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, 3.10 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 MinGW64 C11 compiler on Windows, must be based on gcc 11.2 or higher. It will be *automatically* downloaded if no usable C compiler is found, which is the recommended way of installing it, as Nuitka will also upgrade it for you.
  - Visual Studio 2022 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 English language). It will be used by default if installed.
  - On all other platforms, the `gcc` compiler of at least version 5.1, and below that the `g++` compiler of at least version 4.4 as an alternative.
  - The `clang` compiler on macOS X and most FreeBSD architectures.
  - On Windows the `clang-cl` compiler on Windows can be used if provided by the Visual Studio installer.
- Python: Version 2.6, 2.7 or 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 3.10

> ***Important***
>
> For Python 3.3/3.4 and *only* those, we need other Python version 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 (e.g. on Windows via registry) and you shouldn't notice it as long as they are installed.

Increasingly, other functionality is available when another Python has a certain package installed. For example, onefile compression will work for a Python 2.x when another Python is found that has the `zstandard` package 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.

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 Python 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 it's working 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 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 unless you have a suitable MSVC installed. Say `yes` to both those questions.

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-import-to` and work as well, but the included modules will only become importable *after* you imported the `some_module` name. If these kinds of imports are invisible to Nuitka, e.g. dynamically created, you can use `--include-module` or `--include-package` in that case, but for static imports it should not be needed.

-- note:

```
An extension module can never include other extension modules. You will have to create a wheel for this to be doable.
```

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 inclusion of the package contents needs to be provided manually, otherwise, the package is mostly empty. You can be more specific if you want, and only include part of it, or exclude part of it, e.g. with `--nofollow-import-to='*.tests'` you would not include the unused test part of your code.

> ***Note***
>
> Data files located inside the package will not be embedded by this process, you need to copy them yourself with this approach. Alternatively you can use the file embedding of Nuitka commercial.

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
```

Following 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 run time. This may cause different behavior, but it may also improve your compile time if done wisely.

For data files to be included, use the option `--include-data-files=<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.

To copy some or all file in a directory, use the option `--include-data-files=/etc/*.txt=etc/` where you get to specify shell patterns for the files, and a subdirectory where to put them, indicated by the trailing slash.

To copy a whole folder with all files, you can use `--include-data-dir=/path/to/images=images` which will copy all files including a potential subdirectory structure. You cannot filter here, i.e. if you want only a partial copy, remove the files beforehand.

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. It spares you the need to find the package directory yourself and should be preferred whenever available.

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 mode if you so desire.

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

This will create a single binary, that extracts itself on the target, before running the program. But notice, that accessing files relative to your program is impacted, make sure to read the section Onefile: Finding files as well.

```
# Create a binary that unpacks into a temporary folder
python -m nuitka --onefile program.py
```

> ***Note***
>
> There are more platform specific options, e.g. related to icons, splash screen, and version information, consider the `--help` output for the details of these and check the section *Tweaks_*.

For the unpacking, by default a unique user temporary path one is used, and then deleted, however this default `--onefile-tempdir-spec="%TEMP%/onefile_%PID%_%TIME%"` can be overridden with a path specification that is using then using a cached path, avoiding repeated unpacking, e.g. with `--onefile-tempdir-spec="%CACHE_DIR%/%COMPANY%/%PRODUCT%/%VERSION"` which uses version information, and user specific cache directory.

> ***Note***
>
> Using cached paths will e.g. be relevant too, when Windows Firewall comes into play, because otherwise, the binary will be a different one to it each time it is run.

Currently these expanded tokens are available:

Token	What this Expands to	Example
%TEMP%	User temporary file directory	C:Users...AppDataLocalsTemp
%PID%	Process ID	2772
%TIME%	Time in seconds since the epoch.	1299852985
%PROGRAM%	Full program run-time filename of executable.	C:SomeWhereYourOnefile.exe
%CACHE_DIR%	Cache directory for the user.	C:UsersSomeBodyAppDataLocal
%COMPANY%	Value given as `--company-name`	YourCompanyName
%PRODUCT%	Value given as `--product-name`	YourProductName
%VERSION%	Combination of `--file-version` & `--product-version`	3.0.0.0-1.0.0.0
%HOME%	Home directory for the user.	/home/somebody

Note

It is your responsibility to make the path provided unique, on Windows a running program will be locked, and while using a fixed folder name is possible, it can cause locking issues in that case, where the program gets restarted.

Usually you need to use `%TIME%` or at least `%PID%` to make a path unique, and this is mainly intended for use cases, where e.g. you want things to reside in a place you choose or abide your naming conventions.

Use Case 5 - Setuptools Wheels

If you have a `setup.py`, `setup.cfg` or `pyproject.toml` driven creation of wheels for your software in place, putting Nuitka to use is extremely easy.

Lets start with the most common `setuptools` approach, you can - having Nuitka installed of course, simply execute the target `bdist_nuitka` rather than the `bdist_wheel`. It takes all the options and allows you to specify some more, that are specific to Nuitka.

```
# For setup.py if not you't use other build systems:
setup(
   ...,
   command_options={
      'nuitka': {
         # boolean option, e.g. if you cared for C compilation commands
         '--show-scons': True,
         # options without value, e.g. enforce using Clang
         '--clang': None,
         # options with single values, e.g. enable a plugin of Nuitka
         '--enable-plugin': "pyside2",
         # options with several values, e.g. avoiding including modules
         '--nofollow-import-to' : ["*.tests", "*.distutils"],
      }
   },
)

# For setup.py with other build systems:
# The tuple nature of the arguments is required by the dark nature of
# "setuptools" and plugins to it, that insist on full compatibility,
# e.g. "setuptools_rust"

setup(
   ...,
   command_options={
      'nuitka': {
         # boolean option, e.g. if you cared for C compilation commands
         '--show-scons': ("setup.py", True),
         # options without value, e.g. enforce using Clang
         '--clang': ("setup.py", None),
         # options with single values, e.g. enable a plugin of Nuitka
         '--enable-plugin': ("setup.py", "pyside2"),
         # options with several values, e.g. avoiding including modules
         '--nofollow-import-to' : ("setup.py", ["*.tests", "*.distutils"]),
      }
```

```
   },
)
```

If for some reason, you cannot or do not what to change the target, you can add this to your `setup.py`.

```
# For setup.py
setup(
   ...,
   build_with_nuitka=True
)
```

> ***Note***
>
> To temporarily disable the compilation, you could remove above line, or edit the value to `False` by or take its value from an environment variable if you so choose, e.g. `bool(os.environ.get("USE_NUITKA", "True"))`. This is up to you.

Or you could put it in your `setup.cfg`

```
[metadata]
build_with_nuitka = True
```

And last, but not least, Nuitka also supports the new `build` meta, so when you have a `pyproject.toml` already, simple replace or add this value:

```
[build-system]
requires = ["setuptools>=42", "wheel", "nuitka", "toml"]
build-backend = "nuitka.distutils.Build"

[nuitka]
# These are not recommended, but they make it obvious to have effect.

# boolean option, e.g. if you cared for C compilation commands, leading
# dashes are omitted
show-scons = true

# options with single values, e.g. enable a plugin of Nuitka
enable-plugin = pyside2

# options with several values, e.g. avoiding including modules, accepts
# list argument.
nofollow-import-to = ["*.tests", "*.distutils"]
```

> ***Note***
>
> For the `nuitka` requirement above absolute paths like `C:\Users...\Nuitka` will also work on Linux, use an absolute path with *two* leading slashes, e.g. `//home/.../Nuitka`.

Tweaks

Icons

For good looks, you may specify icons. On Windows, you can provide an icon file, a template executable, or a PNG file. All of these will work and may even be combined:

```
# These create binaries with icons on Windows
python -m nuitka --onefile --windows-icon-from-ico=your-icon.png program.py
python -m nuitka --onefile --windows-icon-from-ico=your-icon.ico program.py
python -m nuitka --onefile --windows-icon-template-exe=your-icon.ico program.py

# These create application bundles with icons on macOS
python -m nuitka --macos-create-app-bundle --macos-app-icon=your-icon.png program.py
python -m nuitka --macos-create-app-bundle --macos-app-icon=your-icon.icns program.py
```

> ***Note***
>
> With Nuitka, you do not have to create platform specific icons, but instead it will convert e.g. PNG, but also other format on the fly during the build.

MacOS Entitlements

Entitlements for an macOS application bundle can be added with the option, `--macos-app-protected-resource`, all values are listed on this page from Apple

An example value would be `--macos-app-protected-resource=NSMicrophoneUsageDescription:Microphone access` for requesting access to a Microphone. After the colon, the descriptive text is to be given.

> ***Note***
>
> Beware that in the likely case of using spaces in the description part, you need to quote it for your shell to get through to Nuitka and not be interpreted as Nuitka arguments.

Console Window

On Windows, the console is opened by programs unless you say so. Nuitka defaults to this, effectively being only good for terminal programs, or programs where the output is requested to be seen. There is a difference in `pythonw.exe` and `python.exe` along those lines. This is replicated in Nuitka with the option `--disable-console`. Nuitka recommends you to consider this in case you are using `PySide6` e.g. and other GUI packages, e.g. `wx`, but it leaves the decision up to you. In case, you know your program is console application, just using `--enable-console` which will get rid of these kinds of outputs from Nuitka.

> ***Note***
>
> The `pythonw.exe` is never good to be used with Nuitka, as you cannot see its output.

Splash screen

Splash screens are useful when program startup is slow. Onefile startup itself is not slow, but your program may be, and you cannot really know how fast the computer used will be, so it might be a good idea to have them. Luckily with Nuitka, they are easy to add for Windows.

For splash screen, you need to specify it as an PNG file, and then make sure to disable the splash screen when your program is ready, e.g. has complete the imports, prepared the window, connected to the database, and wants the splash screen to go away. Here we are using the project syntax to combine the code with the creation, compile this:

```
# nuitka-project: --onefile
# nuitka-project: --onefile-windows-splash-screen-image={MAIN_DIRECTORY}/Splash-Screen.png

# Whatever this is obviously
print("Delaying startup by 10s...")
import time
time.sleep(10)

# Use this code to signal the splash screen removal.
if "NUITKA_ONEFILE_PARENT" in os.environ:
   splash_filename = os.path.join(
      tempfile.gettempdir(),
      "onefile_%d_splash_feedback.tmp" % int(os.environ["NUITKA_ONEFILE_PARENT"]),
   )

   if os.path.exists(splash_filename):
      os.unlink(splash_filename)

print("Done... splash should be gone.")
...

# Rest of your program goes here.
```

Typical Problems

Memory issues and compiler bugs

Sometimes the C compilers will crash saying they cannot allocate memory or that some input was truncated, or similar error messages, clearly from it. There are several options you can explore here:

Ask Nuitka to use less memory

There is a dedicated option `--low-memory` which influences decisions of Nuitka, such that it avoids high usage of memory during compilation at the cost of increased compile time.

Avoid 32 bit C compiler/assembler memory limits

Do not use a 32 bits compiler, but a 64 bit one. If you are using Python with 32 bits on Windows, you most definitely ought to use MSVC as the C compiler, and not MinGW64. The MSVC is a cross compiler, and can use more memory than gcc on that platform. If you are not on Windows, that is not an option of course. Also using the 64 bits Python will work.

Use a minimal virtualenv

When you compile from a living installation, that may well have many optional dependencies of your software installed. Some software, will then have imports on these, and Nuitka will compile them as well. Not only may these be just the trouble makers, they also require more memory, so get rid of that. Of course you do have to check that your program has all needed dependencies before you attempt to compile, or else the compiled program will equally not run.

Use LTO compilation or not

With `--lto=yes` or `--lto=no` you can switch the C compilation to only produce bytecode, and not assembler code and machine code directly, but make a whole program optimization at the end. This will change the memory usage pretty dramatically, and if you error is coming from the assembler, using LTO will most definitely avoid that.

Switch the C compiler to clang

People have reported that programs that fail to compile with gcc due to its bugs or memory usage work fine with clang on Linux. On Windows, this could still be an option, but it needs to be implemented first for the automatic downloaded gcc, that would contain it. Since MSVC is known to be more memory effective anyway, you should go there, and if you want to use Clang, there is support for the one contained in MSVC.

Add a larger swap file to your embedded Linux

On systems with not enough RAM, you need to use swap space. Running out of it is possibly a cause, and adding more swap space, or one at all, might solve the issue, but beware that it will make things extremely slow when the compilers swap back and forth, so consider the next tip first or on top of it.

Limit the amount of compilation jobs

With the `--jobs` option of Nuitka, it will not start many C compiler instances at once, each competing for the scarce resource of RAM. By picking a value of one, only one C compiler instance will be running, and on a 8 core system, that reduces the amount of memory by factor 8, so that's a natural choice right there.

Dynamic `sys.path`

If your script modifies `sys.path` to e.g. insert directories with source code relative to it, Nuitka will not be able to see those. However, if you set the `PYTHONPATH` to the resulting value, it will be able to compile it and find the used modules from these paths as well.

Manual Python File Loading

A very frequent pattern with private code is that it scans plugin directories of some kind, and uses `os.listdir`, checks filenames, and then opens a file and does `exec` on them. This approach is working for Python code, but for compiled code, you should use this much cleaner approach, that works for pure Python code and is a lot less vulnerable.

```
# Using a package name, to locate the plugins, but this can actually
# be also a directory.
scan_path = scan_package.__path__

for item in pkgutil.iter_modules(scan_path):
   # You may want to do it recursively, but we don't do this here in
   # this example.
   if item.ispkg:
      continue
```

```
   # The loader object knows how to do it.
   module_loader = item.module_finder.find_module(item.name)

   # Ignore bytecode only left overs. Deleted files can cause
   # these things, so we just ignore it. Not every load has a
   # filename, so we need to catch that error.
   try:
      if module_loader.get_filename().endswith(".pyc"):
         continue
   except AttributeError:
      # Not a bytecode loader, but e.g. extension module, which is OK in case
      # it was compiled with Nuitka.
      pass

   plugin_module = module_loader.load_module(item.name)

   # At least for Python2, this is not set properly, but we use it for package
   # data loading, so this manual patching up allows these to use proper methods
   # for loading their stuff as well.
   plugin_module.__package__ = scan_package.__name__
```

Missing data files in standalone

If your program fails to file data, it can cause all kinds of different behaviors, 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.

The most common error indicating file absence is of course an uncaught `FileNotFoundError` with a filename. You should figure out what package is missing files and then use `--include-package-data` (preferably), or `--include-data-dir`/`--include-data-files` to include them.

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.

A major weapon in fighting dependency creep should be applied, namely the `anti-bloat` plugin, which offers interesting abilities, that can be put to use and block unneeded imports, giving an error for where they occur. Use it e.g. like this `--noinclude-pytest-mode=nofollow --noinclude-setuptools-mode=nofollow` and e.g. also `--noinclude-custom-mode=setuptools:error` to get the compiler to error out for a specific package. Make sure to check its help output. It can take for each module of your choice, e.g. forcing also that e.g. `PyQt5` is considered uninstalled for standalone mode.

It's also driven by a configuration file, `anti-bloat.yml` that you can contribute to, removing typical bloat from packages. Feel free to enhance it and make PRs towards Nuitka with it.

Onefile: Finding files

There is a difference between `sys.argv[0]` and `__file__` of the main module for onefile more, that is caused by using a bootstrap to a temporary location. The first one will be the original executable path, where as the second one will be the temporary or permanent path the bootstrap executable unpacks to. Data files will be in the later location, your original environment files will be in the former location.

Given 2 files, one which you expect to be near your executable, and one which you expect to be inside the onefile binary, access them like this.

```
# This will find a file *near* your onefile.exe
open(os.path.join(os.path.dirname(sys.argv[0]), "user-provided-file.txt"))
# This will find a file *inside* your onefile.exe
open(os.path.join(os.path.dirname(__file__), "user-provided-file.txt"))
```

Standalone: Finding files

The standard code that normally works, also works, you should refer to `os.path.dirname(__file__)` or use all the packages like `pkgutil`, `pkg_resources`, `importlib.resources` to locate data files near the standalone binary.

> ***Important***
>
> What you should **not** do, is use the current directory `os.getcwd`, assuming that this is the script directory, that is not generally true, and was never good code. Links, to a program, etc. will all fail in bad ways.

Windows Programs without console give no errors

For debugging purposes, remove `--disable-console` or use the options `--windows-force-stdout-spec` and `--windows-force-stderr-spec` with paths as documented for `--windows-onefile-tempdir-spec` above. These can be relative to the program or absolute, so you can see the outputs given.

Deep copying uncompiled functions

Sometimes people use this kind of code, which for packages on PyPI, we deal with by doing source code patches on the fly. If this is in your own code, here is what you can do:

```
def binder(func, name):
   result = types.FunctionType(func.__code__, func.__globals__, name=func.__name__, argdefs=func.__defaults__, closure=func.__closure__)
   result = functools.update_wrapper(result, func)
   result.__kwdefaults__ = func.__kwdefaults__
   result.__name__ = name
   return result
```

Compiled functions cannot be used to create uncompiled ones from, so the above code, will not work. However, there is a dedicated `clone` method, that is specific to them, so use this instead.

```
def binder(func, name):
   try:
      result = func.clone()
```

```
   except AttributeError:
      result = types.FunctionType(func.__code__, func.__globals__, name=func.__name__, argdefs=func.__defaults__, closure=func.__closure__)
      result = functools.update_wrapper(result, func)
      result.__kwdefaults__ = func.__kwdefaults__

   result.__name__ = name
   return result
```

Tips

Nuitka Options in the code

There is support for conditional options, and options using pre-defined variables, this is an example:

```
# Compilation mode, support OS specific.
# nuitka-project-if: {OS} in ("Windows", "Linux", "Darwin", "FreeBSD"):
#    nuitka-project: --onefile
# nuitka-project-if: {OS} not in ("Windows", "Linux", "Darwin", "FreeBSD"):
#    nuitka-project: --standalone

# The PySide2 plugin covers qt-plugins
# nuitka-project: --enable-plugin=pyside2
# nuitka-project: --include-qt-plugins=sensible,qml
```

The comments must be a start of line, and indentation is to be used, to end a conditional block, much like in Python. There are currently no other keywords than the used ones demonstrated above.

You can put arbitrary Python expressions there, and if you wanted to e.g. access a version information of a package, you could simply use `__import__("module_name").__version__` if that would be required to e.g. enable or disable certain Nuitka settings. The only thing Nuitka does that makes this not Python expressions, is expanding `{variable}` for a pre-defined set of variables:

Table with supported variables:

Variable	What this Expands to	Example
{OS}	Name of the OS used	Linux, Windows, Darwin, FreeBSD, OpenBSD
{Version}	Version of Nuitka	e.g. (0, 6, 16)
{Commercial}	Version of Nuitka Commercial	e.g. (0, 9, 4)
{Arch}	Architecture used	x86_64, arm64, etc.
{MAIN_DIRECTORY}	Directory of the compiled file	some_dir/maybe_relative
{Flavor}	Variant of Python	e.g. Debian Python, Anaconda Python

The use of `{MAIN_DIRECTORY}` is recommended when you want to specify a filename relative to the main script, e.g. for use in data file options or user package configuration yaml files,

```
# nuitka-project: --include-data-files={MAIN_DIRECTORY}/my_icon.png=my_icon.png
# nuitka-project: --user-package-configuration-file={MAIN_DIRECTORY}/user.nuitka-package.config.yml
```

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 where things might be non-standard.

For the MSVC compilers and ClangCL setups, using the `clcache` is automatic and included in Nuitka.

Control where Caches live

The storage for cache results of all kinds, downloads, cached compilation results from C and Nuitka, is done in a platform dependent directory as determined by the `appdirs` package. However, you can override it with setting the environment variable `NUITKA_CACHE_DIR` to a base directory. This is for use in environments where the home directory is not persisted, but other paths are.

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, Debian Python2, self compiled Pythons (do not activate `--enable-shared`, not needed), and installs created with `pyenv`.

Note

On Anaconda, you may need to execute `conda install libpython-static`

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.

There is plenty of ways to find that something is missing. Do not manually copy things into the folder, esp. not DLLs, as that's not going to work. Instead make bug reports to get these handled by Nuitka properly.

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 redistribution

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 handles 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, 3.10
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

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, 3.10

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

Nuitka does *not* `sys.frozen` unlike other tools, because it usually triggers inferior code for no reason. For Nuitka, we have the module attribute `__compiled__` to test if a specific module was compiled, and the function attribute `__compiled__` to test if a specific function was compiled.

Providing extra Options to Nuitka C compilation

Nuitka will apply values from the environment variables `CCFLAGS`, `LDFLAGS` during the compilation on top of what it determines to be necessary. Beware of course, that is this is only useful if you know what you are doing, so should this pose an issues, raise them only with perfect information.

Producing a 32 bit binary on a 64 bit Windows system

Nuitka will automatically target the architecture of the Python you are using. If this is 64 bits, it will create a 64 bits binary, if it is 32 bits, it will create a 32 bits binary. You have the option to select the bits when you download the Python. In the output of `python -m nuitka --version` there is a line for the architecture. It `Arch: x86_64` for 64 bits, and just `Arch: x86` for 32 bits.

The C compiler will be picked to match that more or less automatically. If you specify it explicitly and it mismatches, you will get a warning about the mismatch and informed that you compiler choice was rejected.

Performance

This chapter gives an overview, of what to currently expect in terms of performance from Nuitka. It's a work in progress and is updated as we go. The current focus for performance measurements is Python 2.7, but 3.x is going to follow later.

pystone results

The results are the top value from this kind of output, running pystone 1000 times and taking the minimal value. The idea is that the fastest run is most meaningful, and eliminates usage spikes.

```
echo "Uncompiled Python2"
for i in {1..100}; do BENCH=1 python2 tests/benchmarks/pystone.py ; done | sort -n -r | head -n 1
python2 -m nuitka --lto=yes --pgo=yes tests/benchmarks/pystone.py
echo "Compiled Python2"
for i in {1..100}; do BENCH=1 ./pystone.bin ; done | sort -n -r | head -n 1

echo "Uncompiled Python3"
for i in {1..100}; do BENCH=1 python3 tests/benchmarks/pystone3.py ; done | sort -n -r | head -n 1
python3 -m nuitka --lto=yes --pgo=yes tests/benchmarks/pystone3.py
echo "Compiled Python3"
for i in {1..100}; do BENCH=1 ./pystone3.bin ; done | sort -n -r | head -n 1
```

Python	Uncompiled	Compiled LTO	Compiled PGO
Debian Python 2.7	137497.87 (1.000)	460995.20 (3.353)	503681.91 (3.663)
Nuitka Python 2.7	144074.78 (1.048)	479271.51 (3.486)	511247.44 (3.718)

Where to go next

Remember, this project needs constant work. Although the Python compatibility is insanely high, and test suite works near perfectly, there is still more work needed, esp. to make it do more optimization. Try it out, and when popular packages do not work, please make reports on GitHub.

Follow me on Mastodon and Twitter

Nuitka announcements and interesting stuff is pointed to on both the Mastodon and Twitter accounts, but obviously with not too many details, usually pointing to the website, but sometimes I also ask questions there.

@KayHayen on Mastodon. @KayHayen on Twitter.

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. It does contain more information that you think. Do not write something manually. You may always add of course.

  ```
  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:

- `main`

  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 fetch && 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.

Updates for this Manual

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

You will find the current version at: https://nuitka.net/doc/user-manual.html

---

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 https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx (the community editions work just fine).

The latest version is recommended but not required. On the other hand, there is no need to except pre-Windows 10 support, and they might work for you, but support of these configurations is only available to commercial users.