Nuitka User Manual

Contents

Overview

Nuitka is the Python compiler. It is a good replacement for the Python interpreter and compiles **every** construct that CPython 2.6, 2.7, 3.2 and 3.3 offer. You can use all library or and all extension modules freely. It translates the Python into a C level program that then uses "libpython" to execute in the same way as CPython does, in a very, very compatible way.

This document is the recommended first read if you are interested in using Nuitka, understand its use cases, check what you can expect, license, requirements, credits, etc.

Usage

Requirements

- C++ Compiler: You need a compiler with support for C++03

  Currently this means, you need to use either of these compilers:

  - GNU g++ compiler of at least version 4.4
  - The clang compiler on MacOS X or FreeBSD, based on LLVM version 3.2
  - The MinGW compiler on Windows
  - Visual Studion 2008 or higher on Windows
- Python: Version 2.6, 2.7 or 3.2, 3.3 (support for upcoming 3.4 exists partially)

  You need the standard Python implementation, called CPython, to execute Nuitka, because it is closely tied to using it.

  > ***Note***
  >
  > The created binaries can be made executable independent of the Python installation, with `--standalone` option.

- Operating System: Linux, FreeBSD, NetBSD, MacOS X, and Windows (32/64 bits),

  Others may work as well. The portability is expected to be generally good, but the e.g. Scons usage may have to be adapted.
- Architectures: x86, x86_64 (amd64), and arm.

  Other architectures may also work, as Nuitka is generally not using much hardware specifics. These are just the ones tested and known good. Feedback is welcome. Generally the architectures that Debian supports should be considered good.

Command Line

No environment variable changes are needed, you can call the `nuitka` and `nuitka-run` scripts directly without any changes to the environment. You may want to add the `bin` directory to your `PATH` for your convenience, but that step is optional.

Nuitka has a `--help` option to output what it can do:

```
nuitka --help
```

The `nuitka-run` command is the same as `nuitka`, but with different default. It tries to compile and directly execute a Python script:

```
nuitka-run --help
```

These option that is different is `--run`, and passing on arguments after the first non-option to the created binary, so it is somewhat more similar to what plain `python` will do.

License

Nuitka is licensed under the Apache License, Version 2.0; you may not use it except in compliance with the License.

You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Use Cases

Use Case 1 - Program compilation with all modules embedded

If you want to compile a whole program recursively, and not only the single file that is the main program, do it like this:

```
nuitka --recurse-all program.py
```

> ***Note***
>
> The is more fine grained control than `--recurse-all` available. Consider the output of `nuitka --help`.

In case you have a plugin directory, i.e. one which cannot be found by recursing after normal import statements via the `PYTHONPATH` (which would be recommended way), you can always require that a given directory shall also be included in the executable:

```
nuitka --recurse-all --recurse-directory=plugin_dir program.py
```

> ***Note***
>
> If you don't do any dynamic imports, simply setting your `PYTHONPATH` at compilation time will be sufficient for all your needs normally.

Use `--recurse-directory` only if you make `__import__()` calls that Nuitka cannot predict, because they e.g. depend on command line parameters. Nuitka also warns about these, and point to the option.

Note

The resulting binary still depends on CPython and used C extension modules being installed.

If you want to be able to copy it to another machine, use `--standalone` and copy the created `program.dist` directory and execute the `program.exe` put inside.

Use Case 2 - Extension Module compilation

If you want to compile a single extension module, all you have to do is this:

```
nuitka --module some_module.py
```

The resulting file "some_module.so" can then be used instead of "some_module.py". It's left as an exercise to the reader, what happens if both are present.

Note

The option `--recurse-all` and other variants work as well.

Use Case 3 - Package compilation

If you need to compile a whole package and embedded all modules, that is also feasible, use Nuitka like this:

```
nuitka --module some_package --recurse-directory=some_package
```

Note

The recursion into the package directory needs to be provided manually, otherwise the package is empty. Data files located inside the package will not be embedded yet.

Where to go next

Remember, this project is not completed yet. Although the CPython test suite works near perfect, there is still more work needed, to make it do more optimization. Try it out.

Subscribe to its mailing lists

Please visit the mailing list page in order to subscribe the relatively low volume mailing list. All Nuitka issues can be discussed there.

Report issues or bugs

Should you encounter any issues, bugs, or ideas, please visit the Nuitka bug tracker and report them.

Contact me via email with your questions

You are welcome to contact me via email with your questions.

Word of Warning

Consider using this software with caution. Your feedback and patches to Nuitka are very welcome.

Especially report it please, if you find that anything doesn't work, because the project is now at the stage that this should not happen.

Join Nuitka

You are more than welcome to join Nuitka development and help to complete the project in all minor and major ways.

The development of Nuitka occurs in git. We currently have these 2 branches:

- master:

  This branch contains the stable release to which only hotfixes for bugs will be done. It is supposed to work at all times and is supported.

- develop:

  This branch contains the ongoing development. It may at times contain little regressions, but also new features. On this branch the integration work is done, whereas new features might be developed on feature branches.

- factory:

  This branch contains potentially unfinished and incomplete work. It is very frequently subject `git rebase` and the public staging ground, where my work for develop branch lives first. It is intended for testing only and recommended to base any of your own development on.

> ***Note***
>
> I accept patch files, git formatted patch queues (use `git format-patch origin` command), or if you prefer git pull on the social code platforms.
>
> I will do the integration work. If you base your work on "master" or "develop" at any given time, I will do any re-basing required and keep your authorship intact.

> ***Note***
>
> The Developer Manual explains the coding rules, branching model used, with feature branches and hotfix releases, the Nuitka design and much more. Consider reading it to become a contributor. This document is intended for Nuitka users.

Donations

Should you feel that you cannot help Nuitka directly, but still want to support, please consider making a donation and help this way.

Unsupported functionality

The *co_code* attribute of code objects

The code objects are empty for for native compiled functions. There is no bytecode with Nuitka's compiled function objects, so there is no way to provide it.

Optimization

Constant Folding

The most important form of optimization is the constant folding. This is when an operation can be predicted. Currently Nuitka does these for some built-ins (but not all yet), and it does it for binary/unary operations and comparisons.

Constants currently recognized:

```
5 + 6     # operations
5 < 6     # comparisons
range(3)  # built-ins
```

Literals are the one obvious source of constants, but also most likely other optimization steps like constant propagation or function inlining will be. So this one should not be underestimated and a very important step of successful optimizations. Every option to produce a constant may impact the generated code quality a lot.

Status: The folding of constants is considered implemented, but it might be incomplete. Please report it as a bug when you find an operation in Nuitka that has only constants are input and is not folded.

Constant Propagation

At the core of optimizations there is an attempt to determine values of variables at run time and predictions of assignments. It determines if their inputs are constants or of similar values. An expression, e.g. a module variable access, an expensive operation, may be constant across the module of the function scope and then there needs to be none, or no repeated module variable look-up.

Consider e.g. the module attribute `__name__` which likely is only ever read, so its value could be predicted to a constant string known at compile time. This can then be used as input to the constant folding.

```
if __name__ == "__main__":
   # Your test code might be here
   use_something_not_use_by_program()
```

From modules attributes, only `__name__` is currently actually optimized. Also possible would be at least `__doc__`.

Also built-in exception name references are optimized if they are uses as module level read only variables:

```
try:
   something()
except ValueError: # The ValueError is a slow global name lookup normally.
   pass
```

Builtin Call Prediction

For builtin calls like `type`, `len`, or `range` it is often possible to predict the result at compile time, esp. for constant inputs the resulting value often can be precomputed by Nuitka. It can simply determine the result or the raised exception and replace the builtin call with it allowing for more constant folding or code path folding.

```
type( "string" ) # predictable result, builtin type str.
len( [ 1, 2 ] )  # predictable result
range( 3, 9, 2 ) # predictable result
range( 3, 9, 0 ) # predictable exception, range hates that 0.
```

The builtin call prediction is considered implemented. We can simply during compile time emulate the call and use its result or raised exception. But we may not cover all the built-ins there are yet.

Sometimes the result of a built-in should not be predicted when the result is big. A `range()` call e.g. may give too big values to include the result in the binary. Then it is not done.

```
range( 100000 ) # We do not want this one to be expanded
```

Status: This is considered mostly implemented. Please file bugs for built-ins that are predictable but are not computed by Nuitka at compile time.

Conditional Statement Prediction

For conditional statements, some branches may not ever be taken, because of the conditions being possible to predict. In these cases, the branch not taken and the condition check is removed.

This can typically predict code like this:

```
if __name__ == "__main__":
   # Your test code might be here
   use_something_not_use_by_program()
```

or

```
if False:
   # Your deactivated code might be here
```

It will also benefit from constant propagations, or enable them because once some branches have been removed, other things may become more predictable, so this can trigger other optimization to become possible.

Every branch removed makes optimization more likely. With some code branches removed, access patterns may be more friendly. Imagine e.g. that a function is only called in a removed branch. It may be possible to remove it entirely, and that may have other consequences too.

Status: This is considered implemented, but for the maximum benefit, more constants needs to be determined at compile time.

Exception Propagation

For exceptions that are determined at compile time, there is an expression that will simply do raise the exception. These can be propagated, collecting potentially "side effects", i.e. parts of expressions that must still be executed.

Consider the following code:

```
print side_effect_having() + (1 / 0)
print something_else()
```

The `(1 / 0)` can be predicted to raise a `ZeroDivisionError` exception, which will be propagated through the `+` operation. That part is just Constant Propagation as normal.

The call to `side_effect_having` will have to be retained though, but the print statement, can be turned into an explicit raise. The statement sequence can then be aborted and as such the `something_else` call needs no code generation or consideration anymore.

To that end, Nuitka works with a special node that raises an exception and has so called "side_effects" children, yet can be used in generated code as an expression.

Status: The propagation of exceptions is implemented on a very basic level. It works, but exceptions will not propagate through all different expression and statement types. As work progresses or examples arise, the coverage will be extended.

Exception Scope Reduction

Consider the following code:

```
try:
    b = 8
    print range(3, b, 0)
    print "Will not be executed"
except ValueError, e:
    print e
```

The try block is bigger than it needs to be. The statement `b = 8` cannot cause a `ValueError` to be raised. As such it can be moved to outside the try without any risk.

```
b = 8
try:
    print range(3, b, 0)
    print "Will not be executed"
except ValueError, e:
    print e
```

Status: Not yet done yet. The infrastructure is in place, but until exception block inlining works perfectly, there is not much of a point.

Exception Block Inlining

With the exception propagation it is then possible to transform this code:

```
try:
    b = 8
    print range(3, b, 0)
    print "Will not be executed"
except ValueError, e:
    print e
```

```
try:
    raise ValueError, "range() step argument must not be zero"
except ValueError, e:
    print e
```

Which then can be reduced by avoiding the raise and catch of the exception, making it:

```
e = ValueError( "range() step argument must not be zero" )
print e
```

Status: This is not implemented yet.

Empty Branch Removal

For loops and conditional statements that contain only code without effect, it should be possible to remove the whole construct:

```
for i in range(1000):
    pass
```

The loop could be removed, at maximum it should be considered an assignment of variable `i` to `999` and no more.

Another example:

```
if side_effect_free:
   pass
```

The condition should be removed in this case, as its evaluation is not needed. It may be difficult to predict that `side_effect_free` has no side effects, but many times this might be possible.

Status: This is not implemented yet.

Unpacking Prediction

When the length of the right hand side of an assignment to a sequence can be predicted, the unpacking can be replaced with multiple assignments.

```
a, b, c = 1, side_effect_free(), 3
```

```
a = 1
b = side_effect_free()
c = 3
```

This is of course only really safe if the left hand side cannot raise an exception while building the assignment targets.

We do this now, but only for constants, because we currently have no ability to predict if an expression can raise an exception or not.

Status: Not really implemented, and should use `mayHaveSideEffect()` to be actually good at things.

Builtin Type Inference

When a construct like `in xrange()` or `in range()` is used, it is possible to know what the iteration does and represent that, so that iterator users can use that instead.

I consider that:

```
for i in xrange(1000):
    something(i)
```

could translate `xrange(1000)` into an object of a special class that does the integer looping more efficiently. In case `i` is only assigned from there, this could be a nice case for a dedicated class.

Status: Future work, not even started.

Quicker Function Calls

Functions are structured so that their parameter parsing and `tp_call` interface is separate from the actual function code. This way the call can be optimized away. One problem is that the evaluation order can differ.

```
def f(a, b, c):
    return a, b, c

f( c = get1(), b = get2(), a = get3() )
```

This will evaluate first get1(), then get2() and then get3() and then make the call.

In C++ whatever way the signature is written, its order is fixed.

Therefore it will be necessary to have a staging of the parameters before making the actual call, to avoid an re-ordering of the calls to get1(), get2() and get3().

To solve this, we may have to create wrapper functions that allow different order of parameters to C++.

Status: Not even started.

Lowering of iterated Container Types

In some cases, accesses to `list` constants can become `tuple` constants instead.

Consider that:

```
for x in [ 1, 2, 7 ]:
    something( x )
```

Can be optimized into this:

```
for x in ( 1, 2, 7 ):
     something( x )
```

This allows for simpler code to be generated, and less checks needed, because e.g. the `tuple` is clearly immutable, whereas the `list` needs a check to assert that.

Something similar is possible for `set` and in theory also for `dict`. For the later it will be non-trivial though to maintain the order of execution without temporary values introduced. The same thing is done for pure constants of these types, they change to `tuple` values when iterated.

Status: Implemented, needs other optimization to become generally useful, will help others to become possible.

Credits

Contributors to Nuitka

Thanks go to these individuals for their much valued contributions to Nuitka. Contributors have the license to use Nuitka for their own code even if Closed Source.

The order is sorted by time.

- Li Xuan Ji: Contributed patches for general portability issue and enhancements to the environment variable settings.
- Nicolas Dumazet: Found and fixed reference counting issues, `import` packages work, improved some of the English and generally made good code contributions all over the place, solved code generation TODOs, did tree building cleanups, core stuff.
- Khalid Abu Bakr: Submitted patches for his work to support MinGW and Windows, debugged the issues, and helped me to get cross compile with MinGW from Linux to Windows. This was quite a difficult stuff.
- Liu Zhenhai: Submitted patches for Windows support, making the inline Scons copy actually work on Windows as well. Also reported import related bugs, and generally helped me make the Windows port more usable through his testing and information.
- Christopher Tott: Submitted patches for Windows, and general as well as structural cleanups.
- Pete Hunt: Submitted patches for MacOS X support.
- "ownssh": Submitted patches for built-ins module guarding, and made massive efforts to make high quality bug reports. Also the initial "standalone" mode implementation was created by him.
- Juan Carlos Paco: Submitted cleanup patches, creator of the Nuitka GUI, creator of the Ninja IDE plugin for Nuitka.
- "dr. Equivalent": Submitted the Nuitka Logo.
- Johan Holmberg: Submitted patch for Python3 support on MacOS X.
- Umbra: Submitted patches to make the Windows port more usable, adding user provided application icons, as well as MSVC support for large constants and console applications.

Projects used by Nuitka

- The CPython project

  Thanks for giving us CPython, which is the base of Nuitka. We are nothing without it.

- The GCC project

  Thanks for not only the best compiler suite, but also thanks for supporting C++11 which helped to get Nuitka off the ground. Your compiler was the first usable for Nuitka and with little effort.

- The Scons project

  Thanks for tackling the difficult points and providing a Python environment to make the build results. This is such a perfect fit to Nuitka and a dependency that will likely remain.

- The valgrind project

  Luckily we can use Valgrind to determine if something is an actual improvement without the noise. And it's also helpful to determine what's actually happening when comparing.

- The NeuroDebian project

  Thanks for hosting the build infrastructure that the Debian and sponsor Yaroslav Halchenko uses to provide packages for all Ubuntu versions.

- The openSUSE Buildservice

  Thanks for hosting this excellent service that allows us to provide RPMs for a large variety of platforms and make them available immediately nearly at release time.

- The MinGW project

  Thanks for porting the gcc to Windows. This allowed portability of Nuitka with relatively little effort. Unfortunately this is currently limited to compiling CPython with 32 bits, and 64 bits requires MSVC compiler.

- The Buildbot project

  Thanks for creating an easy to deploy and use continous integration framework that also runs on Windows and written and configured in Python. This allows to run the Nuitka tests long before release time.

Updates for this Manual

This document is written in REST. That is an ASCII format which is readable as ASCII, but used to generate PDF or HTML documents.

You will find the current source under: http://nuitka.net/gitweb/?p=Nuitka.git;a=blob_plain;f=README.rst

And the current PDF under: http://nuitka.net/doc/README.pdf