Pobrane z czasopisma Annales AI- Informatica **http://ai.annales.umcs.pl**
Data: 30/06/2024 19:24:58

Annales UMCS Informatica AI XI, 1 (2011) 33–41
DOI: 10.2478/v10065-011-0022-7

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI
http://www.annales.umcs.lublin.pl/

# Comparative evaluation of performance-boosting tools for Python

Jakub Swacha[1]*

[1] *Institute of Informatics Technology in Management, University of Szczecin, Mickiewicza 64,71-101 Szczecin*

**Abstract** – The Python programming language has a number of advantages, such as simple and clear syntax, concise and readable code, and open source implementation with a lot of extensions available, that makes it a great tool for teaching programming to students. Unfortunately, Python, as a very high level interpreted programming language, is relatively slow, which becomes a nuisance when executing computationally intensive programs. There is, however, a number of tools aimed at speeding-up execution of programs written in Python, such as Just-in-Time compilers and automatic translators to statically compiled programming languages. In this paper a comparative evaluation of such tools is done with a focus on the attained performance boost.

## 1 Introduction

Python [**1**] is a relatively new programming language praised for its educational capabilities [**2**]. It is simple: its keyword list is limited, and the syntax does not contain unnecessary formalities. Program lines do not end with semicolons, and there are no logical brackets (like begin / end of Pascal or braces of C), instead of which, Python uses indentation to control the course of program execution - in this way it avoids problems with the unpaired brackets and at the same time forces students to properly format the source code. Python does not require variables to be declared - a typical source of mistakes made by novice programmers. It is very easy in Python to use complex data types, such as lists and dictionaries. Thanks to that, Python makes possible very simple implementations of algorithms that use such data types, which would be much more difficult to implement in other languages. Python is a very concise and efficient language. According to S. McConnell's studies, a single Python

*jakubs@uoo.univ.szczecin.pl

Pobrane z czasopisma Annales AI- Informatica **http://ai.annales.umcs.pl**
Data: 30/06/2024 19:24:58


expression corresponds on average to six C expressions [**3**]. A similar rate was measured for Perl, which, in contrast to Python, is known for its hardly readable source code. Python has been ported to many system platforms and is available free from its official website [**4**].

The problem with Python is its slowness. Antonio Cuni points the following reasons for which Python is intrinsically slow [**5**]:

1. Interpretation overhead (due to code translation);
2. Boxed arithmetic and automatic overflow handling (even simple data types are treated as objects);
3. Dynamic dispatch of operations (the types of arguments are identified on runtime);
4. Dynamic lookup of methods and attributes (the class components are identified on runtime);
5. "The world can change under your feet" (classes and functions can be defined, undefined and redefined on runtime);
6. Extreme introspective and reflective capabilities (that allow not only inspection of a running program environment but also its modification).

The slowness often becomes a nuisance, especially when implementing classic algorithms, which often consist of large number of repetitions of simple operations. When developing industrial applications, one would resort to another programming language, but in the case of education, when one wants to keep with Python for its other advantages, he or she has to look for tools that can make Python programs run faster.

## 2 The competing solutions

The solution to the slowness of Python is compilation. Although some of the language features make it virtually impossible to compile every line of every Python program, it can be done in most cases.

The compilation may be Ahead-Of-Time (AOT), or Just-In-Time (JIT). The AOT compilation is done once and is always static (based merely on the source code analysis). The JIT compilation is done every time the program is run and may be dynamic (adjusting the compiled code based on its actual execution).

Currently, there are seven popular tools that seem to handle this problem in some way:

1. Cython [**6**], a programming language based on Python that can be automatically translated to C or C++ and then compiled.
2. Iron Python [**7**], a .NET-based implementation of Python and as such making use of the .NET's JIT compiler.
3. Jython [**8**], a Java-based implementation of Python and as such making use of the Java Virtual Machine (JVM)'s JIT compiler.
4. Psyco [**9**], a Python module capable of compiling Python functions using a specialized JIT compiler.

Pobrane z czasopisma Annales AI- Informatica **http://ai.annales.umcs.pl**
Data: 30/06/2024 19:24:58



5. PyPy [**5**], an RPython-based implementation of Python, capable of automatically generating tracing JIT compilers.
6. Shedskin [**10**], a tool that can automatically translate Python code to C++ and then compile it.
7. Unladen Swallow [**11**], a C-based implementation of Python, making use of the Low Level Virtual Machine (LLVM [**12**]) JIT compiler.

There are also other, less popular tools of similar type, such as Pyrex [**13**] and py2llvm [**14**]. They were not included in the comparison; the former, because it is closely related to Cython [**6, 13**], the latter due to its very early stage of development [**14**].

Table 1 lists the most important features of the seven Python performance boosters mentioned above.

Table 1. Comparison of Python performance boosters.

| **Tool** | **Cython** | **Iron Python** | **Jython** | **Psyco** |
|---|---|---|---|---|
| Version | 0.14 | 2.6.1 | 2.5.2RC3 | 1.6 |
| CPython | 2.6.1 | 2.6.1 | 2.5.2 | 2.5.4 |
| What it is | Programming language | Python implementation | Python implementation | Python library |
| Compiler | AOT | JIT | JIT | JIT |
| Technology | C/C++ | .NET | Java | Proprietary |
| Compatibility | Almost ful | l Full | Full | Full |
| Code adaptation | Needed | Not needed | Not needed | Very slight |
| Required software | CPython, C++ compiler | .NET Framework | JVM | CPython |
| Disk space | 6.2 | 8.3 | 50.4 | 0.3 |

| **Tool** | **Pypy** | **Shedskin** | **Unladen Swallow** |
|---|---|---|---|
| Version | 1.4 | 0.7 | 2009Q4 |
| CPython | 2.5.2 | 2.6.1 | 2.6.1 |
| What it is | Python implementation | Translation tool | Python implementation |
| Compiler | JIT | AOT | JIT |
| Technology | Proprietary | C++ | LLVM |
| Compatibility | Full | Limited | Full |
| Code adaptation | Not needed | May be needed | Not needed |
| Required software | - | CPython | C++ compiler and tools, LLVM |
| Disk space | 46.9 | 128.0 | 3450.5 |

Remark: disk space given in megabytes, calculated for 4 KB disk cluster size.

The "Version" row shows the version used in the tests (in most cases the last released version). The "CPython" row gives the CPython (the reference Python implementation in C) version that the booster is based on (or compatible with); the general idea was

Pobrane z czasopisma Annales AI- Informatica **http://ai.annales.umcs.pl**
Data: 30/06/2024 19:24:58


to use CPython 2.6.1 as a reference, still not every booster has been released for that version of CPython. The next row tells what the booster actually is; the "Compiler" row shows the type of the compiler used, and the following row lists the technologies which are behind the speed-up. Row "Compatibility" tells whether the boost can be applied to every Python program, whereas the "Code adaptation" row indicates to what extent a program's source code must be modified for a booster to work. The one before last row lists the software required to use the booster (apart from what is included with the booster itself), and the bottom row of the table shows the disk space taken up by the installation of a booster. Notice that it is the total disk space used (not the sum of file sizes, as most of the boosters consist of a large number of small files, which, due to disk space clustering, take up much more disk space than the sum of their component file sizes), and without taking any further steps (as the installation of some of the tools leaves a lot of files which are not required for them to run, yet they are not deleted automatically).

## 3 Installation

Microsoft Windows has been assumed to be the target platform, as it was found by this author to be the operating system family most popular among his students currently.

Cython can be downloaded in a form of a Windows installer; after running it, the user has just to confirm the Python version for which it is installed (as it is a Python extension module). Iron Python can also be downloaded as a Windows installer, which installs a Python instance; the only prerequisite is having installed .NET Framework.

Jython is a similar case, of course Java Virtual Machine is required here instead of .NET Framework. Psyco, like Cython, is a Python module which has its own Windows installer. Again, confirming the Python version is the only user's input needed for installation.

PyPy can be downloaded as a Zip archive containing Windows executables that require no further installation. Shedskin is also distributed as a Windows installer, containing all necessary components, including MinGW C++ compiler.

Unladen Swallow was found to be the most cumbersome to install of all the tested software. First of all, it is only available as a source code within a Subversion repository. As Unladen Swallow is developed for the Linux systems, the files have to be converted to the Windows-based compiler project files using CMake utility. Unladen Swallow requires LLVM, yet it is not included in its repository, therefore it also has to be downloaded as source and compiled. Finally, Unladen Swallow has to be compiled to produce a JIT-compilation-enabled Python instance. It means that, in order to use Unladen Swallow, the end-user has to install Subversion, CMake and C++ compiler unless he or she is a C++ developer and has these components already installed.

Pobrane z czasopisma Annales AI- Informatica **http://ai.annales.umcs.pl**
Data: 30/06/2024 19:24:58



## 4 Usage

Cython is designed as a tool for library development, therefore the code to be compiled must be provided as a separate module. Cython source files have `PYX` extension, they can be compiled into C (or, optionally, C++) code by calling the Python's `setup` function (from `distutils` module) with appropriate parameters; an exemplary Python code is included with Cython distribution. The built module has a `PYD` extension and can be imported from within a Python program.

Psyco is a module which has to be imported into the Python program to make the JIT compiler available. Only functions can be compiled (not the main program code), this little drawback can obviously be worked around by moving a complicated main code into a separate function. There is the `full` function that run once compiles all the program functions.

Shedskin provides a shell script (`init.bat`) that sets up the environment for its usage. After doing it, one has to run `shedskin.exe` which analyzes the specified Python program file in order to determine the used data types and then translates the source code to C++. This process takes quite a long time – sometimes many seconds even for simple programs. Finally, one has to run the bundled `make.exe` in order to build the CPP and HPP files resulting from the previous step into an executable file that can be run directly from the operating system environment.

The remaining solutions: Iron Python, Jython, PyPy, Unladen Swallow are all instances of Python, and their usage is the same as CPython's: one has to run the main interpreter file, specifying the file with the source code of a program to be run as the first parameter. The JIT compilation is turned on by default.

## 5 Performance test methodology

The core idea behind this research was to test the performance of Python-native code after having applied the boost. We wanted to test classic algorithms implementations rather than industrial applications, to simulate the speed perceived by a student implementing and testing such algorithms. Seven such programs, that were found to work correctly with each of the tested boosters, were selected for the tests. Their list, including name, description of the algorithm and its main parameters, as well as the most time-intensive operations, is given in Table 2.

The *LZ* and *Sieve* programs come from the Shedskin's set of examples [**10**], the remaining ones were implemented by this author.

None of the tested programs was modified in any way to suit better any of the boosters, apart from the modifications that were necessary for a booster to work at all.

The time was measured from inside the tested program, using the clock function, i.e., it does not include the time spent either on interpreter startup or compilation. The time was measured for two consecutive runs of each program, and the smaller

Pobrane z czasopisma Annales AI- Informatica **http://ai.annales.umcs.pl**
Data: 30/06/2024 19:24:58


Table 2. List of programs used in performance tests.

| **Programl** | **Description** | **Main elements** |
|---|---|---|
| *BST* | Binary-Search-Tree insert (50000), find (10000) and delete (5000) elements | Recursion, object handling |
| *Fibonacci* | Fibonacci string generator (to 33) | Deep recursion |
| *Knight* | Chess Knight problem (6x6 board) | Recursion, handling lists of lists |
| *LZ* | Lempel-Ziv'77 file compressor and decompressor (10 KB) | Bit operations, I/O operations |
| *Queens* | Chess Queens problem (27x27 board) | List operations, embedded loops |
| *Sieve* | Sieve of Atkin (to 20000000) | Handling long lists, long loops |
| *Sort* | Quicksort (500000 elements) | Handling long lists, recursion |

measurement was registered. The test was supposed to be repeated if the difference between the two measurements exceeded 10%, but such situation did not take place.

## 6 Performance test results

The tests were performed on a Intel Core2Duo 6420 2.13 GHz machine with 2 GB of RAM under the Windows XP SP3 operating system.

The measured execution times are shown in Fig. 1.

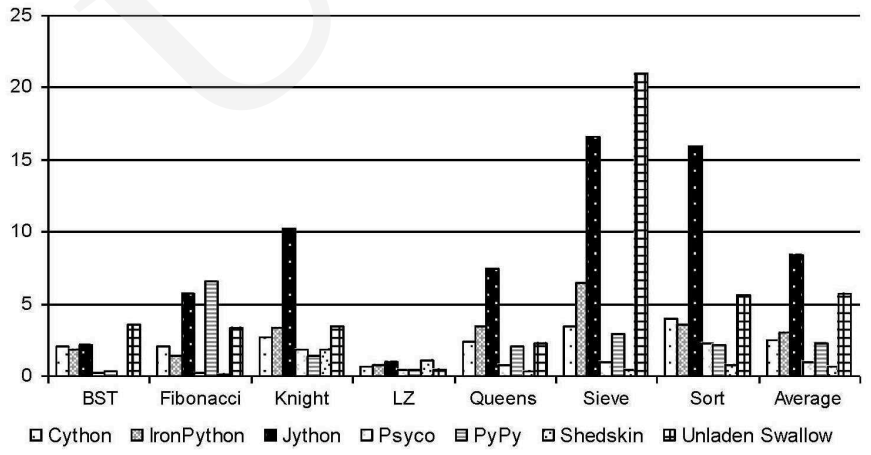


Fig. 1. Measured execution times.

Looking at the average results, it can be clearly seen that the two solutions: Psyco and Shedskin attained times much better than the others whereas the longest execution times were measured for Jython and Unladen Swallow.

Pobrane z czasopisma Annales AI- Informatica **http://ai.annales.umcs.pl**
Data: 30/06/2024 19:24:58



More detailed results are given in the tables to follow. They present the relative speed-up (the difference between the reference execution time and the boosted execution time divided by the reference execution time) attained by the tested performance boosting tools (see Table 3), and by different versions of CPython (see Table 4), all compared to CPython version 2.6.1.

Table 3. Measured relative execution speed-up: Python performance boosters.

| **Tool** | **Cython** | **Iron Python** | **Jython** | **Psyco** | **PyPy** | **Shedskin** | **Unladen Swallow** |
|---|---|---|---|---|---|---|---|
| *BST* | 6.3% | 16.2% | 3.6% | 91.9% | 83.8% | 98.6% | -61.3% |
| *Fibonacci* | 16.1% | 45.6% | -128.2% | 92.3% | -164.5% | 96.8% | -34.7% |
| *Knight* | 9.0% | -10.0% | -239.2% | 40.2% | 51.8% | 40.9% | -13.3% |
| *LZ* | 0.0% | -21.7% | -66.7% | 25.0% | 33.3% | -81.7% | 23.3% |
| *Queens* | 15.1% | -25.2% | -169.1% | 71.2% | 27.0% | 88.1% | 18.3% |
| *Sieve* | 22.4% | -45.4% | -272.0% | 78.1% | 35.8% | 89.3% | -368.9% |
| *Sort* | 17.2% | 26.6% | -226.6% | 53.9% | 56.6% | 84.6% | -15.8% |
| Average | 12.3% | -2.0% | -156.9% | 64.7% | 17.7% | 59.5% | -64.6% |

Table 4. Measured relative execution speed-up: CPython versions.

| **CPython** | **2.4.4** | **2.5.4** | **2.6.1** | **2.7** | **3.0** | **3.1** | **3.2** |
|---|---|---|---|---|---|---|---|
| *BST* | -27.9% | -60.4% | 0.0% | 0.0% | 70.7% | 70.7% | 66.2% |
| *Fibonacci* | -18.5% | -12.5% | 0.0% | -2.0% | -12.9% | -19.8% | -21.0% |
| *Knight* | -35.5% | -36.2% | 0.0% | -5.3% | 10.6% | 8.6% | -20.6% |
| *LZ* | -45.0% | 26.7% | 0.0% | -1.7% | -15.0% | -43.3% | -45.0% |
| *Queens* | -15.1% | -30.6% | 0.0% | 7.6% | 3.2% | 14.0% | 9.0% |
| *Sieve* | -11.4% | -17.4% | 0.0% | 6.3% | -81.4% | -83.4% | -53.0% |
| *Sort* | -18.0% | -21.9% | 0.0% | 3.9% | -32.6% | -29.3% | -39.3% |
| Average | -24.5% | -21.8% | 0.0% | 1.2% | -8.2% | -11.8% | -14.8% |

The first observation is that the results presented in Table 3, with the exception of Psyco, are noticeably different from those presented in the literature (or respective booster documentation), especially to those given by the authors of the tested solutions. Iron Python, instead of significant speed-up, produced a slight slow-down compared to CPython. PyPy, instead of being faster than Psyco, was found to be slower on four of the seven tested programs, and its performance on Fibonacci was very disappointing. Psyco produced the most consistent results; Shedskin, although significantly faster for most programs, stumbled on LZ. The performance of Unladen Swallow was the most disappointing, with a performance boost measured only for two programs, and a shocking performance degradation in the case of *Sieve*.

As for the performance of different CPython versions, one can notice a great improvement attained with version 2.6.1. The new line of Python (versions 3.x) has noticeably improved object handling performance (the case of BST), still the average performance worsened.

Pobrane z czasopisma Annales AI- Informatica **http://ai.annales.umcs.pl**
Data: 30/06/2024 19:24:58


## 7 Conclusions

The obtained results show that the performance boosters can help improve the execution speed of Python programs. The boosted programs worked even 70 times as fast (Shedskin on BST), with an average speed-up factor of about 3 (Psyco). Even though Psyco is one of the oldest performance boosters for Python, in our tests it achieved the results boldly superior to the solutions that were supposed to supersede it (PyPy and Unladen Swallow), and thus remains the best solution for the assumed purpose (speeding up classic algorithms implementations in the context of computer science/programming education). Although Shedskin peformed better than Psyco on all but one of the tested programs, its static translation approach with long compilation time and frequent compatibility issues excludes it from practical usage for the aforementioned purpose.

Presumably, better results could be obtained at least from Cython and Iron Python if the source code was specially prepared for these boosters, still it was not within the scope of this research, which aimed at examining the improvement of execution speed of native Python code without any unnecessary modifications.

## References

[1] Lutz M., Programming Python, O'Reilly, Sebastopol, CA, USA (2001).
[2] Swacha J., New concepts for teaching computer programming to future Information Technology engineers, [in:] Perspective technologies and methods in MEMS design, Lviv Politechnic National University, Lviv, Ukraine (2010): 188.
[3] McConnell S., Code complete: a practical handbook of software construction, Microsoft Press, Redmond, WA, USA (1993).
[4] Python Programming Language - Official Website, http://www.python.org (Visited 2010-12-10).
[5] Cuni A., High performance implementation of Python for CLI/.NET with JIT compiler generation for dynamic languages, Universita di Genova, Genoa, Italy (2010).
[6] Behnel S., Bradshaw R., Seljebotn D. S., Cython: C-Extensions for Python, http://cython.org (Visited 2010-12-10).
[7] Foord M. J., Muirhead Ch., IronPython in Action, Manning Publications, Greenwich, CT, USA (2009).
[8] Pedroni S., Rappin N., Jython Essentials, O'Reilly, Sebastopol, CA, USA (2002).
[9] Rigo A., Representation-Based Just-In-Time Specialization and the Psyco Prototype for Python, [in:] Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, ACM Press, Washington, DC, USA (2004): 15.
[10] Dufour M., Shedskin - An experimental (restricted) Python-to-C++ compiler, http://code.google.com/p/shedskin (Visited 2010-12-10).
[11] Winter C., Yasskin J., Unladen-swallow - A faster implementation of Python, http://code.google.com/p/unladen-swallow (Visited 2010-12-10).
[12] Lattner Ch., Adve V., LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, [in:] Proceedings of the 2004 International Symposium on Code Generation and Optimization, IEEE CS, Palo Alto, CA, USA (2004): 75.
[13] Ewing G., Pyrex - a Language for Writing Python Extension Modules, http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex (Visited 2010-12-10).

Pobrane z czasopisma Annales AI- Informatica **http://ai.annales.umcs.pl**
Data: 30/06/2024 19:24:58



[14] Fujita S., Py2llvm translates Python syntax into LLVM IR, http://code.google.com/p/py2llvm (Visited 2010-12-10).

UMCS