Shed Skin Documentation

Release v0.9.4

Mark Dufour
the Shed Skin contributors

Mar 16, 2017

Contents

CHAPTER 1

An experimental (restricted-Python)-to-C++ compiler

Shed Skin is an *experimental* compiler, that can translate pure, but *implicitly statically* typed Python (2.4-2.6) programs into optimized C++. It can generate stand-alone programs or extension modules that can be imported and used in larger Python programs.

Besides the typing restriction, programs cannot freely use the Python standard library (although about 25 common modules, such as `random` and `re`, are currently supported). Also, not all Python features, such as nested functions and variable numbers of arguments, are supported.

For a set of a 75 non-trivial programs (at over 25,000 lines in total (sloccount)), measurements show a typical speedup of 2-200 times over CPython.

CHAPTER 2

Documentation

Shed Skin documentation

Introduction

Shed Skin is an experimental Python-to-C++ compiler designed to speed up the execution of computation-intensive Python programs. It converts programs written in a restricted subset of Python to C++. The C++ code can be compiled to executable code, which can be run either as a standalone program or as an extension module easily imported and used in a regular Python program.

Shed Skin uses type inference techniques to determine the implicit types used in a Python program, in order to generate the explicit type declarations needed in a C++ version. Because C++ is statically typed, Shed Skin requires Python code to be written such that all variables are (implicitly!) statically typed.

Besides the typing and subset restrictions, supported programs cannot freely use the Python standard library, although about 25 common modules are supported, such as `random` and `re` (see *Library limitations*).

Additionally, the type inference techniques employed by Shed Skin currently do not scale very well beyond several thousand lines of code (the largest compiled program is about 6,000 lines (sloccount)). In all, this means that Shed Skin is currently mostly useful to compile smallish programs and extension modules, that do not make extensive use of dynamic Python features or the standard or external libraries. See here for a collection of 75 non-trivial example programs.

Because Shed Skin is still in an early stage of development, it can also improve a lot. At the moment, you will probably run into some bugs when using it. Please report these, so they can be fixed!

At the moment, Shed Skin is compatible with Python versions 2.4 to 2.7, behaves like 2.6, and should work on Windows and most UNIX platforms, such as GNU/Linux and OSX.

Typing restrictions

Shed Skin translates pure, but implicitly statically typed, Python programs into C++. The static typing restriction means that variables can only ever have a single, static type. So, for example,

```
a = 1
a = '1' # bad
```

is not allowed. However, as in C++, types can be abstract, so that for example,

```
a = A()
a = B() # good
```

where A and B have a common base class, is allowed.

The typing restriction also means that the elements of some collection (list, set, etc.) cannot have different types (because their subtype must also be static). Thus:

```
a = ['apple', 'b', 'c'] # good
b = (1, 2, 3) # good
c = [[10.3, -2.0], [1.5, 2.3], []] # good
```

is allowed, but

```
d = [1, 2.5, 'abc'] # bad
e = [3, [1, 2]] # bad
f = (0, 'abc', [1, 2, 3]) # bad
```

is not allowed. Dictionary keys and values may be of different types:

```
g = {'a': 1, 'b': 2, 'c': 3} # good
h = {'a': 1, 'b': 'hello', 'c': [1, 2, 3]} # bad
```

In the current version of Shed Skin, mixed types are also permitted in tuples of length two:

```
a = (1, [1]) # good
```

In the future, mixed tuples up to a certain length will probably be allowed.

None may only be mixed with non-scalar types (i.e., not with int, float, bool or complex):

```
l = [1]
l = None # good

m = 1
m = None # bad

def fun(x = None): # bad: use a special value for x here, e.g. x = -1
    pass
fun(1)
```

Integers and floats can usually be mixed (the integers become floats). Shed Skin should complain in cases where they can't.

Python subset restrictions

Shed Skin will only ever support a subset of all Python features. The following common features are currently not supported:

- `eval`, `getattr`, `hasattr`, `isinstance`, anything really dynamic
- arbitrary-size arithmetic (integers become 32-bit (signed) by default on most architectures, see *Command-line options*)

- argument (un)packing (`*args` and `**kwargs`)
- multiple inheritance
- nested functions and classes
- unicode
- inheritance from builtins (excluding `Exception` and `object`)
- overloading `__iter__`, `__call__`, `__del__`
- closures

Some other features are currently only partially supported:

- class attributes must always be accessed using a class identifier:

```
self.class_attr # bad
SomeClass.class_attr # good
SomeClass.some_static_method() # good
```

- function references can be passed around, but not method references or class references, and they cannot be contained:

```
var = lambda x, y: x+y # good
var = some_func # good
var = self.some_method # bad, method reference
var = SomeClass # bad
[var] # bad, contained
```

Library limitations

At the moment, the following 25 modules are largely supported. Several of these, such as `os.path`, were compiled to C++ using Shed Skin.

- `array`
- `binascii`
- `bisect`
- `collections` (defaultdict, deque)
- `colorsys`
- `ConfigParser` (no SafeConfigParser)
- `copy`
- `csv` (no Dialect, Sniffer)
- `datetime`
- `fnmatch`
- `getopt`
- `glob`
- `heapq`
- `itertools` (no starmap)
- `math`

- `mmap`
- `os` (some functionality missing on Windows)
- `os.path`
- `random`
- `re`
- `select` (only select function, on UNIX)
- `socket`
- `string`
- `struct` (no Struct, pack_into, unpack_from)
- `sys`
- `time`

Note that any other module, such as `pygame`, `pyqt` or `pickle`, may be used in combination with a Shed Skin generated extension module. For examples of this, see the Shed Skin examples.

See *How to help out in development* on how to help improve or add to the set of supported modules.

Installation

There are two types of downloads available: a self-extracting **Windows** installer and a **UNIX** tarball. But preferrably of course, Shed Skin is installed via your **GNU/Linux** package manager (Shed Skin is available in at least **Debian**, **Ubuntu**, **Fedora** and **Arch**).

Windows

To install the **Windows** version, simply download and start it. If you use **ActivePython** or some other non-standard Python distribution, or **MingW**, please deinstall this first. Note also that the 64-bit version of Python seems to be lacking a file, so it's not possible to build extension modules. Please use the 32-bit version instead.

UNIX

Using a package manager

Example command for when using Ubuntu:

```
sudo apt-get install shedskin
```

Manual installation

To manually install the UNIX tarball, take the following steps:

- download and unpack tarball
- run:

```
sudo python setup.py install
```

Dependencies

To compile and run programs produced by shedskin the following libraries are needed:

- g++, the C++ compiler (version 4.2 or higher).
- pcre development files
- Python development files
- Boehm garbage collection

To install these libraries under Ubuntu, type:

```
sudo apt-get install g++ libpcre++-dev python-all-dev libgc-dev
```

If the Boehm garbage collector is not available via your package manager, the following is known to work. Download for example version 7.2alpha6 from the website, unpack it, and install it as follows:

```
./configure --prefix=/usr/local --enable-threads=posix --enable-cplusplus --enable-
↪thread-local-alloc --enable-large-config
make
make check
sudo make install
```

If the PCRE library is not available via your package manager, the following is known to work. Download for example version 8.12 from the website, unpack it, and build as follows:

```
./configure --prefix=/usr/local
make
sudo make install
```

OSX

Manual installation

To install the UNIX tarball on an **OSX** system, take the following steps:

- download and unpack tarball
- run:

```
sudo python setup.py install
```

Dependencies

To compile and run programs produced by shedskin the following libraries are needed:

- g++, the C++ compiler (version 4.2 or higher; comes with the Apple XCode development environment?)
- pcre development files
- Python development files
- Boehm garbage collection

If the Boehm garbage collector is not available via your package manager, the following is known to work. Download for example version 7.2alpha6 from the website, unpack it, and install it as follows:

```
./configure --prefix=/usr/local --enable-threads=posix --enable-cplusplus --enable-
↪thread-local-alloc --enable-large-config
make
make check
sudo make install
```

If the PCRE library is not available via your package manager, the following is known to work. Download for example version 8.12 from the website, unpack it, and build as follows:

```
./configure --prefix=/usr/local
make
sudo make install
```

Compiling a standalone program

Under Windows, first execute (double-click) the `init.bat` file in the directory where you installed Shed Skin.

To compile the following simple test program, called `test.py`:

```
print 'hello, world!'
```

Type:

```
shedskin test
```

This will create two C++ files, called `test.cpp` and `test.hpp`, as well as a `Makefile`.

To create an executable file, called `test` (or `test.exe`), type:

```
make
```

Generating an extension module

To compile the following program, called `simple_module.py`, as an extension module:

```
# simple_module.py

def func1(x):
    return x+1

def func2(n):
    d = dict([(i, i*i)  for i in range(n)])
    return d

if __name__ == '__main__':
    print func1(5)
    print func2(10)
```

Type:

```
shedskin -e simple_module
make
```

For 'make' to succeed on a non-Windows system, make sure to have the Python development files installed (under **Debian**, install `python-dev`; under **Fedora**, install `python-devel`).

Note that for type inference to be possible, the module must (indirectly) call its own functions. This is accomplished in the example by putting the function calls under the `if __name__=='__main__'` statement, so that they are not executed when the module is imported. Functions only have to be called indirectly, so if func2 calls func1, the call to func1 can be omitted.

The extension module can now be simply imported and used as usual:

```
>>> from simple_module import func1, func2
>>> func1(5)
6
>>> func2(10)
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Limitations

There are some important differences between using the compiled extension module and the original.

1. Only builtin scalar and container types (`int`, `float`, `complex`, `bool`, `str`, `list`, `tuple`, `dict`, `set`) as well as `None` and instances of user-defined classes can be passed/returned. So for instance, anonymous functions and iterators are currently not supported.
2. Builtin objects are completely converted for each call/return from Shed Skin to CPython types and back, including their contents. This means you cannot change CPython builtin objects from the Shed Skin side and vice versa, and conversion may be slow. Instances of user-defined classes can be passed/returned without any conversion, and changed from either side.
3. Global variables are converted once, at initialization time, from Shed Skin to CPython. This means that the value of the CPython version and Shed Skin version can change independently. This problem can be avoided by only using constant globals, or by adding getter/setter functions.
4. Multiple (interacting) extension modules are not supported at the moment. Also, importing and using the Python version of a module and the compiled version at the same time may not work.

Numpy integration

Shed Skin does not currently come with direct support for Numpy. It is possible however to pass a Numpy array to a Shed Skin compiled extension module as a list, using its `tolist` method. Note that this is very inefficient (see above), so it is only useful if a relatively large amount of time is spent inside the extension module. Consider the following example:

```
# simple_module2.py

def my_sum(a):
    """ compute sum of elements in list of lists (matrix) """
    h = len(a) # number of rows in matrix
    w = len(a[0]) # number of columns
    s = 0.0
    for i in range(h):
        for j in range(w):
            s += a[i][j]
    return s

if __name__ == '__main__':
    print my_sum([[1.0, 2.0], [3.0, 4.0]])
```

After compiling this module as an extension module with Shed Skin, we can pass in a Numpy array as follows:

```
>>> import numpy
>>> import simple_module2
>>> a = numpy.array(([1.0, 2.0], [3.0, 4.0]))
>>> simple_module2.my_sum(a.tolist())
10.0
```

Distributing binaries

Windows

To use a generated Windows binary on another system, or to start it without having to double-click `init.bat`, place the following files into the same directory as the binary:

- `shedskin-0.9\shedskin\gc.dll`
- `shedskin-0.9\shedskin-libpcre-0.dll`
- `shedskin-0.9\bin\libgcc_s_dw-1.dll`
- `shedskin-0.9\bin\libstdc++.dll`

UNIX

To use a generated binary on another system, make sure `libgc` and `libpcre3` are installed there. If they are not, and you cannot install them globally, you can place copies of these libraries into the same directory as the binary, using the following approach:

```
$ ldd test
libgc.so.1 => /usr/lib/libgc.so.1
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3
$ cp /usr/lib/libgc.so.1 .
$ cp /lib/x86_64-linux-gnu/libpcre.so.3 .
$ LD_LIBRARY_PATH=. ./test
```

Note that both systems have to be 32- or 64-bit for this to work. If not, Shed Skin must be installed on the other system, to recompile the binary.

Multiprocessing

Suppose we have defined the following function in a file, called `meuk.py`:

```
def part_sum(start, end):
    """ calculate partial sum """
    sum = 0
    for x in xrange(start, end):
        if x % 2 == 0:
            sum -= 1.0 / x
        else:
            sum += 1.0 / x
    return sum

if __name__ == '__main__':
    part_sum(1, 10)
```

To compile this into an extension module, type:

```
shedskin -e meuk
make
```

To use the generated extension module with the `multiprocessing` standard library module, simply add a pure-Python wrapper:

```
from multiprocessing import Pool

def part_sum((start, end)):
    import meuk
    return meuk.part_sum(start, end)

pool = Pool(processes=2)
print sum(pool.map(part_sum, [(1,10000000), (10000001, 20000000)]))
```

Calling C/C++ code

To call manually written C/C++ code, follow these steps:

- Provide Shed Skin with enough information to perform type inference, by providing it with a *type model* of the C/C++ code. Suppose we wish to call a simple function that returns a list with the n smallest prime numbers larger than some number. The following type model, contained in a file called `stuff.py`, is sufficient for Shed Skin to perform type inference:

```
#stuff.py

def more_primes(n, nr=10):
    return [1]
```

- To actually perform type inference, create a test program, called `test.py`, that uses the type model, and compile it:

```
#test.py

import stuff
print stuff.more_primes(100)
```

```
shedskin test
```

- Besides `test.py`, this also compiles `stuff.py` to C++. Now you can fill in manual C/C++ code in `stuff.cpp`. To avoid that it is overwritten the next time `test.py` is compiled, move `stuff.*` to the Shed Skin `lib/` dir.

Standard library

By moving `stuff.*` to `lib/`, we have in fact added support for an arbitrary library module to Shed Skin. Other programs compiled by Shed Skin can now import `stuff` and use `more_primes`. In fact, in the `lib/` directory, you can find type models and implementations for all supported modules. As you may notice, some have been partially converted to C++ using Shed Skin.

Shed Skin types

Shed Skin reimplements the Python builtins with its own set of C++ classes. These have a similar interface to their Python counterparts, so they should be easy to use (provided you have some basic C++ knowledge.) See the class definitions in `lib/builtin.hpp` for details. If in doubt, convert some equivalent Python code to C++, and have a look at the result!

Command-line options

The shedskin command can be given the following options:

- `-a --ann` Output annotated source code (`.ss.py`)
- `-b --nobounds` Disable bounds checking
- `-e --extmod` Generate extension module
- `-f --flags` Provide alternate Makefile flags
- `-g --nogcwarns` Disable runtime GC warnings
- `-l --long` Use long long ("64-bit") integers
- `-m --makefile` Specify alternate Makefile name
- `-n --silent` Silent mode, only show warnings
- `-o --noassert` Disable assert statements
- `-r --random` Use fast random number generator (`rand()`)
- `-s --strhash` Use fast string hashing algorithm (murmur)
- `-w --nowrap` Disable wrap-around checking
- `-x --traceback` Print traceback for uncaught exceptions
- `-L --lib` Add a library directory

For example, to compile the file `test.py` as an extension module, type

```
shedskin -e test
```

or

```
shedskin --extmod test.
```

Using `-b` or `--nobounds` is also very common, as it disables out-of-bounds exceptions (`IndexError`), which can have a large impact on performance.

```
a = [1, 2, 3]
print a[5] # invalid index: out of bounds
```

Performance tips and tricks

Performance tips

- Small memory allocations (e.g. creating a new tuple, list or class instance..) typically do not slow down Python programs by much. However, after compilation to C++, they can quickly become a bottleneck. This is because for each allocation, memory has to be requested from the system, the memory has to be garbage-collected, and

many memory allocations are further likely to cause cache misses. The key to getting very good performance is often to reduce the number of small allocations, for example by rewriting a small list comprehension by a for loop or by avoiding intermediate tuples in some calculation.

- But note that for the idiomatic `for a, b in enumerate(..)`, `for a, b in enumerate(..)` and `for a, b in somedict.iteritems()`, the intermediate small objects are optimized away, and that 1-length strings are cached.
- Several Python features (that may slow down generated code) are not always necessary, and can be turned off. See the section *Command-line options* for details. Turning off bounds checking is usually a very safe optimization, and can help a lot for indexing-heavy code.
- Attribute access is faster in the generated code than indexing. For example, `v.x * v.y * v.z` is faster than `v[0] * v[1] * v[2]`.
- Shed Skin takes the flags it sends to the C++ compiler from the `FLAGS*` files in the Shed Skin installation directory. These flags can be modified, or overruled by creating a local file named `FLAGS`.
- When doing float-heavy calculations, it is not always necessary to follow exact IEEE floating-point specifications. Avoiding this by adding -ffast-math can sometimes greatly improve performance.
- Profile-guided optimization can help to squeeze out even more performance. For a recent version of GCC, first compile and run the generated code with `-fprofile-generate`, then with `-fprofile-use`.
- For best results, configure a recent version of the Boehm GC using `CPPFLAGS="-O3 -march=native" ./configure --enable-cplusplus --enable-threads=pthreads --enable-thread-local-alloc --enable-large-config --enable-parallel-mark`. The last option allows the GC to take advantage of having multiple cores.
- When optimizing, it is extremely useful to know exactly how much time is spent in each part of your program. The program Gprof2Dot can be used to generate beautiful graphs for a stand-alone program, as well as the original Python code. The program OProfile can be used to profile an extension module.

To use Gprof2dot, download `gprof2dot.py` from the website, and install Graphviz. Then:

```
shedskin program
make program_prof
./program_prof
gprof program_prof | gprof2dot.py | dot -Tpng -ooutput.png
```

To use OProfile, install it and use it as follows.

```
shedskin -e extmod
make
sudo opcontrol --start
python main_program_that_imports_extmod
sudo opcontrol --shutdown
opreport -l extmod.so
```

Tricks

- The following two code fragments work the same, but only the second one is supported:

```
statistics = {'nodes': 28, 'solutions': set()}
```

```
class statistics: pass
s = statistics(); s.nodes = 28; s.solutions = set()
```

- The evaluation order of arguments to a function or print changes with translation to C++, so it's better not to depend on this:

```
print 'hoei', raw_input() # raw_input is called before printing 'hoei'!
```

- Tuples with different types of elements and length > 2 are currently not supported. It can however be useful to 'simulate' them:

```
class mytuple:
    def __init__(self, a, b, c):
        self.a, self.b, self.c = a, b, c
```

- Block comments surrounded by #{ and #} are ignored by Shed Skin. This can be used to comment out code that cannot be compiled. For example, the following will only produce a plot when run using CPython:

```
print "x =", x
print "y =", y
#{
import pylab as pl
pl.plot(x, y)
pl.show()
#}
```

How to help out in development

Open source projects thrive on feedback. Please send in bug reports, patches or other code, or suggestions about this document; or join the mailing list and start or participate in discussions. There is also an "easytask" issue label for possible tasks to start out with.

If you are a student, you might want to consider applying for the yearly Google Summer of Code or GHOP projects. Shed Skin has so far successfully participated in one Summer of Code and one GHOP.

CHAPTER 3

Frequently occurring discussions

Frequently occurring discussions

My god, why?

Because we can!

But I heard we can't! Type inference doesn't scale! it's exponential...

Does the graph published here look exponential? Since publishing it, the largest program has almost doubled in size, and it still fits this (slightly quadratic?) curve.

Type inference for arbitrary python code may be intractable, but that doesn't say anything about statically restricted programs. There just hasn't been much research on the topic in recent years, and failed experiments from way back still seem to linger in people's minds. There have been real improvements though, and computers are infinitely faster these days.

The perceived problem is also somewhat academic, because one could observe most types in a running python program (or store analysis results of a previous session), making type inference much easier. shedskin doesn't do this, because so far it's not needed.

But still, a JIT compiler is easier to use, and just as fast.

shedskin makes a different trade-off than the typical JIT compiler. Rather than try and give a good speedup for arbitrary python code, it is a tool that explicitly sacrifices some flexibility in order to maximize performance for certain types of programs. This may make it useless in many cases, and useful in some.

It is very useful to have a good JIT compiler when writing, say, a Django application. But when you are writing a neural network implementation, for example, it is not easy for a JIT to come close to the performance of a static C++ compiler. Here is an interesting performance comparison between different python implementions. Note that the first two tests can run a lot faster still when using 'shedskin -b'.

Static compilation has other advantages. If you know what you are doing, for example, it is more transparent how your code is optimized. Static typing also gives you a guarantee of type-correctness - compilation with shedskin sometimes actually uncovers bugs in the original code.

No sir, I don't like it. Restricted python is not python!

Code accepted by shedskin is still pure python code. You can still debug it with CPython, use most of the python builtins, list comprehensions, and more than 20 common standard library modules. The shedskin example test set shows that a statically restricted subset of python can still be quite useful.

shedskin can also generate extension modules for you, that can be incorporated into larger python programs. So you can use unrestricted python code and arbitrary libraries in your main program, but still get a speedup in some critical piece of code, while keeping everything in pure python.

For example, the pylot raytracer example uses Tk and multiprocessing in combination with a shedskin-compiled extension module, and the c64 emulator example uses pygtk for its interface.

If you want to have ultimate performance, use manual C

True, but:

- not everyone can program in C
- not everyone likes to program in C
- not every program can be made much faster in C
- statically compiled python code can often be fast enough
- a single python version is easier to debug and maintain

Well, almost true, because you will have to use assembly language for ultimate performance of course.

But wait, those JIT compilers will be faster than manual assembly language!

That's of course ridiculous. but it's true JITs will probably be 'fast enough' in more and more cases, so it's just not worth it to spend time optimizing things further. shedskin is there to push performance further if needed, with potentially much less user effort than having to resort to manual C.

Integration is not straightforward

shedskin has its own implementation of the python builtins, so when objects are passed between shedskin and cpython, they often have to be converted, which can be very slow indeed. Additionally, not every type of object can be passed. For example, numpy is currently not supported, so numpy arrays have to be passed via their 'tolist' method.

shedskin is a tool, that can be very useful at times, but often you'll have to puzzle a bit how to best use it in a given situation.

My program doesn't become faster after compilation

In most cases, this is because C++ is not faster at IO, allocating small objects or string operations. there's often no use in compiling a program for speed if one of these is the bottleneck.

Working around small object allocations can often make the resulting C++ much faster. See the *documentation* for other performance tips.

shedskin doesn't terminate for my program

We'd really like to hear about such programs, as they can be very useful to improve shedskin's type inference engine. Please post them on the mailing list.

shedskin is known to have trouble analyzing actually dynamic types. First please make sure that you are not mixing different types together, such as integers and strings in the same list or variable.

If you are not sure where the dynamic types come from, try to compile a small version of your program first, and add to it, until you see which part of the code triggers them.

What about parallellization?

The preferred way to parallelize shedskin-compiled programs is to generate a single extension module (shedskin -e), and use this in combination with the multiprocessing library. This way, the original program will also run faster than when using threads (no issues with the Global Interpreter Lock).

See the pylot raytracer example for an example of this.

What about MSVC?

Occasionally, patches are committed to improve support for MSVC. It is not officially supported, however.

Several modules will probably not work very well, especially the 'os' module.

Patches to improve support for MSVC are welcome.