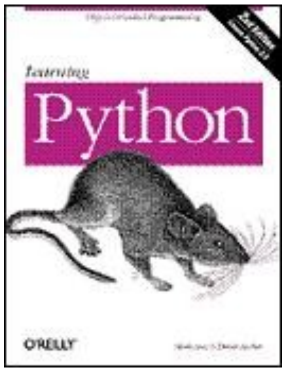
Python

How Python Runs Programs
Contributed by O'Reilly Media
2004−07−06

Python is a language and an interpreter that executes other programs. Get a quick look at program execution, how to launch code and how Python runs it. This chapter is from *Learning Python*, second edition, by Mark Lutz and David Ascher (ISBN: 0−596−00281−5, O'Reilly, 2003).



Introduction

This chapter and the next give a quick look at program execution—how you launch code, and how Python runs it. In this chapter, we explain the Python interpreter. Chapter 3 will show you how to get your own programs up and running.

Startup details are inherently platform−specific, and some of the material in this chapter may not apply to the platform you work on, so you should feel free to skip parts not relevant to your intended use. In fact, more advanced readers who have used similar tools in the past, and prefer to get to the meat of the language quickly, may want to file some of this chapter away for future reference. For the rest of you, let's learn how to run some code.

Introducing the Python Interpreter

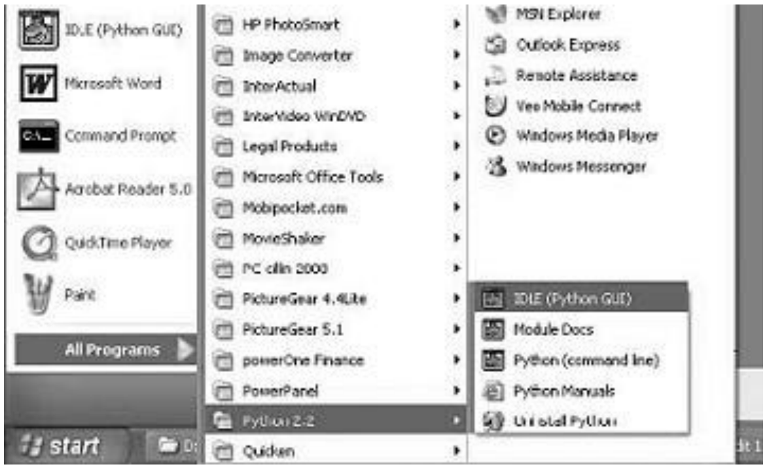
So far, we've mostly been talking about Python as a programming language. But as currently implemented, it's also a software package called an *interpreter*. An interpreter is a kind of program that executes other programs. When you write Python programs, the Python interpreter reads your program, and carries out the instructions it contains. In effect, the interpreter is a layer of software logic between your code and the computer hardware on your machine.

When the Python package is installed on your machine, it generates a number of components—minimally, an interpreter and a support library. Depending on how you use it, the Python interpreter may take the form of an executable program, or a set of libraries linked into another program. Depending on which flavor of Python you run, the interpreter itself may be implemented as a C program, a set of Java classes, or other. Whatever form it takes, the Python code you write must always be run by this interpreter. And to do that, you must first install a Python interpreter on your computer.

Python installation details vary per platform, and are covered in depth in Appendix A. In short:

- Windows users fetch and run a self−installing executable file, which puts Python on their machine. Simply double−click and say Yes or Next at all prompts.
- Linux and Unix users typically either install Python from RPM files, or compile it from its full source−code distribution package.
- Other platforms have installation techniques relevant to that platform. For instance, files are synched on Palm Pilots.

Python itself may be fetched from the downloads page at Python's web site, *www. python.org*. It may also be found through various other distribution channels. You may have Python already available on your machine, especially on Linux and Unix. If you're working on Windows, you'll usually find Python in the Start menu, as captured in Figure 2−1 (we'll learn what these menu items mean in a moment). On Unix and Linux, Python probably lives in your */usr* directory tree.



Because installation details are so platform−specific, we'll finesse the rest of this story here. (For more details on the installation process, consult Appendix A.) For the purposes of this chapter and the next, we'll assume that you've got Python ready to go.

What it means to write and run a Python script depends on whether you look at these tasks as a programmer or as a Python interpreter. Both views offer important perspective on Python programming.

The Programmer's View

In its simplest form, a Python program is just a text file containing Python statements. For example, the following file, named *script1.py*, is one of the simplest Python scripts we could dream up, but it passes for an official Python program:

```
print 'hello world'
print 2 ** 100
```

This file contains two Python print statements, which simply print a string (the text in quotes) and a numeric expression result (2 to the power 100) to the output stream. Don't worry about the syntax of this code yet—for this chapter, we're interested only in getting it to run. We'll explain the print statement, and why you can raise 2 to the power 100 in Python without overflowing, in later parts of this book.

You can create such a file of statements with any text editor you like. By convention, Python program files are given names that end in “.py”; technically, this naming scheme is required only for files that are “imported,” as shown later in this book, but most Python files have *.py* names for consistency.

After you’ve typed these statements into a text file in one way or another, you must tell Python to *execute* the file—which simply means to run all the statements from top to bottom in the file, one after another. Python program files may be launched by command lines, by clicking their icons, and with other standard techniques. We’ll demonstrate how to invoke this execution in the next chapter. If all goes well, you’ll see the results of the two print statements show up somewhere on your computer— by default, usually in the same window you were in when you ran the program:

```
hello world
1267650600228229401496703205376
```

For example, here’s how this script ran from a DOS command line on a Windows laptop, to make sure it didn’t have any silly typos:

```
D:\temp> python script1.p y
hello world
1267650600228229401496703205376
```

We’ve just run a Python script that prints a string and a number. We probably won’t win any programming awards with this code, but it’s enough to capture the basics of program execution.

Python’s View

The brief description of the prior section is fairly standard for scripting languages, and is usually all that most Python programmers need to know. You type code into text files, and run those files through the interpreter. Under the hood, though, a bit more happens when you tell Python to “go.” Although knowledge of Python internals is not strictly required for Python programming, a basic understanding of the runtime structure of Python can help you grasp the bigger picture of program execution.

When you instruct Python to run your script, there are a few steps that Python carries out before your code actually starts crunching away. Specifically, it’s first compiled to something called byte code, and then routed to something called a virtual machine.

Byte code compilation

Internally, and almost completely hidden from you, Python first compiles your *source code* (the statements in your file) into a format known as *byte code*. Compilation is simply a translation step, and byte code is a lower−level, and platform−independent, representation of your source code. Roughly, each of your source statements is translated into a group of byte code instructions. This byte code translation is performed to speed execution—byte code can be run much quicker than the original source code statements.

You’ll notice the prior paragraph said that this is *almost* completely hidden from you. If the Python process has write−access on your machine, it will store the byte code of your program in files that end with a *.pyc* extension (“.pyc” means compiled “.py” source). You will see these files show up on your computer after you’ve run a few programs. Python saves byte code like this as a startup speed optimization. The next time you run your program, Python will load the *.pyc* and skip the compilation step, as long as you haven’t changed your source code since the byte code was saved. Python automatically checks the time stamps of source and byte code files to know when it must recompile.

If Python cannot write the byte code files to your machine, your program still works—the byte code is generated in memory and simply discarded on program exit.* However, because *.pyc* files speed startup time, you'll want to make sure they are written for larger programs. Byte code files are also one way to ship Python pro−grams—Python is happy to run a program if all it can find are *.pyc* files, even if the original *.py* source files are absent. (See the section "Frozen Binaries" later in this chapter for another shipping option.)

* And strictly speaking, byte code is saved only for files that are imported, not for the top−level file of a program. We'll explore imports in Chapter 3, and again in Part V. Byte code is also never saved for code typed at the interactive prompt, which is described in Chapter 3.

Once your program has been compiled to byte code (or the byte code has been loaded from *.pyc* files), it is shipped off for execution to something generally known as the Python Virtual Machine (PVM, for the more acronym−inclined among you). The PVM sounds more impressive than it is; really, it's just a big loop that iterates through your byte code instructions, one by one, to carry out their operations. The PVM is the runtime engine of Python; it's always present as part of the Python system, and is the component that truly runs your scripts. Technically, it's just the last step of what is called the Python interpreter.

Figure 2−2 illustrates the runtime structure described. Keep in mind that all of this complexity is deliberately hidden to Python programmers. Byte code compilation is automatic, and the PVM is just part of the Python system that you have installed on your machine. Again, programmers simply code and run file your machine. Again, programmers simply code and run file of statements.

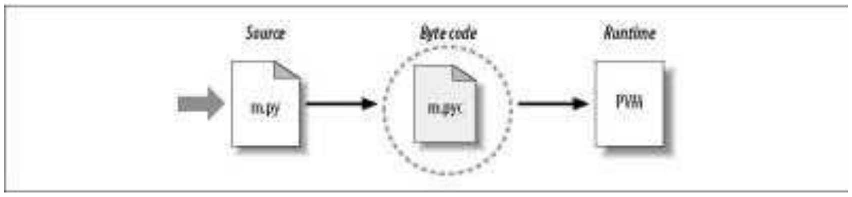


Figure 2−2. Runtime execution model

Performance implications

Readers with a background in fully compiled languages such as C and C++ might notice a few differences in the Python model. For one thing, there is usually no build or "make" step in Python work: code runs immediately after it is written. For another, Python byte code is not binary machine code (e.g., instructions for an Intel chip). Byte code is a Python−specific representation.

This is why some Python code may not run as fast as C or C++, as described in Chapter 1—the PVM loop, not the CPU chip, still must interpret the byte code, and byte code instructions require more work than CPU instructions. On the other hand, unlike classic interpreters, there is still a compile step internally—Python does not need to reanalyze and reparse each source statement repeatedly. The net effect is that pure Python code runs somewhere between a traditional compiled language, and a traditional interpreted language. See Chapter 1 for more on Python performance.

Development implications

Another ramification of Python's execution model is that there is really no distinction between the development and execution environments. That is, the systems that compile and execute your source code are really one in the same. This similarity may have a bit more significance to readers with a background in traditional compiled languages; but in Python, the compiler is always present at runtime, and is part of the system that runs programs.

This makes for a much more rapid development cycle—there is no need to precompile and link before execution may begin. Simply type and run the code. This also adds a much more dynamic flavor to the language—it is possible, and often very convenient, for Python programs to construct and execute other Python programs at runtime. The `eval` and `exec` built−ins, for instance, accept and run strings containing Python program code. This structure is also why Python lends itself to product customization—because Python code can be changed on the fly, users can modify the Python parts of a system onsite, without needing to have or compile the entire system's code.

Before moving on, we should point out that the internal execution flow described in the prior section reflects the standard implementation of Python today, and is not really a requirement of the Python language itself. Because of that, the execution model is prone to change with time. In fact, there are already a few systems that modify the picture in Figure 2−2 somewhat. Let's take a few moments to explore the most prominent of these variations.

Python Implementation Alternatives

Really, as this book is being written, there are two primary implementations of the Python language— *CPython* and *Jython*—along with a handful of secondary implementations such as *Python.net*. CPython is the standard implementation; all the others have very specific purposes and roles. All implement the same Python language, but execute programs in different ways.**CPython**

The original, and standard, implementation of Python is usually called CPython, when you want to contrast it with the other two. Its name comes from the fact that it is coded in portable ANSI C language code. This is the Python that you fetch from *www.python.org*, get with the ActivePython distribution, and have automatically in most Linux machines. If you've found a preinstalled version of Python on your machine, it's probably CPython as well, unless your company is using Python in very specialized ways.

Unless you want to script Java or .NET applications with Python, you probably want to use the standard CPython system. Because it is the reference implementation of the language, it tends to run fastest, be the most complete, and be more robust than the alternative systems. Figure 2−2 reflects CPython's runtime architecture.

Jython

The Jython system (originally known as JPython) is an alternative implementation of the Python language, targeted for integration with the Java programming language. Jython consists of Java classes that compile Python source code to Java byte code, and then route the resulting byte code to the Java Virtual Machine (JVM). Programmers still code Python statements in *.py* text files as usual; the Jython system essentially just replaces the rightmost two bubbles in Figure 2−2 with Java−based equivalents.

Jython's goal is to allow Python code to script Java applications, much as CPython allows Python to script C and C++ components. Its integration with Java is remarkably seamless. Because Python code is translated to Java byte code, it looks and feels like a true Java program at runtime. Jython scripts can serve as web applets and servlets, build Java−based

GUIs, and so on. Moreover, Jython includes integration support that allows Python code to import and use Java classes, as though they were coded in Python. Because Jython is slower and less robust than CPython, it is usually seen as a tool of interest primarily to Java developers.

Python.NET

A third, and still somewhat experimental implementation of Python, is designed to allow Python programs to integrate with applications coded to Microsoft's .NET framework. .NET and its C# programming language runtime system are designed to be a language−neutral object communication layer, in the spirit of Microsoft's earlier COM model. Python.NET allows Python programs to act as both client and server components, accessible from other .NET languages.

By implementation, Python.NET is very much like Jython—it replaces the last two bubbles in Figure 2−2 for execution in the .NET environment. Also like Jython, Python.NET has a special focus—it is primarily of interest to developers integrating Python with .NET components. (Python.NET's evolution is unclear as we write this; for more details, consult Python online resources.)

CPython, Jython, and Python.NET all implement the Python language in similar ways: by compiling source code to byte code, and executing the byte code on an appropriate virtual machine. The *Psyco* system is not another Python implementation, but a component that extends the byte code execution model to make programs run faster. In terms of Figure 2−2, Psyco is an enhancement to the PVM, which collects and uses type information while the program runs, to translate portions of the program's byte code all the way down to real binary machine code for faster execution. Psyco accomplishes this translation without requiring either changes to the code, or a separate compilation step during development.

Roughly, while your program runs, Psyco collects information about the kinds of objects being passed around; that information can be used to generate highly−efficient machine code tailored for those object types. Once generated, the machine code then replaces the corresponding part of the original byte code, to speed your program's overall execution. The net effect is that, with Psyco, your program becomes much quicker over time, and as it is running. In ideal cases, some Python code may become as fast as compiled C code under Psyco.

Because this translation from byte code happens at program runtime, Psyco is generally known as a *just−in−time* (JIT) compiler. Psyco is actually a bit more than JIT compilers you may have seen for the Java language. Really, Psyco is a *specializing JIT compiler*—it generates machine code tailored to the data types that your program actually uses. For example, if a part of your program uses different data types at different times, Psyco may generate a different version of machine code to support each different type combination.

Psyco has been shown to speed Python code dramatically. According to its web page, Psyco provides "2× to 100× speed−ups, typically 4×, with an unmodified Python interpreter and unmodified source code, just a dynamically loadable C extension module." Of equal significance, the largest speedups are realized for algorithmic code written in pure Python—exactly the sorts of code you might normally migrate to C to optimize. With Psyco, such migrations become even less important.

Psyco is also not yet a standard part of Python; you will have to fetch and install it separately. It is also still something of a research project, so you'll have to track its evolution online. For more details on the Psyco extension, and other JIT efforts that may arise, consult *http://www.python.org*; Psyco's home page currently resides at *http://psyco.sourceforge.net*.

Frozen Binaries

Sometimes when people ask for a “real” Python compiler, what they really seek is simply a way to generate a standalone binary executable from their Python programs. This is more a packaging and shipping idea than an execution−flow concept, but is somewhat related. With the help of third−party tools that you can fetch off the Web, it is possible to turn your Python programs into true executables—known as *frozen binaries* in the Python world.

Frozen binaries bundle together the byte code of your program files, along with the PVM (interpreter) and any Python support files your program needs, into a single package. There are some variations on this theme, but the end result can be a single binary executable program (e.g., an *.exe* file on Windows), which may be shipped easily to customers. In Figure 2−2, it is as though the byte code and PVM are merged into a single component—a frozen binary file.

Today, three primary systems are capable of generating frozen binaries: *Py2exe* (for Windows), *Installer* (similar, but works on Linux and Unix too, and is also capable of generating self−installing binaries), and *freeze* (the original). You may have to fetch these tools separately from Python itself, but they are available free of charge. They are also constantly evolving, so see *http://www.python.org* and the Vaults of Parnassus web site for more on these tools. To give you an idea of the scope of these systems, Py2exe can freeze standalone programs that use the Tkinter, Pmw, wxPython, and PyGTK GUI libraries; programs that use the *pygame* game programming toolkit; win32com client programs; and more.

Frozen binaries are not the same as the output of a true compiler—they run byte code through a virtual machine. Hence, apart from a possible startup improvement, frozen binaries run at the same speed as the original source files. Frozen binaries are also not small (they contain a PVM), but are not unusually large by current standards of large. Because Python is embedded in the frozen binary, Python does not have to be installed on the receiving end in order to run your program. Moreover, because your code is embedded in the frozen binary, it is effectively hidden from recipients.

This single file−packaging scheme is especially appealing to developers of commercial software. For instance, a Python−coded user interface program based on the Tkinter toolkit can be frozen into an executable file, and shipped as a self−contained program on CD or on the Web. End users do not need to install, or even have to know about, Python.

Finally, note that the runtime execution model sketched here is really an artifact of the current implementation, and not the language itself. For instance, it’s not impossible that a full, traditional compiler for Python source to machine code may appear during the shelf life of this book (although one has not in over a decade). New byte code formats and implementation variants may also be adopted in the future. For instance:

- The emerging *Parrot* project aims to provide a common byte code format, virtual machine, and optimization techniques, for a variety of programming languages (see *http://www.python.org*).
- The *Stackless Python* system is a standard CPython implementation variant, which does not save state on the C language call stack. This makes Python more easily ported to small stack architectures, and opens up novel programming possibilities such as co−routines.
- The new *PyPy* project is an attempt to reimplement the PVM in Python itself, in order to enable new implementation techniques.

Although such future implementation schemes may alter the runtime structure of Python somewhat, it seems likely that the byte code compiler will still be the standard for some time to come. The portability and runtime flexibility of byte code are important features to many Python systems. Moreover, adding type constraint declarations to support static compilation would break the flexibility, conciseness, simplicity, and overall spirit of Python coding. Due to Python’s highly dynamic nature, any future implementation will likely retain many artifacts of the current PVM.