Nuitka User Manual

Overview

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.

Nuitka is **the** Python compiler. It is written in Python. It is a seamless replacement or extension to the Python interpreter and compiles **every** construct that CPython 2.6, 2.7, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9 have, when itself run with that Python version.

It then executes uncompiled code and compiled code together in an extremely compatible manner.

You can use all Python library modules and all extension modules freely.

Nuitka translates the Python modules into a C level program that then uses `libpython` and static C files of its own to execute in the same way as CPython does.

All optimization is aimed at avoiding overhead, where it's unnecessary. None is aimed at removing compatibility, although slight improvements will occasionally be done, where not every bug of standard Python is emulated, e.g. more complete error messages are given, but there is a full compatibility mode to disable even that.

Usage

Requirements

- C Compiler: You need a compiler with support for C11 or alternatively for C++03 [1]

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

  - The `gcc` compiler of at least version 5.1, or the g++ compiler of at least version 4.4 as an alternative.
  - The `clang` compiler on macOS X or FreeBSD.
  - The MinGW64 C11 compiler on Windows, must be based on gcc 8 or higher. It will be automatically downloaded if not found, which is the recommended way of installing it.
  - Visual Studio 2019 or higher on Windows [2], older versions will work but only supported for commercial users. Configure to use the English language pack for best results (Nuitka filters away garbage outputs, but only for that language).
  - On Windows the `clang-cl` compiler on Windows can be used if provided by the Visual Studion installer.
- Python: Version 2.6, 2.7 or 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9

> ***For Python 3.3, and 3.4 and only those versions, we need other Python versions as a compile time dependency.***
>
> Nuitka itself is fully compatible with all listed versions, but Scons as an internally used tool is not.

For these versions, you *need* a Python2 or Python 3.5 or higher installed as well, but only during the compile time only. That is for use with Scons (which orchestrates the C compilation), which does not support the same Python versions as Nuitka.

In addition, on Windows, Python2 cannot be used because `clcache` does not work with it, there a Python 3.5 or higher needs to be installed.

Nuitka finds these needed Python versions (on Windows via registry) and you shouldn't notice it as long as they are installed.

Moving binaries to other machines

The created binaries can be made executable independent of the Python installation, with `--standalone` and `--onefile` options.

Binary filename suffix

The created binaries have an `.exe` suffix on Windows. On other platforms they have no suffix for standalone mode, or `.bin` suffix, that you are free to remove or change, or specify with the `-o` option.

The suffix for acceleration mode is added just to be sure that the original script name and the binary name do not ever collide, so we can safely do an overwrite without destroying the original source file.

It has to be CPython, Anaconda Python.

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

On Windows, for Python not installed system-wide and acceleration mode, you need to copy the `PythonXX.DLL` alongside it, something Nuitka does automatically.

It cannot be from Windows app store

It is known that Windows app store Python definitely does not work, it's checked against. And on macOS "pyenv" likely does **not** work.

- 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. Make sure to match Windows Python and C compiler architecture, or else you will get cryptic error messages.

- Architectures: x86, x86_64 (amd64), and arm, likely many more

  Other architectures are expected to also work, out of the box, as Nuitka is generally not using any hardware specifics. These are just the ones tested and known to be good. Feedback is welcome. Generally, the architectures that Debian supports can be considered good and tested too.

Command Line

The recommended way of executing Nuitka is `<the_right_python> -m nuitka` to be absolutely certain which Python interpreter you are using, so it is easier to match with what Nuitka has.

The next best way of executing Nuitka bare that is from a source checkout or archive, with no environment variable changes, most noteworthy, you do not have to mess with `PYTHONPATH` at all for Nuitka. You just execute 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.

Moreover, if you want to execute with the right interpreter, in that case, be sure to execute `<the_right_python> bin/nuitka` and be good.

> ***Pick the right Interpreter***
>
> If you encounter a `SyntaxError` you absolutely most certainly have picked the wrong interpreter for the program you are compiling.

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

```
nuitka --help
```

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

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

This 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.

Installation

For most systems, there will be packages on the download page of Nuitka. But you can also install it from source code as described above, but also like any other Python program it can be installed via the normal `python setup.py install` routine.

License

Tutorial Setup and build on Windows

This is basic steps if you have nothing installed, of course if you have any of the parts, just skip it.

Setup

Install Python

- Download and install from https://www.python.org/downloads/windows
- Select one of `Windows x86-64 web-based installer` (64 bits Python, recommended) or `x86 executable` (32 bits Python) installer.
- Verify using command `python --version`.

Install Nuitka

- `python -m pip install nuitka`
- Verify using command `python -m nuitka --version`

Write some code and test

Create a folder for the Python code

- mkdir HelloWorld
- make a python file named **hello.py**

```
def talk(message):
    return "Talk " + message

def main():
    print( talk("Hello World"))

if __name__ == "__main__":
    main()
```

Test your program

Do as you normally would. Running Nuitka on code that works incorrectly is not easier to debug.

```
python hello.py
```

Build it using

```
python -m nuitka --mingw64 hello.py
```

> ***Note***
>
> This will prompt you to download a C caching tool (to speed up repeated compilation of generated C code) and a MinGW64 based C compiler. Say yes to those.

If you like to have full output add `--show-progress` and `--show-scons`.

Run it

Execute the `hello.exe` created near `hello.py`.

Distribute

To distribute, build with `--standalone` option, which will not output a single executable, but a whole folder. Copy the resulting `hello.dist` folder to the other machine and run it.

You may also try `--onefile` which does create a single file, but make sure that the mere standalone is working, before turning to it, as it will make the debugging only harder, e.g. in case of missing data files.

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:

```
python -m nuitka --follow-imports program.py
```

> ***Note***
>
> There are more fine grained controls than `--follow-imports` available. Consider the output of `nuitka --help`. Including less modules into the compilation, but instead using normal Python for it will make it faster to compile.

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

```
python -m nuitka --follow-imports --include-plugin-directory=plugin_dir program.py
```

> ***Note***
>
> If you don't do any dynamic imports, simply setting your `PYTHONPATH` at compilation time is what you should do.

Use `--include-plugin-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 filename will be `program.exe` on Windows, `program.bin` on other platforms.

Note

The resulting binary still depend 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` (Windows) or `program` (other platforms) put inside.

Use Case 2 - Extension Module compilation

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

```
python -m nuitka --module some_module.py
```

The resulting file `some_module.so` can then be used instead of `some_module.py`.

Note

It's left as an exercise to the reader, to find out what happens if both are present.

Note

The option `--follow-imports` and other variants work as well, but the included modules will only become importable *after* you imported the `some_module` name.

Note

The resulting extension module can only be loaded into a CPython of the same version and doesn't include other extension modules.

Use Case 3 - Package compilation

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

```
python -m nuitka --module some_package --include-package=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.

Use Case 4 - Program Distribution

For distribution to other systems, there is the standalone mode which produces a folder for which you can specify `--standalone`.

```
python -m nuitka --standalone program.py
```

Follow all imports is default in this mode. You can selectively exclude modules by specifically saying `--nofollow-import-to`, but then an `ImportError` will be raised when import of it is attempted at program runtime.

For data files to be included, use the option `--include-data-file=<source>=<target>` where the source is a file system path, but target has to be specified relative. For standalone you can also copy them manually, but this can do extra checks, and for onefile mode, there is no manual copying possible.

For package data, there is a better way, using `--include-package-data` which detects data files of packages automatically and copies them over. It even accepts patterns in shell style.

With data files, you are largely on your own. Nuitka keeps track of ones that are needed by popular packages, but it might be incomplete. Raise issues if you encounter something in these.

When that is working, you can use the onefile if you so desire.

```
python -m nuitka --onefile program.py
```

This will create a single binary, which on Linux will not even unpack itself, but instead loop back mount its contents as a filesystem and use that. On Windows, there are two modes, one which is copying it to the AppData of your company specified, to also use it as a cache, and one which does it in the temporary directory.

Typical Problems

Dynamic `sys.path`

If your script modifies `sys.path` to e.g. insert directories with source code relative to it, Nuitka will currently not be able to see those. However, if you set the `PYTHONPATH` to the resulting value, you will be able to compile it.

Missing data files in standalone

If your program fails to file data, it can cause all kinds of different behaviours, e.g. a package might complain it is not the right version, because a `VERSION` file check defaulted to unknown. The absence of icon files or help texts, may raise strange errors.

Often the error paths for files not being present are even buggy and will reveal programming errors like unbound local variables. Please look carefully at these exceptions keeping in mind that this can be the cause. If you program works without standalone, chances are data files might be cause.

Missing DLLs in standalone

Nuitka has plugins that deal with copying DLLs. For NumPy, SciPy, Tkinter, etc.

These need special treatment to be able to run on other systems. Manually copying them is not enough and will given strange errors. Sometimes newer version of packages, esp. NumPy can be unsupported. In this case you will have to raise an issue, and use the older one.

Dependency creep in standalone

Some packages are a single import, but to Nuitka mean that more than a thousand packages (literally) are to be included. The prime example of Pandas, which does want to plug and use just about everything you can imagine. Multiple frameworks for syntax highlighting everything imaginable take time.

Nuitka will have to learn effective caching to deal with this in the future. Right now, you will have to deal with huge compilation times for these.

Tips

Python command line flags

For passing things like `-O` or `-S` to Python, to your compiled program, there is a command line option name `--python-flag=` which makes Nuitka emulate these options.

The most important ones are supported, more can certainly be added.

Caching compilation results

The C compiler, when invoked with the same input files, will take a long time and much CPU to compile over and over. Make sure you are having `ccache` installed and configured when using gcc (even on Windows). It will make repeated compilations much faster, even if things are not yet not perfect, i.e. changes to the program can cause many C files to change, requiring a new compilation instead of using the cached result.

On Windows, with gcc Nuitka supports using `ccache.exe` which it will offer to download from an official source and it automatically. This is the recommended way of using it on Windows, as other versions can e.g. hang.

Nuitka will pick up `ccache` if it's in found in system `PATH`, and it will also be possible to provide if by setting `NUITKA_CCACHE_BINARY` to the full path of the binary, this is for use in CI systems.

For the Visual Studio compilers, you are just one `pip install clcache` command away. To make Nuitka use those, set `NUITKA_CLCACHE_BINARY` to the full path of `clcache.exe`, which will be in the scripts folder of the Python, you installed it into.

Runners

Avoid running the `nuitka` binary, doing `python -m nuitka` will make a 100% sure you are using what you think you are. Using the wrong Python will make it give you `SyntaxError` for good code or `ImportError` for installed modules. That is happening, when you run Nuitka with Python2 on Python3 code and vice versa. By explicitly calling the same Python interpreter binary, you avoid that issue entirely.

Fastest C Compilers

The fastest binaries of `pystone.exe` on Windows with 64 bits Python proved to be significantly faster with MinGW64, roughly 20% better score. So it is recommended for use over MSVC. Using `clang-cl.exe` of Clang7 was faster than MSVC, but still significantly slower than MinGW64, and it will be harder to use, so it is not recommended.

On Linux for `pystone.bin` the binary produced by `clang6` was faster than `gcc-6.3`, but not by a significant margin. Since gcc is more often already installed, that is recommended to use for now.

Differences in C compilation times have not yet been examined.

Unexpected Slowdowns

Using the Python DLL, like standard CPython does can lead to unexpected slowdowns, e.g. in uncompiled code that works with Unicode strings. This is because calling to the DLL rather than residing in the DLL causes overhead, and this even happens to the DLL with itself, being slower, than a Python all contained in one binary.

So if feasible, aim at static linking, which is currently only possible with Anaconda Python on non-Windows.

Windows Standalone executables and dependencies

The process of making standalone executables for Windows traditionally involves using an external dependency walker in order to copy necessary libraries along with the compiled executables to the distribution folder.

Using the external dependency walker is quite a time consuming, and may copy some unnecessary libraries along the way (better have too much than missing).

There's also an experimental alternative internal dependency walker that relies on pefile which analyses PE imports of executables and libraries.

This implementation shall create smaller Standalone distributions since it won't include Windows' equivalent of the standard library, and will speed-up first Nuitka compilations by an order of magnitude.

In order to use it, you may enable the internal dependency walker by using the following switch:

```
python -m nuitka --standalone --windows-dependency-tool=pefile myprogram.py
```

> ***Note***
>
> The pefile dependency walker will test all dependencies of the distribution folder.
>
> Optionally, it is also possible to check all recursive dependencies of included libraries using the following switch along with the above one:

```
python -m nuitka --standalone --windows-dependency-tool=pefile --experimental=use_pefile
```

Note

Some modules may have hidden dependencies outside of their directory. In order for the pefile dependency walker to find them, you may also scan the whole site-packages directory for missing dependencies using the following switch along with the two above:

```
python -m nuitka --standalone --windows-dependency-tool=pefile --experimental=use_pefile
```

Note

Be aware that using this switch will increase compilation time a lot.

Windows errors with resources

On Windows, the Windows Defender tool and the Windows Indexing Service both scan the freshly created binaries, while Nuitka wants to work with it, e.g. adding more resources, and then preventing operations randomly due to holding locks. Make sure to exclude your compilation stage from these services.

Windows standalone program redistribuation

Whether compiling with MingW or MSVC, the standalone programs have external dependencies to Visual C Runtime libraries. Nuitka tries to ship those dependent DLLs by copying them from your system.

Beginning with Microsoft Windows 10, Microsoft ships *ucrt.dll* (Universal C Runtime libraries) which rehook calls to *api-ms-crt-*.dll*.

With earlier Windows platforms (and wine/ReactOS), you should consider installing Visual C Runtime libraries before executing a Nuitka standalone compiled program.

Depending on the used C compiler, you'll need the following redist versions:

Visual C version	Redist Year	CPython
14.2	2019	3.5, 3.6, 3.7, 3.8, 3.9
14.1	2017	3.5, 3.6, 3.7, 3.8
14.0	2015	3.5, 3.6, 3.7, 3.8
10.0	2010	3.3, 3.4
9.0	2008	2.6, 2.7, 3.0, 3.1, 3.2

When using MingGW64, you'll need the following redist versions:

MingGW64 version	Redist Year	CPython
8.1.0	2015	3.5, 3.6, 3.7, 3.8, 3.9

Once the corresponding runtime libraries are installed on the target system, you may remove all *api-ms-crt-*.dll* files from your Nuitka compiled dist folder.

Detecting Nuitka at run time

It doesn't set `sys.frozen` unlike other tools. For Nuitka, we have the module attribute `__compiled__` to test if a specific module was compiled.

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, esp. to make it do more optimization. Try it out.

Follow me on Twitter

Nuitka announcements and interesting stuff is pointed to on the Twitter account, but obviously with no details. @KayHayen.

I will not answer Nuitka issues via Twitter though, rather make occasional polls, and give important announcements, as well as low-level posts about development ongoing.

Report issues or bugs

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

Best practices for reporting bugs:

- Please always include the following information in your report, for the underlying Python version. You can easily copy&paste this into your report.

```
python -m nuitka --version
```

- Try to make your example minimal. That is, try to remove code that does not contribute to the issue as much as possible. Ideally come up with a small reproducing program that illustrates the issue, using `print` with different results when that programs runs compiled or native.
- If the problem occurs spuriously (i.e. not each time), try to set the environment variable `PYTHONHASHSEED` to `0`, disabling hash randomization. If that makes the problem go away, try increasing in steps of `1` to a hash seed value that makes it happen every time, include it in your report.
- Do not include the created code in your report. Given proper input, it's redundant, and it's not likely that I will look at it without the ability to change the Python or Nuitka source and re-run it.
- Do not send screenshots of text, that is bad and lazy. Instead, capture text outputs from the console.

Word of Warning

Consider using this software with caution. Even though many tests are applied before releases, things are potentially breaking. Your feedback and patches to Nuitka are very welcome.

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 3 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 unfinished and incomplete work. It is very frequently subject to `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. When updating it, you very often will get merge conflicts. Simply resolve those by doing `git reset --hard origin/factory` and switch to the latest version.

> ***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 native compiled functions. There is no bytecode with Nuitka's compiled function objects, so there is no way to provide it.

PDB

There is no tracing of compiled functions to attach a debugger to.

Optimization

Constant Folding

The most important form of optimization is the constant folding. This is when an operation can be fully predicted at compile time. Currently, Nuitka does these for some built-ins (but not all yet, somebody to look at this more closely will be very welcome!), and it does it e.g. for binary/unary operations and comparisons.

Constants currently recognized:

```
5 + 6     # binary operations
not 7     # unary 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 in that not all possible cases are caught. Please report it as a bug when you find an operation in Nuitka that has only constants as input and is not folded.

Constant Propagation

At the core of optimizations, there is an attempt to determine the 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()
```

Status

From modules attributes, only `__name__` is currently actually optimized. Also possible would be at least `__doc__`. In the future, this may improve as SSA is expanded to module variables.

Built-in Name Lookups

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

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

Status

This works for all built-in names. When an assignment is done to such a name, or it's even local, then, of course, it is not done.

Built-in Call Prediction

For built-in 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 built-in call with that value, allowing for more constant folding or code path reduction.

```
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 raises due to 0.
```

Status

The built-in 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 pre-computed, but should not be computed by Nuitka at compile time with specific values.

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
   use_something_not_use_by_program()
```

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 need 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 upwards, collecting potentially "side effects", i.e. parts of expressions that were executed before it occurred, and still have to 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 `side_effect_having()` will have to be retained though, but the `print` does not and 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 is wrapped with a so-called "side_effects" expression, but yet can be used in the code as an expression having a value.

> ***Status***
>
> The propagation of exceptions is mostly implemented but needs handling in every kind of operations, and not all of them might do it already. As work progresses or examples arise, the coverage will be extended. Feel free to generate bug reports with non-working examples.

Exception Scope Reduction

Consider the following code:

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

Status

This is considered done. For every kind of operation, we trace if it may raise an exception. We do however *not* track properly yet, what can do a `ValueError` and what cannot.

Exception Block Inlining

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

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

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

Which then can be lowered in complexity 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.

> ***Status***
>
> This is not implemented yet, as it requires us to track iterators, and their side effects, as well as loop values, and exit conditions. Too much yet, but we will get there.

Another example:

```
if side_effect_free:
   pass
```

The condition check 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 considered implemented. The conditional statement nature is removed if both branches are empty, only the condition is evaluated and checked for truth (in cases that could raise an exception).

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 implemented yet. Will need us to see through the unpacking of what is an iteration over a tuple, we created ourselves. We are not there yet, but we will get there.

Built-in 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 have to evaluate first `get1()`, then `get2()` and only then `get3()` and then make the function call with these values.

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

> ***Status***
>
> Not even started. A re-formulation that avoids the dictionary to call the function, and instead uses temporary variables appears to be relatively straight forward once we do that kind of parameter analysis.

Lowering of iterated Container Types

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

Consider that:

```
for x in [a, b, c]:
    something(x)
```

Can be optimized into this:

```
for x in (a, b, c):
     something(x)
```

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

Status

Implemented, even works for non-constants. Needs other optimization to become generally useful, and will itself help other optimization to become possible. This allows us to e.g. only treat iteration over tuples, and not care about sets.

In theory, something similar is also possible 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.

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 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.
- David Cortesi: Submitted patches and test cases to make macOS port more usable, specifically for the Python3 standalone support of Qt.

- Andrew Leech: Submitted github pull request to allow using "-m nuitka" to call the compiler. Also pull request to improve "bist_nuitka" and to do the registration.
- Pawe■ K: Submitted github pull request to remove glibc from standalone distribution, saving size and improving robustness considering the various distributions.
- Orsiris de Jong: Submitted github pull request to implement the dependency walking with *pefile* under Windows.
- Jorj X. McKie: Submitted github pull requests with NumPy plugin to retain its accelerating libraries, and Tkinter to include the TCL distribution on Windows.

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 making it easy supporting to get Nuitka off the ground. Your compiler was the first usable for Nuitka and with very 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 MinGW64 project

  Thanks for porting the gcc to Windows. This allowed portability of Nuitka with relatively little effort.

- The Buildbot project

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

- The isort project

  Thanks for making nice import ordering so easy. This makes it so easy to let your IDE do it and clean up afterward.

- The black project

  Thanks for making a fast and reliable way for automatically formatting the Nuitka source code.

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: https://nuitka.net/gitweb/?p=Nuitka.git;a=blob_plain;f=README.rst

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

---

1 Support for this C11 is a given with gcc 5.x or higher or any clang version.

The MSVC compiler doesn't do it yet. But as a workaround, as the C++03 language standard is very overlapping with C11, it is then used instead where the C compiler is too old. Nuitka used to require a C++ compiler in the past, but it changed.

2 Download for free from http://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx (the Express editions work just fine).

The latest version is recommended if not required. There is no need to use older versions, they might in fact not work.