UNITED STATES

12 Pythons for every programming need

Whether its speed, memory safety, portability, a micro footprint, data tools, or something else, one of these Python distros probably has it

By Serdar Yegulalp

Senior Writer, InfoWorld

SEP 25, 2019 3:00 AM PDT

When you choose Python for software development, you choose a large language ecosystem with a wealth of packages covering all manner of programming needs. But in addition to libraries for everything from GUI development to machine learning, you can also choose from a number of Python runtimes—and some of these runtimes may be better suited to the use case you have at hand than others.

Here is a brief tour of Python distributions, from the standard implementation (CPython) to versions optimized for speed (PyPy), for special use cases (Anaconda, ActivePython), for different language runtimes (Jython, IronPython), and even for cutting-edge experimentation (PyCopy, MesaPy).

[Find the best of Python on InfoWorld: The best new features in Python 3.8. • 24 Python libraries for every Python developer. • 13 Python web frameworks compared. • 5 essential Python tools for data science. | Keep up with hot topics in programming with InfoWorld's App Dev Report newsletter.]

Table of Contents

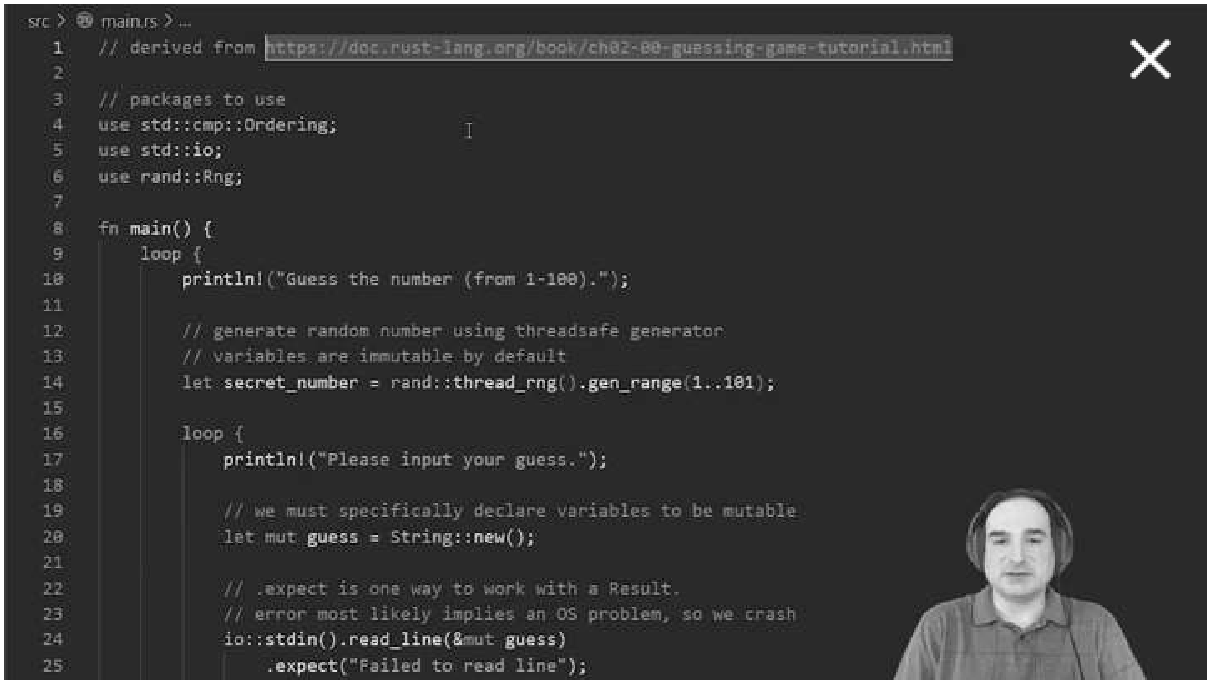
SHOW MORE

CPython

CPython is the reference implemenation of Python, the standard version that all other Python incarnations look to. CPython is written in C, as implied by the name, and it is produced by the same core group of people responsible for all of the top-level decisions about the Python language.

CPython use cases

Because CPython is the reference implementation of Python, it is the most conservative in terms of its optimizations. This is by design. Python's maintainers want CPython to be the most broadly compatible and standardized implementation of Python available.



```
// derived from https://doc.rust-lang.org/book/ch02-00-guessing-game-tutorial.html

// packages to use
use std::cmp::Ordering;
use std::io;
use rand::Rng;

fn main() {
    loop {
        println!("Guess the number (from 1-100).");

        // generate random number using threadsafe generator
        // variables are immutable by default
        let secret_number = rand::thread_rng().gen_range(1..101);

        loop {
            println!("Please input your guess.");

            // we must specifically declare variables to be mutable
            let mut guess = String::new();

            // .expect is one way to work with a Result.
            // error most likely implies an OS problem, so we crash
            io::stdin().read_line(&mut guess)
                .expect("Failed to read line");
```

CPython is your best choice when compatibility and conformity to Python standards matter more than raw performance and other concerns. CPython is also useful for the expert who wants to work with Python in its most fundamental incarnation, and who is willing to forgo certain conveniences.

For example, with CPython, you have to do a little more lifting to set up virtual environments. Other distros (Anaconda, in particular) provide more automation around workspace setup.

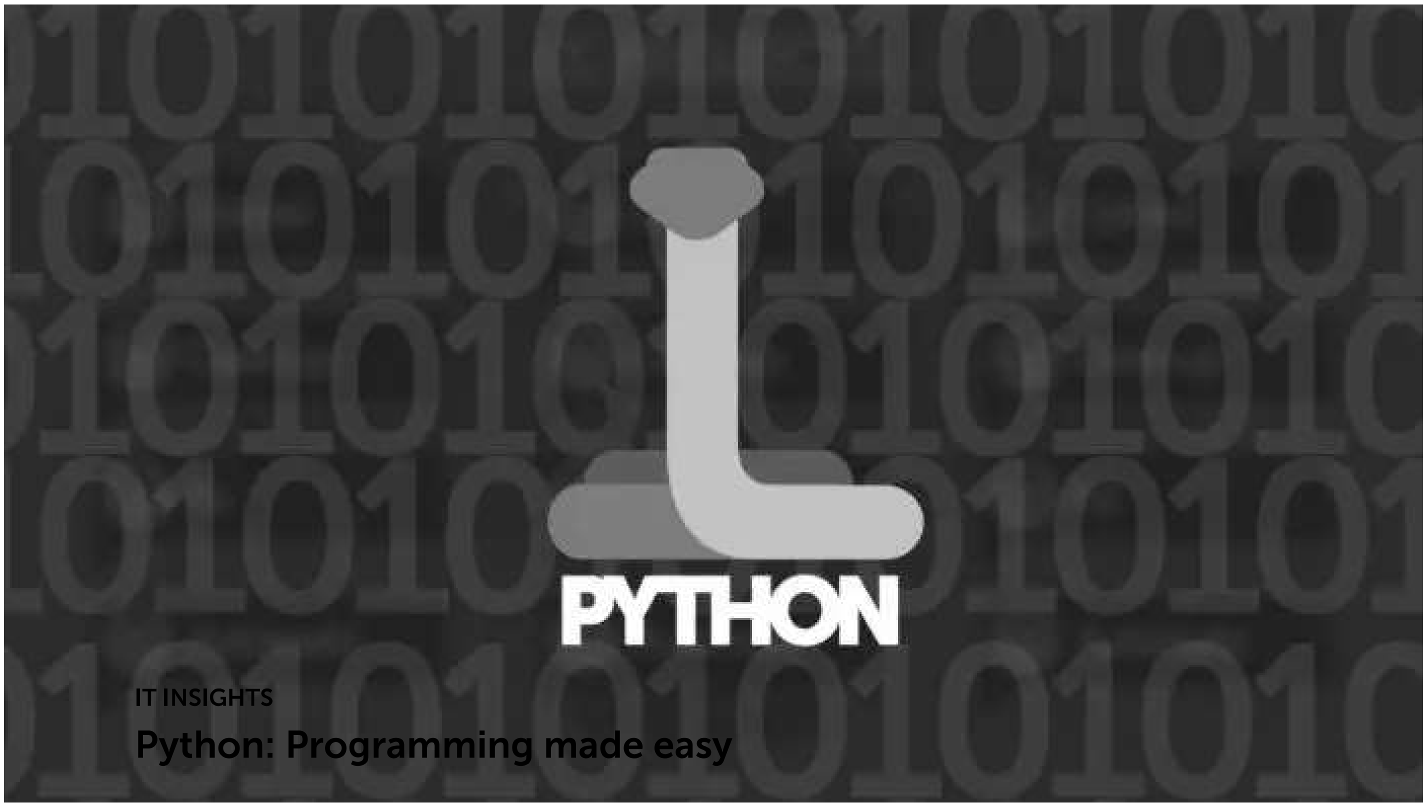
CPython limitations

CPython does not have the performance optimizations found in other editions of Python. There is no native JIT (just-in-time) compiler, no accelerated math libraries, and no third-party additions for the sake of performance. Those are all things you can add on your own, but they're not bundled. Again, all this is by design, to ensure maximum compatibility and to allow CPython to serve as a reference implementation, but it means any performance optimizations are up to the developer.

Further, CPython provides only a baseline set of tools for working with Python. The pip package manager, for instance, obtains and installs packages from Python's native PyPI package repository. Pip will even install precompiled binaries (via the wheel distribution format) if they are provided by the developer, but it won't install any dependencies that packages might have *outside* of PyPI.

Related video: How Python makes programming easier

Perfect for IT, Python simplifies many kinds of work, from system automation to working in cutting-edge fields like machine learning.



Anaconda Python

Anaconda, produced by Anaconda, Inc. (formerly Continuum Analytics), is designed for Python developers who need a distribution backed by a commercial provider and with support plans for enterprises. The chief use cases for Anaconda Python are math, statistics, engineering, data analysis, machine learning, and related applications.

Anaconda Python use cases

Anaconda bundles many of the most common libraries used in commercial and scientific Python work—SciPy, NumPy, Numba, and so on—and makes many more of them accessible via a custom package mamagement system.

Anaconda stands out from other distributions in how it integrates all these pieces. When installed, Anaconda provides a desktop app—the Anaconda Navigator—that makes every aspect of the Anaconda environment available through a convenient GUI. Finding components, keeping them up to date, and working with them is a good deal easier out of the box with Anaconda than with CPython.

Another boon is the way Anaconda handles components from outside the Python ecosystem if they're required for a specific package. The `conda` package manager, created specifically for Anaconda, handles installing both Python packages and third-party, external software requirements.

Anaconda Python limitations

Because Anaconda includes so many useful libraries, and can install even more with only a few keystrokes, the size of an Anaconda installation can be much larger than CPython. A basic CPython installation runs about 100MB; Anaconda installations can grow to gigabytes in size. This can be an issue in situations where you have resource constraints.

One way to reduce Anaconda's footprint is to install Miniconda, a stripped-down version of Anaconda that includes only the absolute minimum of pieces needed to get up and running. You can then add packages to Miniconda as you see fit, with an eye toward how much space each piece consumes.

ActivePython

Like Anaconda, ActivePython is created and maintained by a for-profit company—in this case, ActiveState, which markets a number of language runtimes along with the multi-language Komodo IDE.

ActivePython use cases

ActivePython is aimed at enterprise users and data scientists—people who want to use Python, but don't want to spend a lot of effort assembling and managing a Python installation. ActivePython uses Python's regular `pip` package manager, but also supplies a few hundred common libraries as verified pack-ins, along with some common libraries with third-party dependencies such as the Intel Math Kernel Library.

ActivePython limitations

There is one potential drawback to ActivePython's approach to handling packages with external dependencies. If you want to upgrade to a newer version of a project with complex dependencies (e.g., TensorFlow), you will need to upgrade your ActivePython installation as well. In environments where development tends to be tied to a specific version of a project, this is less of an issue. But in environments where development tends to track cutting-edge versions, it could present a problem.

PyPy

A drop-in replacement for the CPython interpreter, PyPy uses just-in-time (JIT) compilation to speed up the execution of Python programs. Depending on the task being performed, the performance gains can be dramatic.

PyPy use cases

A common complaint about Python generally, and CPython in particular, is speed. By default Python runs many times slower than C, sometimes hundreds of times slower. PyPy JIT-compiles Python code to machine language, providing a 7.7x speedup over CPython on average. Some tasks run as much as 50x faster.

The best part is that little to no effort is required on the part of the developer to unlock these gains. Swap out CPython for PyPy, and for the most part you're done.

PyPy limitations

PyPy has always performed best with "pure" Python applications. Python packages that interface with C libraries, such as NumPy, have not fared as well due to the way PyPy has emulated CPython's native binary interfaces. Over time, though, PyPy's developers have whittled away at this issue, and made PyPy far more compatible with the majority of Python packages that depend on C extensions. In short, support for C extensions is still limited, but far less so than it used to be.

Another possible downside with PyPy is the size of the runtime. The core CPython runtime on Windows, excluding the standard library, is around 4MB, while the PyPy runtime is around 32MB. Note too that PyPy has long emphasized the 2.x branch of Python, so, for example, PyPy for Python 3.x is currently available for Windows only in a 32-bit beta-test version. (PyPy is available in 64-bit versions for Python 2.x and 3.x for Linux and MacOS.)

Jython

The JVM (Java Virtual Machine) serves as the runtime for a great many languages besides Java. The long list includes Groovy, Scala, Clojure, Kotlin, and, yes, Python, by way of the Jython project.

Jython use cases

Jython compiles Python 2.x code to JVM bytecode and runs the resulting program on the JVM. In some cases a Jython-compiled program will run faster than its CPython counterpart, but not always.

The biggest advantage Jython provides is direct interoperability with the rest of the Java ecosystem. Java is used even more widely than Python. Running Python on the JVM allows Python developers to tap into an enormous ecosystem of libraries and frameworks that they otherwise wouldn't be able to use. By the same token, Jython allows Java developers to use Python libraries.

Jython limitations

The biggest drawback to Jython is that it supports only the 2.x branch of Python. Support for Python 3.x is under development but has been for some time. So far nothing has been released.

Note too that while Jython brings Python to the JVM, it does not bring Python to Android. As there is currently no port of Jython to Android proper, Jython can't be used to develop Android applications.

IronPython

Just as Jython is an implementation of Python on the JVM, IronPython is an implementation of Python on the .Net runtime, or CLR (Common Language Runtime). IronPython uses the DLR (Dynamic Language Runtime) of the CLR to allow Python programs to run with the same degree of dynamism that they do in CPython.

IronPython use cases

Like Jython, IronPython is a bridge. The big use case is interoperability between Python and the .Net universe. Existing .Net assemblies can be loaded in IronPython programs using Python's native import and object-manipulation syntax. It is also possible to compile IronPython code into an assembly and run it as-is or invoke it from other languages. However, note that the MSIL (Microsoft Intermediate Language) in the assembly cannot be directly accessed from other .Net languages, as it is not compliant with the Common Language Specification.

IronPython limitations

Like Jython, IronPython currently supports only Python 2.x. However, work is under way to create an IronPython 3.x implementation.

WinPython

As the name implies, WinPython is a Python distribution created specifically for users of Microsoft Windows. CPython's earlier editions for Windows were not well designed, and it was difficult for Windows users to take full advantage of the Python ecosystem. CPython's Windows edition has improved over time, but WinPython still offers many things not found in CPython.

WinPython use cases

WinPython's main attraction is that it's a self-contained edition of Python. It doesn't have to be installed on the machine where it runs; it just needs to be unpacked into a directory. This makes WinPython useful in cases where software can't be installed on a given system, in scenarios where a preconfigured Python runtime needs to be distributed along with the applications to run on it, or where multiple editions of Python need to run side by side without interfering with each other.

WinPython also bundles a slew of data science oriented packages—NumPy, Pandas, SciPy, Matplotlib, etc.—so they can be used right away, without additional installation steps. Also included is a C/C++ compiler, since many Windows machines don't have one included, and many Python extensions require or can make use of it.

WinPython limitations

One limitation of WinPython is that it might include too much by default for some use cases. To remedy that, WinPython's creators provide a "zero" version of each WinPython edition, containing only the most minimal possible install of the product. More packages can be added later, either with Python's own `pip` tool or WinPython's WPPM utility.

Python Portable

Python Portable is the CPython runtime in a self-contained package. It comes courtesy of the PortableDevApps collection of similarly self-contained applications.

Python Portable use cases

Like WinPython, Python Portable includes a slew of packages for scientific computing—Matplotlib, Numba, SymPy, SciPy, Cython, and others. Also like WinPython, Python Portable runs without needing to be formally installed on the Windows host; it can live in any directory or on a removable drive. Also included is the Spyder IDE and Python's pip package manager, so you can add, change, or remove packages as needed.

Python Portable limitations

Unlike WinPython, Python Portable does not include a C/C++ compiler. You'll need to provide a C compiler to make use of code written with Cython (and thus compiled to C).

Experimental Python distributions

These distributions make significant changes to Python—either because they're using Python as a starting point for something entirely new, or because they're making strategic changes to standard Python. By and large, these Pythons are not recommended for production use yet.

If you're living with a Python 2.x codebase for the foreseeable future, you might want to check out our article about the experimental Python distributions keeping Python 2.x alive.

MicroPython

MicroPython provides a minimal subset of the Python language that can run on extremely low-end hardware such as microcontrollers. MicroPython implements Python 3.4 with some differences. It's easy to write MicroPython code if you know Python, but existing code may not run as-is.

Pycopy

Page 1 of 2 ›

How to choose a low-code development platform