Nuitka User Manual

This document is the recommended first read when you start using **Nuitka**. On this page, you will learn more about **Nuitka** fundamentals, such as license type, use cases, requirements, and credits.

Table of Contents

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 Python 2 (2.6, 2.7) and Python 3 (3.4 - 3.11) 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.

Requirements

To ensure smooth operation of **Nuitka**, make sure to follow system requirements, that include the following components:

C Compiler

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

[1]Support for this C11 is given with gcc 5.x or higher or any clang version.
The older MSVC compilers don't do it yet. But as a workaround, with Python 3.10 or older, the C++03 language standard is significantly overlapping with C11, it is then used instead.

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]. 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

Python 2 (2.6, 2.7) and **Python 3** (3.4 — 3.11) are supported. If at any moment, there is a stable Python release that is not in this list, rest assured it is being worked on and will be added.

Important

For Python 3.4 and *only* that version, 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. 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

[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 to support pre-Windows 10 versions, and they might work for you, but support of these configurations is only available to commercial users.

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 overwrite the binary without destroying the original source file.

It **has to** be CPython, Anaconda Python, or Homebrew

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 the Windows app store

It is known that Windows app store Python definitely does not work, it’s checked against.

It **cannot be** pyenv on macOS

It is known that macOS ”pyenv” does **not** work. Use Homebrew instead for self compiled Python installations. But note that standalone mode will be worse on these platforms and not be as backward compatible with older macOS versions.

Operating System

Supported Operating Systems: Linux, FreeBSD, NetBSD, macOS, and Windows (32 bits/64 bits/ARM).

Others will work as well. The portability is expected to be generally good, but the e.g. Nuitka’s internal Scons usage may have to be adapted or need flags passed. Make sure to match Python and C compiler architecture, or else you will get cryptic error messages.

Architecture

Supported Architectures are x86, x86_64 (amd64), and arm, likely many, 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.

Usage

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.

Notice for integration with GitHub workflows there is this Nuitka-Action that you should use that makes it really easy to integrate. You ought to start with a local compilation though, but this will be easiest for cross platform compilation with Nuitka.

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**

```python
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 fewer 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, and that come from a directory, for everything from your Python installation, use `--include-module` or `--include-package`.

Note

The resulting filename will be `program.exe` on Windows, `program.bin` on other platforms, but `--output-filename` allows changing that.

Note

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

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

Important

The filename of the produced extension module must not be changed as Python insists on a module name derived function as an entry point, in this case `PyInit_some_module` and renaming the file will not change that. Match the filename of the source code to what the binary name should be.

Note

If both the extension module and the source code of it are in the same directory, the extension module is loaded. Changes to the source code only have effect once you recompile.

Note

The option `--follow-import-to` works 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 like, 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 the target has to be specified relative. For the standalone mode, you can also copy them manually, but this can do extra checks, and for the 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.

Important

Nuitka does not consider data files code, do not include DLLs, or Python files as data files, and expect them to work, they will not, unless you really know what you are doing.

In the following, non-code data files are all files, not matching on of these criterions.

Suffix	Rationale	Solution
`.py`	Nuitka trims even the stdlib modules to be included. If it doesn't see Python code, there is no dependencies analyzed, and as a result it will just not work.	Use `--include-module` on them instead
`.pyc`	Same as `.py`.	Use `--include-module` on them from their source code instead.
`.pyo`	Same as `.pyc`.	Use `--include-module` on them from their source code instead.
`.pyw`	Same as `.py`.	For including multiple programs, use multiple `--main` arguments instead.
`.pyi`	These are ignored, because they are code-like and not needed at run time. For the `lazy` package that actually would depend on them, we made a compile time solution that removes the need.	Raise an issue if 3rd part software needs it.
`.pyx`	These are ignored, because they are Cython source code not used at run time	

Suffix	Rationale	Solution
`.dll`	These are ignored, since they **usually** are not data files. For the cases where 3rd party packages do actually used them as data, e.g. `.NET` packages, we solve that in package configuration for it.	Create Nuitka Package configuration for those, with `dll` section for the package that uses them. For rare cases, data-files section with special configuration might be the correct thing to do.
`.dylib`	These are ignored, since they macOS extension modules or DLLs.	Need to add configuration with `dll` section or `depends` that are missing
`.so`	These are ignored, since they Linux, BSD, etc. extension modules or DLLs.	Need to add configuration with `dll` section or `depends` that are missing
`.exe`	The are binaries to Windows.	You can add Nuitka Package configuration to include those as DLLs and mark them as `executable: yes`
`.bin`	The are binaries to non-Windows, otherwise same as `.exe`.	

Also folders are ignored, these are `site-packages`, `dist-packages` and `vendor-packages` which would otherwise include a full virtualenv, which is never a good thing to happen. And the `__pycache__` folder is also always ignored. On non-MacOS the file `.DS_Store` is ignored too, and `py.typed` folders have only meaning to IDEs, and are ignored like `.pyi` files .

To copy a whole folder with all non-code files, you can use `--include-data-dir=/path/to/images=images` which will place those in the destination, and if you want to use the `--noinclude-data-files` option to remove them. Code files are as detailed above DLLs, executables, Python files, etc. and will be ignored. For those you can use the `--include-data-files=/binaries/*.exe=binary/` form to force them, but that is not recommended and known to cause issues at run-time.

For package data, there is a better way, namely using `--include-package-data`, which detects all non-code data files of packages automatically and copies them over. It even accepts patterns in a shell style. It spares you the need to find the package directory yourself and should be preferred whenever available. Functionally it's very similar to `--include-data-dir` but it has the benefit to locate the correct folder for you.

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. Even better, raise PRs with enhancements to the

Nuitka package configuration. With want 3rd party software to just work out of the box.

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}` which uses version information, and user-specific cache directory.

Note

Using cached paths will be relevant, e.g. 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\...\AppData\Locals\Temp
{PID}	Process ID	2772
{TIME}	Time in seconds since the epoch.	1299852985
{PROGRAM}	Full program run-time filename of executable.	C:\SomeWhere\YourOnefile.exe
{PROGRAM_BASE}	No-suffix of run-time filename of executable.	C:\SomeWhere\YourOnefile
{CACHE_DIR}	Cache directory for the user.	C:\Users\SomeBody\AppData\Local
{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
{NONE}	When provided for file outputs, `None` is used	see notice below

Token	What this Expands to	Example
{NULL}	When provided for file outputs, `os.devnull` is used	see notice below

Important

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.

Important

For disabling output and stderr with `--force-stdout-spec` and `--force-stderr-spec` the values `{NONE}` and `{NULL}` achieve it, but with different effect. With `{NONE}`, the corresponding handle becomes `None`. As a result, e.g. `sys.stdout` will be `None`, which is different from `{NULL}` where it will be backed by a file pointing to `os.devnull`, i.e. you can write to it.

With `{NONE}`, you may e.g. get `RuntimeError: lost sys.stdout` in case it does get used; with `{NULL}` that never happens. However, some libraries handle this as input for their logging mechanism, and on Windows this is how you are compatible with `pythonw.exe` which is behaving like `{NONE}`.

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.

Let's 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 you don't use other build systems:
setup(
   # Data files are to be handled by setuptools and not Nuitka
   package_data={"some_package": ["some_file.txt"]},
   ...,
   # This is to pass Nuitka options.
   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(
    # Data files are to be handled by setuptools and not Nuitka
    package_data={"some_package": ["some_file.txt"]},
    ...,
    # This is to pass Nuitka options.
    ...,
    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 want 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 the 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.getenv("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"

# Data files are to be handled by setuptools and not Nuitka
[tool.setuptools.package-data]
some_package = ['data_file.txt']

[tool.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`.

Note

Whatever approach you take, data files in these wheels are not handled by Nuitka at all, but by setuptools. You can, however, use the data file embedding of Nuitka commercial. In that case, you actually would embed the files inside the extension module itself, and not as a file in the wheel.

Use Case 6 — Multidist

If you have multiple programs, that each should be executable, in the past you had to compile multiple times, and deploy all of these. With standalone mode, this, of course, meant that you were fairly wasteful, as sharing the folders could be done, but wasn't really supported by Nuitka.

Enter `Multidist`. There is an option `--main` that replaces or adds to the positional argument given. And it can be given multiple times. When given multiple times, Nuitka will create a binary that contains the code of all the programs given, but sharing modules used in them. They therefore do not have to be distributed multiple times.

Let's call the basename of the main path, and entry point. The names of these must, of course, be different. Then the created binary can execute either entry point, and will react to what `sys.argv[0]` appears to it. So if executed in the right way (with something like `subprocess` or OS API you can control this name), or by renaming or copying the binary, or symlinking to it, you can then achieve the miracle.

This allows to combine very different programs into one.

Note

This feature is still experimental. Use with care and report your findings should you encounter anything that is undesirable behavior

This mode works with standalone, onefile, and mere acceleration. It does not work with module mode.

Use Case 7 — Building with GitHub Workflows

For integration with GitHub workflows there is this Nuitka-Action that you should use that makes it really easy to integrate. You ought to start with a local compilation though, but this will be easiest for cross platform compilation with Nuitka.

This is an example workflow that builds on all 3 OSes

```
jobs:
build:
   strategy:
      matrix:
      os: [macos-latest, ubuntu-latest, windows-latest]

   runs-on: ${{ matrix.os }}

   steps:
      - name: Check-out repository
      uses: actions/checkout@v3

      - name: Setup Python
      uses: actions/setup-python@v4
      with:
         python-version: '3.10'
         cache: 'pip'
```

```yaml
            cache-dependency-path: |
               **/requirements*.txt

      - name: Install your Dependencies
        run: |
           pip install -r requirements.txt -r requirements-dev.txt

      - name: Build Executable with Nuitka
        uses: Nuitka/Nuitka-Action@main
        with:
           nuitka-version: main
           script-name: your_main_program.py
           # many more Nuitka options available, see action doc, but it's best
           # to use nuitka-project: options in your code, so e.g. you can make
           # a difference for macOS and create an app bundle there.
           onefile: true

      - name: Upload Artifacts
        uses: actions/upload-artifact@v3
        with:
           name: ${{ runner.os }} Build
           path: | # match what's created for the 3 OSes
              build/*.exe
              build/*.bin
              build/*.app/**/*
```

If you app is a GUI, e.g. `your_main_program.py` should contain these comments as explained in Nuitka Options in the code since on macOS this should then be a bundle.

```python
# Compilation mode, standalone everywhere, except on macOS there app bundle
# nuitka-project-if: {OS} in ("Windows", "Linux", "FreeBSD"):
#    nuitka-project: --onefile
# nuitka-project-if: {OS} == "Darwin":
#    nuitka-project: --standalone
#    nuitka-project: --macos-create-app-bundle
#
# Debugging options, controlled via environment variable at compile time.
# nuitka-project-if: os.getenv("DEBUG_COMPILATION", "no") == "yes"
#     nuitka-project: --enable-console
# nuitka-project-else:
#     nuitka-project: --disable-console
```

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 formats 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:Microphon` `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 the splash screen, you need to specify it as a PNG file, and then make sure to disable the splash screen when your program is ready, e.g. has completed 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, tempfile, os
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.
```

Reports

For analysis of your program and Nuitka packaging, there is the Compilation Report available. You can also make custom reports by providing your template, with a few of them built-in to Nuitka. These reports carry all the detail information, e.g. when a module was attempted to be imported, but not found, you can see where that happens. For bug reporting, it is very much recommended to provide the report.

Version Information

You can attach copyright and trademark information, company name, product name, and so on to your compilation. This is then used in version information for the created binary on Windows, or application bundle on macOS. If you find something that is lacking, please let us know.

Typical Problems

Deployment Mode

By default, Nuitka compiles without `--deployment` which leaves a set of safe guards and helpers on, that are aimed at debugging wrong uses of Nuitka.

This is a new feature, and implements a bunch of protections and helpers, that are documented here.

Fork bombs (self-execution)

So after compilation, `sys.executable` is the compiled binary. In case of packages like `multiprocessing`, `joblib`, or `loky` what these typically do is to expect to run from a full `python` with `sys.executable` and then to be able to use its options like `-c command` or `-m module_name` and then be able to launch other code temporarily or permanently as a service daemon.

With Nuitka however, this executes your program again, and puts these arguments, in `sys.argv` where you maybe ignore them, and then you fork yourself again to launch the helper daemons. Sometimes this ends up spawning CPU count processes that spawn CPU count processes that... this is called a fork bomb, and with almost all systems, that freezes them easily to death.

That is why e.g. this happens with default Nuitka:

```
./hello.dist/hello.bin -l fooL -m fooM -n fooN -o fooO -p
Error, the program tried to call itself with '-m' argument. Disable with '--no-deployment-fl
```

Your program may well have its own command line parsing, and not use an unsupported package that does attempt to re-execute. In this case, you need at *compile time* to use `--no-deployment-flag=self-execution` which disables this specific guard.

Misleading Messages

Some packages output what they think is helpful information about what the reason of a failed import might mean. With compiled programs there are very often just plain wrong. We try and repair those in non-deployment mode. Here is an example, where we change a message that asks to pip install (which is not the issue) to point the user to the include command that makes an `imageio` plugin work.

```
- module-name: 'imageio.core.imopen'
  anti-bloat:
    - replacements_plain:
        '`pip install imageio[{config.install_name}]` to install it': '`--include-module={co
        'err_type = ImportError': 'err_type = RuntimeError'
      when: 'not deployment'
```

And much more

The deployment mode is relatively new and has constantly more features added, e.g. something for `FileNotFoundError` should be coming soon.

Disabling All

All these helpers can of course be disabled at once with `--deployment` but keep in mind that for debugging, you may want to re-enable it. You might want to use Nuitka Project options and an environment variable to make this conditional.

Should you disable them all?

We believe, disabling should only happen selectively, but with PyPI upgrades, your code changes, all of these issues can sneak back in. The space saving of deployment mode is currently negligible, so attempt to not do it, but review what exists, and if you know that it cannot affect you, or if it does, you will not need it. Some of the future ones, will clearly be geared at beginner level usage.

Windows Virus scanners

Binaries compiled on Windows with default settings of Nuitka and no further actions taken might be recognized by some AV vendors as malware. This is avoidable, but only in Nuitka commercial there is actual support and instructions for how to do it, seeing this as a typical commercial only need. https://nuitka.net/doc/commercial.html

Linux Standalone

For Linux standalone it is pretty difficult to build a binary that works on other Linux versions. This is mainly because on Linux, much software is built specifically targeted to concrete DLLs. Things like glibc used, are then encoded into the binary built, and it will not run with an older glibc, just to give one critical example.

The solution is to build on the oldest OS that you want to see supported. Picking that and setting it up can be tedious, so can be login, and keeping it secure, as it's something you put your source code on.

To aid that, Nuitka commercial has container based builds, that you can use. This uses dedicated optimized Python builds, targets CentOS 7 and supports even newest Pythons and very old OSes that way using recent C compiler chains all turn key solution. The effort needs to be compensated to support Nuitka development for Linux, there you need to purchase it https://nuitka.net/doc/commercial.html but even a sponsor license will be cheaper than doing it yourself.

Program crashes system (fork bombs)

A fork bomb is a program that starts itself over and over. This can easily happen, since `sys.executable` for compiled programs is not a Python interpreter, and packages that try to do multiprocessing in a better way, often relaunch themselves through this, and Nuitka needs and does have handling for these with known packages. However, you may encounter a situation where the detection of this fails. See deployment option above that is needed to disable this protection.

When this fork bomb happens easily all memory, all CPU of the system that is available to the user is being used, and even the most powerful build system will go down in flames sometimes needing a hard reboot.

For fork bombs, we can use `--experimental=debug-self-forking` and see what it does, and we have a trick, that prevents fork bombs from having any actual success in their bombing. Put this at the start of your program.

```python
import os, sys

if "NUITKA_LAUNCH_TOKEN" not in os.environ:
   sys.exit("Error, need launch token or else fork bomb suspected.")
else:
   del os.environ["NUITKA_LAUNCH_TOKEN"]
```

Actually Nuitka is trying to get ahold of them without the deployment option already, finding ”-c” and ”-m” options, but it may not be perfect or not work well with a package (anymore).

Memory issues and compiler bugs

In some cases, the C compilers will crash saying they cannot allocate memory or that some input was truncated, or similar error messages, clearly from it. These are example error messages, that are a sure sign of too low memory, there is no end to them.

```
# gcc
fatal error: error writing to -: Invalid argument
Killed signal terminated program
# MSVC
fatal error C1002: compiler is out of heap space in pass 2
fatal error C1001: Internal compiler error
```

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 bit 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 bit 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 troublemakers, they also require more memory, so get rid of that. Of course, you do have to check that your program has all the 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 your 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 an 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`, e.g. inserts 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 e.g. uses `os.listdir`, then considers Python filenames, and then opens a file and does `exec` on them. This approach works 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.

```python
# Using a package name, to locate the plugins. This is also a sane
# way to organize them into a directory.
scan_path = scan_package.__path__

for item in pkgutil.iter_modules(scan_path):
   importlib.import_module(scan_package.__name__ + "." + item.name)

   # You may want to do it recursively, but we don't do this here in
   # this example. If you'd like to, handle that in this kind of branch.
   if item.ispkg:
      ...
```

Missing data files in standalone

If your program fails to find data file, it can cause all kinds of different behavior, e.g. a package might complain it is not the right version because a `VERSION` file check defaulted to an 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 your program works without standalone, chances are data files might be the 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/EXEs 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 give 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.

If you want to manually add a DLL or an EXE because it is your project only, you will have to use user Yaml files describing where they can be found. This is described in detail with examples in the Nuitka Package Configuration page.

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. Presently, 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. Please don't hesitate to enhance it and make PRs towards Nuitka with it.

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`, or assume that this is the script directory, e.g. with paths like `data/`.

If you did that, it was never good code. Links, to a program, launching from another directory, etc. will all fail in bad ways. Do not make assumptions about the directory your program is started from.

In case you mean to refer to the location of the `.dist` folder for files that are to reside near the binary, there is `__compiled__.containing_dir` that also abstracts all differences with `--macos-create-app-bundle` and the `.app` folder a having more nested structure.

```python
# This will find a file *near* your app or dist folder
try:
   open(os.path.join(__compiled__.containing_dir, "user-provided-file.txt"))
except NameError:
   open(os.path.join(os.path.dirname(sys.argv[0]), "user-provided-file.txt"))
```

Onefile: Finding files

There is a difference between `sys.argv[0]` and `__file__` of the main module for the onefile mode, that is caused by using a bootstrap to a temporary location. The first one will be the original executable path, whereas 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.

```python
# 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"))

# This will find a file *near* your onefile binary and work for standalone too
try:
   open(os.path.join(__compiled__.containing_dir, "user-provided-file.txt"))
except NameError:
   open(os.path.join(os.path.dirname(sys.argv[0]), "user-provided-file.txt"))
```

Windows Programs without console give no errors

For debugging purposes, remove `--disable-console` or use the options `--force-stdout-spec` and `--force-stderr-spec` with paths as documented for `--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=
   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__, argde
      result = functools.update_wrapper(result, func)
      result.__kwdefaults__ = func.__kwdefaults__

   result.__name__ = name
   return result
```

Modules: Extension modules are not executable directly

A package can be compiled with Nuitka, no problem, but when it comes to executing it, `python -m compiled_module` is not going to work and give the error `No code object available for AssertsTest` because the compiled module is not source code, and Python will not just load it. The closest would be `python -c "import compile_module"` and you might have to call the main function yourself.

To support this, the CPython `runpy` and/or `ExtensionFileLoader` would need improving such that Nuitka could supply its compiled module object for Python to use.

Tips

Nuitka Options in the code

One clean way of providing options to Nuitka, that you will always use for your program, is to put them into the main file you compile. There is even support for conditional options, and options using pre-defined variables, this is an example:

```
# Compilation mode, support OS-specific options
# nuitka-project-if: {OS} in ("Windows", "Linux", "Darwin", "FreeBSD"):
#    nuitka-project: --onefile
# nuitka-project-else:
#    nuitka-project: --standalone

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

The comments must be at the start of lines, and indentation inside of them 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. (1, 6, 0)
{Commercial}	Version of Nuitka Commercial	e.g. (2, 1, 0)
{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.co
```

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 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.

On macOS and Intel, there is an automatic download of a `ccache` binary from our site, for arm64 arches, it's recommended to use this setup, which installs Homebrew and ccache in there. Nuitka picks that one up automatically if it on that kind of machine. You need and should not use Homebrew with Nuitka otherwise, it's not the best for standalone deployments, but we can take `ccache` from there.

```
export HOMEBREW_INSTALL_FROM_API=1
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/instal
eval $(/opt/homebrew/bin/brew shellenv)
brew install ccache
```

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.

There is also per cache control of these caches, here is a table of environment variables that you can set before starting the compilation, to make Nuitka store some of these caches in an entirely separate space.

Cache name	Environment Variable	Data Put there
downloads	NUITKA_CACHE_DIR_DOWNLOADS	Downloads made, e.g. dependency walker
ccache	NUITKA_CACHE_DIR_CCACHE	Object files created by gcc
clcache	NUITKA_CACHE_DIR_CLCACHE	Object files created by MSVC
bytecode	NUITKA_CACHE_DIR_BYTECODE	Bytecode of demoted modules
dll-dependencies	NUITKA_CACHE_DIR_DLL_DEPENDENCIES	DLL dependencies

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 to copy necessary libraries along with the compiled executables to the distribution folder.

There are 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 on the target machines. However, notice that compilation using the 14.3 based version is always recommended, working and best supported, unless you want to target Windows 7.

Visual C version	Redist Year	CPython
14.3	2022	3.11
14.2	2019	3.5-3.10
14.1	2017	3.5-3.8
14.0	2015	3.5-3.8
10.0	2010	3.4
9.0	2008	2.6, 2.7

When using MingGW64 as downloaded by Nuitka, you'll need the following redist versions:

MingGW64 version	Redist Year	CPython
WinLibs automatic download	2015	2.6, 2.7, 3.4- 3.11

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 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 bit, it will create a 64 bit binary, if it is 32 bit, it will create a 32 bit 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's `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 your compiler choice was rejected.

Compilation Report

When you use `--report=compilation-report.xml` Nuitka will create an XML file with detailed information about the compilation and packaging process. This is growing in completeness with every release and exposes module usage attempts, timings of the compilation, plugin influences, data file paths, DLLs, and reasons why things are included or not.

At this time, the report contains absolute paths in some places, with your private information. The goal is to make this blended out by default because we also want to become able to compare compilation reports from different setups, e.g. with updated packages, and see the changes to Nuitka. The report is, however, recommended for your bug reporting.

Also, another form is available, where the report is free form and according to a Jinja2 template of yours, and one that is included in Nuitka. The same information as used to produce the XML file is accessible. However, right now, this is not yet documented, but we plan to add a table with the data. For a reader of the source code that is familiar with Jinja2, however, it will be easy to do it now already.

If you have a template, you can use it like this `--report-template=your_template.rst.j2:your_report.rst` and of course, the usage of restructured text, is only an example. You can use Markdown, your own XML, or whatever you see fit. Nuitka will just expand the template with the compilation report data.

Currently, the following reports are included in Nuitka. You just use the name as a filename, and Nuitka will pick that one instead.

Report Name	Status	Purpose
LicenseReport	experimental	Distributions used in a compilation with license texts

Note

The community can and should contribute more report types and help enhancing the existing ones for good looks.

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

echo "Uncompiled Python3"
for i in {1..100}; do BENCH=1 python3 tests/benchmarks/pystone3.py ; done | sort -rn | head
```

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

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 than 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 the program 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.

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.