Nuitka Developer Manual

Contents

The purpose of this Developer Manual is to present the current design of Nuitka, the project rules, and the motivations for choices made. It is intended to be a guide to the source code, and to give explanations that don't fit into the source code in comments form.

It should be used as a reference for the process of planning and documenting decisions we made. Therefore we are e.g. presenting here the type inference plans before implementing them. And we update them as we proceed.

It grows out of discussions and presentations made at conferences as well as private conversations or issue tracker.

Milestones

1. Feature parity with CPython, understand all the language construct and behave absolutely compatible.

   Feature parity has been reached for CPython 2.6 and 2.7. We do not target any older CPython release. For CPython 3.3 up to 3.8 it also has been reached. We do not target the older and practically unused CPython 3.0 to 3.2 releases.

   This milestone was reached. Dropping support for Python 2.6 and 3.3 is an option, should this prove to be any benefit. Currently it is not, as it extends the test coverage only.

2. Create the most efficient native code from this. This means to be fast with the basic Python object handling.

   This milestone was reached, although of course, micro optimizations to this are happening all the time.

3. Then do constant propagation, determine as many values and useful constraints as possible at compile time and create more efficient code.

   This milestone is considered almost reached. We continue to discover new things, but the infrastructure is there, and these are easy to add.

4. Type inference, detect and special case the handling of strings, integers, lists in the program.

   This milestone is considered in progress.

5. Add interfacing to C code, so Nuitka can turn a `ctypes` binding into an efficient binding as written with C.

   This milestone is planned only.

6. Add hints module with a useful Python implementation that the compiler can use to learn about types from the programmer.

   This milestone is planned only.

Version Numbers

For Nuitka we use semantic versioning, initially with a leading zero still, once we pass release `0.9`, the scheme will indicate the `10` through using `1.0`.

Current State

Nuitka top level works like this:

- `nuitka.tree.Building` outputs node tree
- `nuitka.optimization` enhances it as best as it can

- `nuitka.finalization` prepares the tree for code generation
- `nuitka.code_generation.CodeGeneration` orchestrates the creation of code snippets
- `nuitka.code_generation.*Codes` knows how specific code kinds are created
- `nuitka.MainControl` keeps it all together

This design is intended to last.

Regarding types, the state is:

- Types are always `PyObject *`, and only a few C types, e.g. `nuitka_bool` and `nuitka_void` and more are coming. Even for objects, often it's know that things are e.g. really a `PyTupleObject **`, but no C type is available for that yet.
- There are a some specific use of types beyond "compile time constant", that are encoded in type and value shapes, which can be used to predict some operations, conditions, etc. if they raise, and result types they give.
- In code generation, the supported C types are used, and sometimes we have specialized code generation, e.g. a binary operation that takes an `int` and a `float` and produces a `float` value. There will be fallbacks to less specific types.

The expansion with more C types is currently in progress, and there will also be alternative C types, where e.g. `PyObject *` and `C long` are in an enum that indicates which value is valid, and where special code will be available that can avoid creating the `PyObject **` unless the later overflows.

Setting up the Development Environment for Nuitka

Currently there are very different kinds of files that we need support for. This is best addressed with an IDE. We cover here how to setup the most common one.

Visual Studio Code

Download Visual Studio Code from here: https://code.visualstudio.com/download

At this time, this is the recommended IDE for Linux and Windows. This is going to cover the plugins to install. Configuration is part of the `.vscode` in your Nuitka checkout. If you are not familiar with Eclipse, this is Free Software IDE,designed to be universally extended, and it truly is. There are plugins available for nearly everything.

The extensions to be installed are part of the Visual Code recommendations in `.vscode/extensions.json` and you will be prompted about that and ought to install these.

Eclipse / PyCharm

Don't use these anymore, we consider Visual Studio Code to be far superior for delivering a nice out of the box environment.

Commit and Code Hygiene

In Nuitka we have tools to auto format code, you can execute them manually, but it's probably best to execute them at commit time, to make sure when we share code, it's already well format, and to avoid noise doing cleanups.

The kinds of changes also often cause unnecessary merge conflicts, while the auto format is designed to format code also in a way that it avoids merge conflicts in the normal case, e.g. by doing imports one item per line.

In order to set up hooks, you need to execute these commands:

```
# Where python is the one you use with Nuitka, this then gets all
# development requirements, can be full PATH.
python -m pip install -r requirements-devel.txt
python ./misc/install-git-hooks.py
```

These commands will make sure that the `autoformat-nuitka-source` is run on every staged file content at the time you do the commit. For C files, it may complain unavailability of `clang-format`, follow it's advice. You may call the above tool at all times, without arguments to format call Nuitka source code.

Should you encounter problems with applying the changes to the checked out file, you can always execute it with `COMMIT_UNCHECKED=1` environment set.

Coding Rules Python

These rules should generally be adhered when working on Nuitka code. It's not library code and it's optimized for readability, and avoids all performance optimization for itself.

Tool to format

There is a tool `bin/autoformat-nuitka-source` which is to apply automatic formatting to code as much as possible. It uses `black` (internally) for consistent code formatting. The imports are sorted with `isort` for proper order.

The tool (mostly `black` and `isort`) encodes all formatting rules, and makes the decisions for us. The idea being that we can focus on actual code and do not have to care as much about other things. It also deals with Windows new lines, trailing space, etc. and even sorts PyLint disable statements.

Identifiers

Classes

Classes are camel case with leading upper case. Functions and methods are with leading verb in lower case, but also camel case. Variables and arguments are lower case with _ as a separator.

```
class SomeClass:
    def doSomething(some_parameter):
        some_var = ("foo", "bar")
```

Base classes that are abstract have their name end with `Base`, so that a meta class can use that convention, and readers immediately know, that it will not be instantiated like that.

Functions

Function calls use keyword argument preferably. These are slower in CPython, but more readable:

```
getSequenceCreationCode(
    sequence_kind=sequence_kind, element_identifiers=identifiers, context=context
)
```

When the names don't add much value, sequential calls can be done:

```
context.setLoopContinueTarget(handler_start_target)
```

Here, `setLoopContinueTarget` will be so well known that the reader is expected to know the argument names and their meaning, but it would be still better to add them. But in this instance, the variable name already indicates that it is.

Module/Package Names

Normal modules are named in camel case with leading upper case, because of their role as singleton classes. The difference between a module and a class is small enough and in the source code they are also used similarly.

For the packages, no real code is allowed in their `__init__.py` and they must be lower case, like e.g. `nuitka` or `codegen`. This is to distinguish them from the modules.

Packages shall only be used to group things. In `nuitka.code_generation` the code generation packages are located, while the main interface is `nuitka.code_generation.CodeGeneration` and may then use most of the entries as local imports.

There is no code in packages themselves. For programs, we use `__main__` package to carry the actual code.

Names of modules should be plurals if they contain classes. Example is that a `Nodes` module that contains a `Node` class.

Context Managers

Names for context manages start with `with`

In order to easily recognize that something is to be used as a context manager, we follow a pattern of naming them `withSomething`, to make that easily recognized.

```
with withEnvironmentPathAdded(os.path.join(sys.prefix, "bin")):
    with withDirectoryChange(self.qt_datadir):
        ...
```

This makes these easy to recognize even in their definition.

Prefer list contractions over built-ins

This concerns `map`, `filter`, and `apply`. Usage of these built-ins is highly discouraged within Nuitka source code. Using them is considered worth a warning by "PyLint" e.g. "Used built-in function 'map'". We should use list contractions instead, because they are more readable.

List contractions are a generalization for all of them. We love readability and with Nuitka as a compiler, there won't be any performance difference at all.

There are cases where a list contraction is faster because you can avoid to make a function call. And there may be cases, where map is faster, if a function must be called. These calls can be very expensive in CPython, and if you introduce a function, just for `map`, then it might be slower.

But of course, Nuitka is the project to free us from what is faster and to allow us to use what is more readable, so whatever is faster, we don't care. We make all options equally fast and let people choose.

For Nuitka the choice is list contractions as these are more easily changed and readable.

Look at this code examples from Python:

```
class A:
    def getX(self):
        return 1
```

```
    x = property(getX)


class B(A):
    def getX(self):
        return 2


A().x == 1  # True
B().x == 1  # True (!)
```

This pretty much is what makes properties bad. One would hope `B().x` to be 2, but instead it's not changed. Because of the way properties take the functions and not members, and because they then are not part of the class, they cannot be overloaded without re-declaring them.

Overloading is then not at all obvious anymore. Now imagine having a setter and only overloading the getter. How to update the property easily?

So, that's not likable about them. And then we are also for clarity in these internal APIs too. Properties try and hide the fact that code needs to run and may do things. So let's not use them.

For an external API you may exactly want to hide things, but internally that has no use, and in Nuitka, every API is internal API. One exception may be the `hints` module, which will gladly use such tricks for an easier write syntax.

Coding Rules C

For the static C parts, e.g. compiled types, helper codes, the `clang-format` from LLVM project is used, the tool `autoformat-nuitka-source` does this for us.

We always have blocks for conditional statements to avoid typical mistakes made by adding a statement to a branch, forgetting to make it a block.

The "git flow" model

- The flow is used for releases and occasionally subsequent hot fixes.

  A few feature branches were used so far. It allows for quick delivery of fixes to both the stable and the development version, supported by a git plug-in, that can be installed via "apt-get install git-flow".

- Stable (`main` branch)

  The stable version, is expected to pass all the tests at all times and is fully supported. As soon as bugs are discovered, they are fixed as hot fixes, and then merged to develop by the "git flow" automatically.

- Development (`develop` branch)

  The future release, supposedly in almost ready for release state at nearly all times, but this is as strict. It is not officially supported, and may have problems and at times inconsistencies. Normally this branch is supposed to not be rebased. For severe problems it may be done though.

- Factory (default feature branch)

  Code under construction. We publish commits there, that may not hold up in testing, and before it enters develop branch. Factory may have severe regressions frequently, and commits become **rebased all the time**, so do not base your patches on it, please prefer the `develop` branch for that, unless of course, it's about factory code itself.

- Personal branches (jorj, orsiris, others as well)

  We are currently not using this, but it's an option.

- Feature Branches

  We are not currently using these. They could be used for long lived changes that extend for multiple release cycles and are not ready yet. Currently we perform all changes in steps that can be included in releases or delay making those changes.

Nuitka "git/github" Workflow

- Forking and cloning

  You need to have git installed and GitHub account. Goto Nuitka repository <https://github.com/Nuitka/Nuitka> and fork the repository.

  To clone it to your local machine execute the following your git bash:

```
git clone https://github.com/your-user-name/Nuitka.git
cd Nuitka
git remote add upstream https://github.com/Nuitka/Nuitka.git
```

- Create a Branch

```
git checkout develop
git pull --rebase upstream
git checkout -b feature_branch
```

  If you are having merge conflicts while doing the previous step, then check out (DON'T FORGET TO SAVE YOUR CHANGES FIRST IF ANY): <https://stackoverflow.com/questions/1125968/how-do-i-force-git-pull-to-overwrite-local-files>

- In case you have an existing branch rebase it to develop

```
git fetch upstream
git rebase upstream/develop
```

  Fix the merge conflicts if any and continue or skip commit if it is not your. Sometimes for important bug fixes, develop history gets rewritten. In that case, old and new commits will conflict during your rebase, and skipping is the best way to go.

```
git rebase --continue
# not your commit:
git rebase --skip
```

  If anything goes wrong while rebasing:

```
git rebase --abort
```

- Making changes

```
git commit -a -m "Commit Message"
git push -u origin # once, later always:
git push
```

API Documentation and Guidelines

There is API documentation generated with `doxygen`, available at this location .

To ensure meaningful `doxygen` output, the following guidelines must be observed when creating or updating Python source:

Use of Standard Python __doc__ Strings

Every class and every method should be documented via the standard Python delimiters (`""" ... """`) in the usual way.

Special doxygen Anatomy of __doc__

> ***Note***
>
> We are replacing Doxygen with sphinx, this is all obsolete

- Immediately after the leading `"""`, and after 1 space on the same line, enter a brief description or title of the class or method. This must be 1 line and be followed by at least 1 empty line.

- Depending on the item, choose from the following "sections" to describe what the item is and does.

  Each section name is coded on its own line, aligned with the leading `"""` and followed by a colon ":". Anything following the section, must start on a new line and be indented by 4 spaces relative to the section. Except for the first section (`Notes:`) after the title, sections need not be preceded by empty lines -- but it is good practice to still do that.

  - `Notes:` detailed description of the item, any length.

    May contain line breaks with each new line starting aligned with previous one. The text will automatically be joined across line breaks and be reformatted in the browser.

    If you describe details for a class, you can do so **without** using this section header and all formatting will still work fine. If you however omit the `Notes:` for methods, then the text will be interpreted **as code**, be shown in an ugly monospaced font, and no automatic line breaks will occur in the browser.

  - `Args:` positional arguments.

    Each argument then follows, starting on a new line and indented by 4 spaces. The argument name must be followed by a colon `:` or double hash `--`, followed by a description of arbitrary length.

    The description can be separated by line breaks.

  - `Kwargs:` keyword arguments. Same rules as for args.
  - `Returns:` description of what will be returned if applicable (any length).
  - `Yields:` synonymous for `Returns:`.
  - `Raises:` name any exceptions that may be raised.
  - `Examples:` specify any example code.

```
def foo(p1, p2, kw1=None, kw2=None):
    """This is an example method.

    Notes:
        It does one or the other indispensable things based on some parameters
        and proudly returns a dictionary.

    Args:
        p1: parameter one
        p2: parameter two

    Kwargs:
        kw1: keyword one
        kw2: keyword two

    Returns:
        A dictionary calculated from the input.

    Raises:
        ValueError, IndexError

    Examples:
        >>> foo(1, 2, kw1=3, kw2=4)
        {'a': 4, 'b': 6}
    """
```

Checking the Source

The static checking for errors is currently done with `PyLint`. In the future, Nuitka itself will gain the ability to present its findings in a similar way, but this is not a priority, and we are not there yet.

So, we currently use `PyLint` with options defined in a script.

```
./bin/check-nuitka-with-pylint
```

The above command is expected to give no warnings. It is also run on our CI and we will not merge branches that do not pass.

Running the Tests

This section describes how to run Nuitka tests.

Running all Tests

The top level access to the tests is as simple as this:

```
./tests/run-tests
```

For fine grained control, it has the following options:

```
--skip-basic-tests    The basic tests, execute these to check if Nuitka is
                      healthy. Default is True.
--skip-syntax-tests   The syntax tests, execute these to check if Nuitka
                      handles Syntax errors fine. Default is True.
--skip-program-tests  The programs tests, execute these to check if Nuitka
                      handles programs, e.g. import recursions, etc. fine.
                      Default is True.
--skip-package-tests  The packages tests, execute these to check if Nuitka
                      handles packages, e.g. import recursions, etc. fine.
                      Default is True.
--skip-optimizations-tests
                      The optimization tests, execute these to check if
                      Nuitka does optimize certain constructs fully away.
                      Default is True.
--skip-standalone-tests
                      The standalone tests, execute these to check if Nuitka
                      standalone mode, e.g. not referring to outside,
                      important 3rd library packages like PyQt fine. Default
                      is True.
--skip-reflection-test
                      The reflection test compiles Nuitka with Nuitka, and
                      then Nuitka with the compile Nuitka and compares the
                      outputs. Default is True.
--skip-cpython26-tests
                      The standard CPython2.6 test suite. Execute this for
                      all corner cases to be covered. With Python 2.7 this
                      covers exception behavior quite well. Default is True.
--skip-cpython27-tests
                      The standard CPython2.7 test suite. Execute this for
```

```
                      all corner cases to be covered. With Python 2.6 these
                      are not run. Default is True.
--skip-cpython32-tests
                      The standard CPython3.2 test suite. Execute this for
                      all corner cases to be covered. With Python 2.6 these
                      are not run. Default is True.
--skip-cpython33-tests
                      The standard CPython3.3 test suite. Execute this for
                      all corner cases to be covered. With Python 2.x these
                      are not run. Default is True.
--skip-cpython34-tests
                      The standard CPython3.4 test suite. Execute this for
                      all corner cases to be covered. With Python 2.x these
                      are not run. Default is True.
--skip-cpython35-tests
                      The standard CPython3.5 test suite. Execute this for
                      all corner cases to be covered. With Python 2.x these
                      are not run. Default is True.
--skip-cpython36-tests
                      The standard CPython3.6 test suite. Execute this for
                      all corner cases to be covered. With Python 2.x these
                      are not run. Default is True.
--skip-cpython37-tests
                      The standard CPython3.7 test suite. Execute this for
                      all corner cases to be covered. With Python 2.x these
                      are not run. Default is True.
--skip-cpython38-tests
                      The standard CPython3.8 test suite. Execute this for
                      all corner cases to be covered. With Python 2.x these
                      are not run. Default is True.
--skip-cpython39-tests
                      The standard CPython3.9 test suite. Execute this for
                      all corner cases to be covered. With Python 2.x these
                      are not run. Default is True.
--skip-cpython310-tests
                      The standard CPython3.10 test suite. Execute this for
                      all corner cases to be covered. With Python 2.x these
                      are not run. Default is True.
--no-python2.6        Do not use Python 2.6 even if available on the system.
                      Default is False.
--no-python2.7        Do not use Python 2.7 even if available on the system.
                      Default is False.
--no-python3.3        Do not use Python 3.3 even if available on the system.
                      Default is False.
--no-python3.4        Do not use Python 3.4 even if available on the system.
                      Default is False.
--no-python3.5        Do not use Python 3.5 even if available on the system.
                      Default is False.
--no-python3.6        Do not use Python 3.6 even if available on the system.
                      Default is False.
--no-python3.7        Do not use Python 3.7 even if available on the system.
                      Default is False.
--no-python3.8        Do not use Python 3.8 even if available on the system.
                      Default is False.
```

```
--no-python3.9        Do not use Python 3.9 even if available on the system.
                      Default is False.
--no-python3.10       Do not use Python 3.10 even if available on the system.
                      Default is False.
--coverage            Make a coverage analysis, that does not really check.
                      Default is False.
```

You will only run the CPython test suites, if you have the submodules of the Nuitka git repository checked out. Otherwise, these will be skipped with a warning that they are not available.

The policy is generally, that `./test/run-tests` running and passing all the tests on Linux and Windows shall be considered sufficient for a release, but of course, depending on changes going on, that might have to be expanded.

Basic Tests

You can run the "basic" tests like this:

```
./tests/basics/run_all.py search
```

These tests normally give sufficient coverage to assume that a change is correct, if these "basic" tests pass. The most important constructs and built-ins are exercised.

To control the Python version used for testing, you can set the `PYTHON` environment variable to e.g. `python3.5` (can also be full path), or simply execute the `run_all.py` script directly with the intended version, as it is portable across all supported Python versions, and defaults testing with the Python version is run with.

Syntax Tests

Then there are "syntax" tests, i.e. language constructs that need to give a syntax error.

It sometimes so happens that Nuitka must do this itself, because the `ast.parse` doesn't see the problem and raises no `SyntaxError` of its own. These cases are then covered by tests to make sure they work as expected.

Using the `global` statement on a function argument is an example of this. These tests make sure that the errors of Nuitka and CPython are totally the same for this:

```
./tests/syntax/run_all.py search
```

Program Tests

Then there are small "programs" tests, that e.g. exercise many kinds of import tricks and are designed to reveal problems with inter-module behavior. These can be run like this:

```
./tests/programs/run_all.py search
```

Generated Tests

There are tests, which are generated from Jinja2 templates. They aim at e.g. combining at types with operations, in-place or not, or large constants. These can be run like this:

```
./tests/generated/run_all.py search
```

Compile Nuitka with Nuitka

And there is the "compile itself" or "reflected" test. This test makes Nuitka compile itself and compare the resulting C++ when running compiled to non-compiled, which helps to find in-determinism.

The test compiles every module of Nuitka into an extension module and all of Nuitka into a single binary.

That test case also gives good coverage of the `import` mechanisms, because Nuitka uses a lot of packages and imports between them.

```
./tests/reflected/compile_itself.py
```

Internal/Plugin API

The documentation from the source code for both the Python and the C parts are published as Nuitka API and arguably in a relatively bad shape as we started generating those with Doxygen only relatively late.

```
doxygen ./doc/Doxyfile
xdg-open html
```

Improvements have already been implemented for plugins: The plugin base class defined in `PluginBase.py` (which is used as a template for all plugins) is fully documented in Doxygen now. The same is true for the recently added standard plugins `NumpyPlugin.py` and `TkinterPlugin.py`. These will be uploaded very soon.

Going forward, this will also happen for the remaining standard plugins.

Please find here a detailed description of how to write your own plugin.

To learn about plugin option specification consult this document.

Working with the CPython suites

The CPython test suites are different branches of the same submodule. When you update your git checkout, they will frequently become detached. In this case, simply execute this command:

```
git submodule foreach 'git fetch && git checkout $(basename $(pwd)) && \
git reset --hard origin/$(basename $(pwd))'
```

When adding a test suite, for a new version, proceed like this:

```
# Switch to a new branch.
git checkout CPython39
git branch CPython310
git checkout CPython310

# Delete all but root commit
git rebase -i root
rm -rf test
cp ~/repos/Nuitka-references/final/Python-3.10.0/Lib/test test
git add test
```

```
# Update commit message to mention proper Python version.
git commit --amend

# Push to github, setting upstream for branch.
git push -u

# Cherry pick the removal commits from previous branches.
git log origin/CPython39 --reverse --oneline | grep ' Removed' | cut -d' ' -f1 | xargs git cherry-pick
# While being prompted for merge conflicts with the deleted files:
git status | sed -n 's/deleted by them://p' | xargs git rm --ignore-unmatch x ; git cherry-pick --continue

# Push to github, this is useful.
git push

# Cherry pick the first commit of 'run_all.py', the copy it from the last state, and amend the commits.
git log --reverse origin/CPython39 --oneline -- run_all.py | head -1 | cut -d' ' -f1 | xargs git cherry-pick
git checkout origin/CPython39 -- run_all.py
chmod +x run_all.py
sed -i -e 's#python3.9#python3.10#' run_all.py
git commit --amend --no-edit run_all.py

# Same for 'update_doctest_generated.py'
git log --reverse origin/CPython39 --oneline -- update_doctest_generated.py | head -1 | cut -d' ' -f1 | xargs git cherry-pick
git checkout origin/CPython39 -- update_doctest_generated.py
chmod +x update_doctest_generated.py
sed -i -e 's#python3.9#python3.10#' update_doctest_generated.py
git commit --amend --no-edit update_doctest_generated.py

# Same for .gitignore
git log --reverse origin/CPython39 --oneline -- .gitignore | head -1 | cut -d' ' -f1 | xargs git cherry-pick
git checkout origin/CPython39 -- .gitignore
git commit --amend --no-edit .gitignore

# Now cherry-pick all commits of test support, these disable network, audio, GUI, random filenames and more
# and are crucial for deterministic outputs and non-reliance on outside stuff.
git log --reverse origin/CPython39 --oneline -- test/support/__init__.py | tail -n +2 | cut -d' ' -f1 | xargs git cherry-pick

git push
```

Design Descriptions

These should be a lot more and contain graphics from presentations given. It will be filled in, but not now.

Nuitka Logo

The logo was submitted by "dr. Equivalent". It's source is contained in `doc/Logo` where 3 variants of the logo in SVG are placed.

- Symbol only (symbol)

```
.. image:: doc/images/Nuitka-Logo-Symbol.png
   :alt: Nuitka Logo
```

- Text next to symbol (horizontal)

```
.. image:: doc/images/Nuitka-Logo-Horizontal.png
   :alt: Nuitka Logo
```

- Text beneath symbol (vertical)

```
.. image:: doc/images/Nuitka-Logo-Vertical.png
   :alt: Nuitka Logo
```

From these logos, PNG images, and "favicons", and are derived.

The exact ImageMagick commands are in `nuitka/tools/release/Documentation`, but are not executed each time, the commands are also replicated here:

```
convert -background none doc/Logo/Nuitka-Logo-Symbol.svg doc/images/Nuitka-Logo-Symbol.png
convert -background none doc/Logo/Nuitka-Logo-Vertical.svg doc/images/Nuitka-Logo-Vertical.png
convert -background none doc/Logo/Nuitka-Logo-Horizontal.svg doc/images/Nuitka-Logo-Horizontal.png

optipng -o2 doc/images/Nuitka-Logo-Symbol.png
optipng -o2 doc/images/Nuitka-Logo-Vertical.png
optipng -o2 doc/images/Nuitka-Logo-Horizontal.png
```

Choice of the Target Language

- Choosing the target language was important decision. factors were:
    - The portability of Nuitka is decided here
    - How difficult is it to generate the code?
    - Does the Python C-API have bindings?
    - Is that language known?
    - Does the language aid to find bugs?

The *decision for C11* is ultimately one for portability, general knowledge of the language and for control over created code, e.g. being able to edit and try that quickly.

The current status is to use pure C11. All code compiles as C11, and also in terms of workaround to missing compiler support as C++03. This is mostly needed, because MSVC does not support C. Naturally we are not using any C++ features, just the allowances of C++ features that made it into C11, which is e.g. allowing late definitions of variables.

Use of Scons internally

Nuitka does not involve Scons in its user interface at all; Scons is purely used internally. Nuitka itself, being pure Python, will run without any build process just fine.

Nuitka simply prepares `<program>.build` folders with lots of files and tasks scons to execute the final build, after which Nuitka again will take control and do more work as necessary.

> ***Note***
>
> When we speak of "standalone" mode, this is handled outside of Scons, and after it, creating the ".dist" folder. This is done in `nuitka.MainControl` module.

For interfacing to Scons, there is the module `nuitka.build.SconsInterface` that will support calling `scons` - potentially from one of two inline copies (one for before / one for Python 3.5 or later). These are mainly used on Windows or when using source releases - and passing arguments to it. These arguments are passed as `key=value`, and decoded in the scons file of Nuitka.

The scons file is named `SingleExe.scons` for lack of better name. It's really wrong now, but we have yet to find a better name. It once expressed the intention to be used to create executables, but the same works for modules too, as in terms of building, and to Scons, things really are the same.

The scons file supports operation in multiple modes for many things, and modules is just one of them. It runs outside of Nuitka process scope, even with a different Python version potentially, so all the information must be passed on the command line.

What follows is the (lengthy) list of arguments that the scons file processes:

- `source_dir`

  Where is the generated C source code. Scons will just compile everything it finds there. No list of files is passed, but instead this directory is being scanned.

- `nuitka_src`

  Where do the include files and static C parts of Nuitka live. These provide e.g. the implementation of compiled function, generators, and other helper codes, this will point to where `nuitka.build` package lives normally.

- `module_mode`

  Build a module instead of a program.

- `result_base`

  This is not a full name, merely the basename for the result to be produced, but with path included, and the suffix comes from module or executable mode.

- `debug_mode`

  Enable debug mode, which is a mode, where Nuitka tries to help identify errors in itself, and will generate less optimal code. This also asks for warnings, and makes the build fail if there are any. Scons will pass different compiler options in this case.

- `python_debug`

  Compile and link against Python debug mode, which does assertions and extra checks, to identify errors, mostly related to reference counting. May make the build fail, if no debug build library of CPython is available. On Windows it is possible to install it for CPython3.5 or higher.

- `full_compat_mode`

  Full compatibility, even where it's stupid, i.e. do not provide information, even if available, in order to assert maximum compatibility. Intended to control the level of compatibility to absurd.

- `experimental_mode`

  Do things that are not yet accepted to be safe.

- `lto_mode`

  Make use of link time optimization of gcc compiler if available and known good with the compiler in question. So far, this was not found to make major differences.

- `disable_console`

  Windows subsystem mode: Disable console for windows builds.

- `unstriped_mode`

  Unstriped mode: Do not remove debug symbols.

- `clang_mode`

  Clang compiler mode, default on macOS X and FreeBSD, optional on Linux.

- `mingw_mode`

  MinGW compiler mode, optional and useful on Windows only.

- `standalone_mode`

  Building a standalone distribution for the binary.

- `show_scons`

Show scons mode, output information about Scons operation. This will e.g. also output the actual compiler used, output from compilation process, and generally debug information relating to be build process.

- `python_prefix`

  Home of Python to be compiled against, used to locate headers and libraries.

- `target_arch`

  Target architecture to build. Only meaningful on Windows.

- `python_version`

  The major version of Python built against.

- `abiflags`

  The flags needed for the Python ABI chosen. Might be necessary to find the folders for Python installations on some systems.

- `icon_path`

  The icon to use for Windows programs if given.

Locating Modules and Packages

The search for modules used is driven by `nuitka.importing.Importing` module.

- Quoting the `nuitka.importing.Importing` documentation:

  Locating modules and package source on disk.

  The actual import of a module would already execute code that changes things. Imagine a module that does `os.system()`, it would be done during compilation. People often connect to databases, and these kind of things, at import time.

  Therefore CPython exhibits the interfaces in an `imp` module in standard library, which one can use those to know ahead of time, what file import would load. For us unfortunately there is nothing in CPython that is easily accessible and gives us this functionality for packages and search paths exactly like CPython does, so we implement here a multi step search process that is compatible.

  This approach is much safer of course and there is no loss. To determine if it's from the standard library, one can abuse the attribute `__file__` of the `os` module like it's done in `isStandardLibraryPath` of this module.

  End quoting the `nuitka.importing.Importing` documentation.

- Role

  This module serves the recursion into modules and analysis if a module is a known one. It will give warnings for modules attempted to be located, but not found. These warnings are controlled by a while list inside the module.

The decision making and caching are located in the `nuitka.tree` package, in modules `nuitka.tree.Recursion` and `nuitka.tree.ImportCache`. Each module is only considered once (then cached), and we need to obey lots of user choices, e.g. to compile a standard library or not.

Hooking for module `import` process

Currently, in generated code, for every `import` a normal `__import__()` built-in call is executed. The `nuitka/build/static_src/MetaPathBasedLoader.c` file provides the implementation of a `sys.meta_path` hook.

This meta path based importer allows us to have the Nuitka provided module imported even when imported by non-compiled code.

> ***Note***
>
> Of course, it would make sense to compile time detect which module it is that is being imported and then to make it directly. At this time, we don't have this inter-module optimization yet, mid-term it should become easy to add.

Supporting `__class__` of Python3

In Python3 the handling of `__class__` and `super` is different from Python2. It used to be a normal variable, and now the following things have changed.

- The use of the `super` variable name triggers the addition of a closure variable `__class__`, as can be witnessed by the following code:

```
class X:
    def f1(self):
        print(locals())

    def f2(self):
        print(locals())
        super  # Just using the name, not even calling it.


x = X()
x.f1()
x.f2()
```

  Output is:

```
{'self': <__main__.X object at 0x7f1773762390>''} {'self':
<__main__.X object at 0x7f1773762390>, '__class__': <class
'__main__.X'>}
```

- This value of `__class__` is also available in the child functions.
- The parser marks up code objects usage of "super". It doesn't have to be a call, it can also be a local variable. If the `super` built-in is assigned to another name and that is used without arguments, it won't work unless `__class__` is taken as a closure variable.
- As can be seen in the CPython3 code, the closure value is added after the class creation is performed.
- It appears, that only functions locally defined to the class are affected and take the closure.

This left Nuitka with the strange problem, of how to emulate that.

The solution is this:

- Under Python3, usage of `__class__` as a reference in a child function body is mandatory. It remains that way until all variable names have been resolved.

- **When recognizing calls to `super` without arguments, make the arguments**

  into variable reference to `__class__` and potentially `self` (actually first argument name).

- After all variables have been known, and no suspicious unresolved calls to anything named `super` are down, then unused references are optimized away by the normal unused closure variable.

- Class dictionary definitions are added.

  These are special direct function calls, ready to propagate also "bases" and "metaclass" values, which need to be calculated outside.

  The function bodies used for classes will automatically store `__class__` as a shared local variable, if anything uses it. And if it's not assigned by user code, it doesn't show up in the "locals()" used for dictionary creation.

  Existing `__class__` local variable values are in fact provided as closure, and overridden with the built class , but they should be used for the closure giving, before the class is finished.

  So `__class__` will be local variable of the class body, until the class is built, then it will be the `__class__` itself.

Frame Stack

In Python, every function, class, and module has a frame. It is created when the scope is entered, and there is a stack of these at run time, which becomes visible in tracebacks in case of exceptions.

The choice of Nuitka is to make this an explicit element of the node tree, that are as such subject to optimization. In cases, where they are not needed, they may be removed.

Consider the following code.

```
def f():
    if someNotRaisingCall():
        return somePotentiallyRaisingCall()
    else:
        return None
```

In this example, the frame is not needed for all the code, because the condition checked wouldn't possibly raise at all. The idea is the make the frame guard explicit and then to reduce its scope whenever possible.

So we start out with code like this one:

```
def f():
    with frame_guard("f"):
        if someNotRaisingCall():
            return somePotentiallyRaisingCall()
        else:
            return None
```

This is to be optimized into:

```
def f():
    if someNotRaisingCall():
        with frame_guard("f"):
            return somePotentiallyRaisingCall()
    else:
        return None
```

Notice how the frame guard taking is limited and may be avoided, or in best cases, it might be removed completely. Also this will play a role when in-lining function. The frame stack entry will then be automatically preserved without extra care.

> ***Note***
>
> In the actual code, `nuitka.nodes.FrameNodes.StatementsFrame` is represents this as a set of statements to be guarded by a frame presence.

Parameter Parsing

The parsing of parameters is very convoluted in Python, and doing it in a compatible way is not that easy. This is a description of the required process, for an easier overview.

Input

The input is an argument `tuple` (the type is fixed), which contains the positional arguments, and potentially an argument `dict` (type is fixed as well, but could also be `NULL`, indicating that there are no keyword arguments.

Keyword dictionary

The keyword argument dictionary is checked first. Anything in there, that cannot be associated, either raise an error, or is added to a potentially given star dict argument. So there are two major cases.

- No star dict argument: Iterate over dictionary, and assign or raise errors.

  This check covers extra arguments given.

- With star dict argument: Iterate over dictionary, and assign or raise errors.

  Interesting case for optimization are no positional arguments, then no check is needed, and the keyword argument dictionary could be used as the star argument. Should it change, a copy is needed though.

What's noteworthy here, is that in comparison to the keywords, we can hope that they are the same value as we use. The interning of strings increases chances for non-compiled code to do that, esp. for short names.

We then can do a simple `is` comparison and only fall back to real string == comparisons, after all of these failed. That means more code, but also a lot faster code in the positive case.

Argument tuple

After this completed, the argument tuple is up for processing. The first thing it needs to do is to check if it's too many of them, and then to complain.

For arguments in Python2, there is the possibility of them being nested, in which case they cannot be provided in the keyword dictionary, and merely should get picked from the argument tuple.

Otherwise, the length of the argument tuple should be checked against its position and if possible, values should be taken from there. If it's already set (from the keyword dictionary), raise an error instead.

SSA form for Nuitka

The SSA form is critical to how optimization works. The so called trace collections builds up traces. These are facts about how this works:

- Assignments draw from a counter unique for the variable, which becomes the variable version. This happens during tree building phase.
- References are associated with the version of the variable active.

  This can be a merge of branches. Trace collection does do that and provides nodes with the currently active trace for a variable.

The data structures used for trace collection need to be relatively compact as the trace information can become easily much more data than the program itself.

Every trace collection has these:

- variable_actives

  Dictionary, where per "variable" the currently used version is. Used to track situations changes in branches. This is the main input for merge process.

- variable_traces

  Dictionary, where "variable" and "version" form the key. The values are objects with or without an assignment, and a list of usages, which starts out empty.

  These objects have usages appended to them. In "onVariableSet", a new version is allocated, which gives a new object for the dictionary, with an empty usages list, because each write starts a new version. In "onVariableUsage" the version is detected from the current version. It may be not set yet, which means, it's a read of an undefined value (local variable, not a parameter name), or unknown in case of global variable.

  These objects may be told that their value has escaped. This should influence the value friend they attached to the initial assignment. Each usage may have a current value friend state that is different.

When merging branches of conditional statements, the merge shall apply as follows:

- Branches have their own collection

  Thee have potentially deviating sets of `variable_actives`. These are children of an outer collections.

- Case a) One branch only.

  For that branch a collection is performed. As usual new assignments generate a new version making it "active", references then related to these "active" versions.

  Then, when the branch is merged, for all "active" variables, it is considered, if that is a change related to before the branch. If it's not the same, a merge trace with the branch condition is created with the one active in the collection before that statement.

- Case b) Two branches.

  When there are two branches, they both as are treated as above, except for the merge.

  When merging, a difference in active variables between the two branches creates the merge trace.

> ***Note***
>
> For conditional expressions, there are always only two branches. Even if you think you have more than one branch, you do not. It's always nested branches, already when it comes out of the `ast` parser.

Trace structure, there are different kinds of traces.

- Initial write of the version

  There may be an initial write for each version. It can only occur at the start of the scope, but not later, and there is only one. This might be known to be "initialized" (parameter variables of functions are like that) or "uninitialized", or "unknown".

- Merge of other one or two other versions

  This combines two or more previous versions. In cases of loop exits or entries, there are multiple branches to combine potentially. These branches can have vastly different properties.

- Becoming unknown.

  When control flow escapes, e.g. for a module variable, any write can occur to it, and it's value cannot be trusted to be unchanged. These are then traced as unknown.

All traces have a base class `ValueTraceBase` which provides the interface to query facts about the state of a variable in that trace. It's e.g. of some interest, if a variable must have a value or must not. This allows to e.g. omit checks, know what exceptions might raise.

Loop SSA

For loops we have the addition difficulty that we need would need to look ahead what types a variable has at loop exit, but that is a cyclic dependency.

Our solution is to consider the variable types at loop entry. When these change, we drop all gained information from inside the loop. We may e.g. think that a variable is a `int` or `float`, but later recognize that it can only be a float. Derivations from `int` must be discarded, and the loop analysis restarted.

Then during the loop, we assign an incomplete loop trace shape to the variable, which e.g. says it was an `int` initially and additional type shapes, e.g. `int or long` are then derived. If at the end of the loop, a type produced no new types, we know we are finished and mark the trace as a complete loop trace.

If it is not, and next time, we have the same initial types, we add the ones derived from this to the starting values, and see if this gives more types.

Python Slots in Optimization

Basic Slot Idea

For almost all the operations in Python, a form of overloading is available. That is what makes it so powerful.

So when you write an expression like this one:

```
1.0 + something
```

This something will not just blindly work when it's a float, but go through a slot mechanism, which then can be overloaded.

```
class SomeStrangeFloat:
    def __float__(self):
        return 3.14


something = SomeStrangeFloat()
# ...
1.0 + float(something) // 4.140000000000001
```

Here it is the case, that this is used by user code, but more often this is used internally. Not all types have all slots, e.g. `list` does not have `__float__` and therefore will refuse an addition to a `float` value, based on that.

Another slot is working here, that we didn't mention yet, and that is `__add__` which for some times will be these kinds of conversions or it will not do that kind of thing, e.g. something do hard checks, which is why this fails to work:

```
[] + ()
```

As a deliberate choice, there is no `__list__` slot used. The Python designers are aiming at solving many things with slots, but they also accept limitations.

There are many slots that are frequently used, most often behind your back (`__iter__`, `__next__`, `__lt__`, etc.). The list is large, and tends to grow with Python releases, but it is not endless.

Representation in Nuitka

So a slot in Nuitka typically has an owning node. We use `__len__` as an example here. In the `computeExpression` the `len` node named `ExpressionBuiltinLen` has to defer the decision what it computes to its argument.

```
def computeExpression(self, trace_collection):
    return self.subnode_value.computeExpressionLen(
        len_node=self, trace_collection=trace_collection
    )
```

That decision then, in the absence of any type knowledge, must be done absolutely carefully and conservative, as could see anything executing here.

That examples this code in `ExpressionBase` which every expression by default uses:

```
def computeExpressionLen(self, len_node, trace_collection):
    shape = self.getValueShape()

    has_len = shape.hasShapeSlotLen()

    if has_len is False:
        return makeRaiseTypeErrorExceptionReplacementFromTemplateAndValue(
            template="object of type '%s' has no len()",
            operation="len",
            original_node=len_node,
            value_node=self,
        )
    elif has_len is True:
```

```
        iter_length = self.getIterationLength()

        if iter_length is not None:
            from .ConstantRefNodes import makeConstantRefNode

            result = makeConstantRefNode(
                constant=int(iter_length),  # make sure to downcast long
                source_ref=len_node.getSourceReference(),
            )

            result = wrapExpressionWithNodeSideEffects(new_node=result, old_node=self)

            return (
                result,
                "new_constant",
                "Predicted 'len' result from value shape.",
            )

    self.onContentEscapes(trace_collection)

    # Any code could be run, note that.
    trace_collection.onControlFlowEscape(self)

    # Any exception may be raised.
    trace_collection.onExceptionRaiseExit(BaseException)

    return len_node, None, None
```

Notice how by default, known `__len__` but unpredictable or even unknown if a `__len__` slot is there, the code indicates that its contents and the control flow escapes (could change things behind out back) and any exception could happen.

Other expressions can know better, e.g. for compile time constants we can be a whole lot more certain:

```
def computeExpressionLen(self, len_node, trace_collection):
    return trace_collection.getCompileTimeComputationResult(
        node=len_node,
        computation=lambda: len(self.getCompileTimeConstant()),
        description="""Compile time constant len value pre-computed.""",
    )
```

In this case, we are using a function that will produce a concrete value or the exception that the `computation` function raised. In this case, we can let the Python interpreter that runs Nuitka do all the hard work. This lives in `CompileTimeConstantExpressionBase` and is the base for all kinds of constant values, or even built-in references like the name `len` itself and would be used in case of doing `len(len)` which obviously gives an exception.

Other overloads do not currently exist in Nuitka, but through the iteration length, most cases could be addressed, e.g. `list` nodes typical know their element counts.

The C side

When a slot is not optimized away at compile time however, we need to generate actual code for it. We figure out what this could be by looking at the original CPython implementation.

```
PyObject *builtin_len(PyObject *self, PyObject *v) {
    Py_ssize_t res;
```

```
    res = PyObject_Size(v);
    if (res < 0 && PyErr_Occurred())
        return NULL;
    return PyInt_FromSsize_t(res);
}
```

We find a pointer to `PyObject_Size` which is a generic Python C/API function used in the `builtin_len` implementation:

```
Py_ssize_t PyObject_Size(PyObject *o) {
    PySequenceMethods *m;

    if (o == NULL) {
        null_error();
        return -1;
    }

    m = o->ob_type->tp_as_sequence;
    if (m && m->sq_length)
        return m->sq_length(o);

    return PyMapping_Size(o);
}
```

On the C level, every Python object (the `PyObject *`) as a type named `ob_type` and most of its elements are slots. Sometimes they form a group, here `tp_as_sequence` and then it may or may not contain a function. This one is tried in preference. Then, if that fails, next up the mapping size is tried.

```
Py_ssize_t PyMapping_Size(PyObject *o) {
    PyMappingMethods *m;

    if (o == NULL) {
        null_error();
        return -1;
    }

    m = o->ob_type->tp_as_mapping;
    if (m && m->mp_length)
        return m->mp_length(o);

    type_error("object of type '%.200s' has no len()", o);
    return -1;
}
```

This is the same principle, except with `tp_as_mapping` and `mp_length` used.

So from this, we can tell how `len` gets at what could be a Python class `__len__` or other built-in types.

In principle, every slot needs to be dealt with in Nuitka, and it is assumed that currently all slots are supported on at least a very defensive level, to avoid unnoticed escapes of control flow.

Built-in call optimization

For calls to built-in names, there is typically a function in Python that delegates to the type constructor (e.g. when we talk about `int` that just creates an object passing the arguments of the call) or its own special implementation as we saw with the `len`.

For each built-in called, we have a specialized node, that presents to optimization the actions of the built-in. What are the impact, what are the results. We have seen the resulting example for `len` above, but how do we get there.

In Python, built-in names are used only if there is no module level variable of the name, and of course no local variable of that name.

Therefore, optimization of a built-in name is only done if it turns out the actually assigned in other code, and then when the call comes, arguments are checked and a relatively static node is created.

Code Generation towards C

Currently, Nuitka uses Pure C and no C++ patterns at all. The use of C11 requires on some platforms to compile the C11 using a C++ compiler, which works relatively well, but also limits the amount of C11 that can be used.

Exceptions

To handle and work with exceptions, every construct that can raise has either a `bool` or `int` return code or `PyObject *` with `NULL` return value. This is very much in line with that the Python C-API does.

Every helper function that contains code that might raise needs these variables. After a failed call, our variant of `PyErr_Fetch` called `FETCH_ERROR_OCCURRED` must be used to catch the defined error, unless some quick exception cases apply. The quick exception means, `NULL` return from C-API without a set exception means e.g. `StopIteration`.

As an optimization, functions that raise exceptions, but are known not to do so, for whatever reason, could only be asserted to not do so.

Statement Temporary Variables

For statements and larger constructs the context object track temporary values, that represent references. For some, these should be released at the end of the statement, or they represent a leak.

The larger scope temporary variables, are tracked in the function or module context, where they are supposed to have explicit `del` to release their references.

Local Variables Storage

Closure variables taken are to be released when the function object is later destroyed. For in-lined calls, variables are just passed, and it does not become an issue to release anything.

For function exit, owned variables, local or shared to other functions, must be released. This cannot be a `del` operation, as it also involves setting a value, which would be wrong for shared variables (and wasteful to local variables, as that would be its last usage). Therefore we need a special operation that simply releases the reference to the cell or object variable.

Exit Targets

Each error or other exit releases statement temporary values and then executes a `goto` to the exit target. These targets need to be setup. The `try`/`except` will e.g. catch error exits.

Other exits are `continue`, `break`, and `return` exits. They all work alike.

Generally, the exits stack of with constructs that need to register themselves for some exit types. A loop e.g. registers the `continue` exit, and a contained `try/finally` too, so it can execute the final code should it be needed.

Frames

Frames are containers for variable declarations and cleanups. As such, frames provide error exits and success exits, which remove the frame from the frame stack, and then proceed to the parent exit.

With the use of non `PyObject **` C types, but frame exception exits, the need to convert those types becomes apparent. Exceptions should still resolve the C version. When using different C types at frame exception exits, there is a need to trace the active type, so it can be used in the correct form.

Abortive Statements

The way `try/finally` is handled, copies of the `finally` block are made, and optimized independently for each abort method. The ones there are of course, `return`, `continue`, and `break`, but also implicit and explicit `raise` of an exception.

Code trailing an abortive statement can be discarded, and the control flow will follow these "exits".

Constant Preparation

Early versions of Nuitka, created all constants for the whole program for ready access to generated code, before the program launches. It did so in a single file, but that approach didn't scale well.

Problems were

- Even unused code contributed to start-up time, this can become a lot for large programs, especially in standalone mode.
- The massive amount of constant creation codes gave backend C compilers a much harder time than necessary to analyse it all at once.

The current approach is as follows. Code generation detects constants used in only one module, and declared `static` there, if the module is the only user, or `extern` if it is not. Some values are forced to be global, as they are used pre-main or in helpers.

These `extern` values are globally created before anything is used. The `static` values are created when the module is loaded, i.e. something did import it.

We trace used constants per module, and for nested ones, we also associate them. The global constants code is special in that it can only use `static` for nested values it exclusively uses, and has to export values that others use.

Language Conversions to make things simpler

There are some cases, where the Python language has things that can in fact be expressed in a simpler or more general way, and where we choose to do that at either tree building or optimization time.

The `assert` *statement*

The `assert` statement is a special statement in Python, allowed by the syntax. It has two forms, with and without a second argument. The later is probably less known, as is the fact that raise statements can have multiple arguments too.

The handling in Nuitka is:

```
assert value
# Absolutely the same as:
if not value:
    raise AssertionError
```

```
assert value, raise_arg
# Absolutely the same as:
if not value:
    raise AssertionError(raise_arg)
```

This makes assertions absolutely the same as a raise exception in a conditional statement.

This transformation is performed at tree building already, so Nuitka never knows about `assert` as an element and standard optimizations apply. If e.g. the truth value of the assertion can be predicted, the conditional statement will have the branch statically executed or removed.

The "comparison chain" expressions

In Nuitka we have the concept of an outline, and therefore we can make the following re-formulation instead:

```
a < b() > c < d


def _comparison_chain():  # So called "outline" function
    tmp_a = a
    tmp_b = b()

    tmp = tmp_a < tmp_b

    if not tmp:
        return tmp

    del tmp_a
    tmp_c = c

    tmp = tmp_b > tmp_c

    if not tmp:
        return tmp

    del tmp_b

    return tmp_c < d


_comparison_chain()
```

This transformation is performed at tree building already. The temporary variables keep the value for the use of the same expression. Only the last expression needs no temporary variable to keep it.

What we got from this, is making the checks of the comparison chain explicit and comparisons in Nuitka to be internally always about two operands only.

The `execfile` *built-in*

Handling is:

```
execfile(filename)
# Basically the same as:
exec(compile(open(filename).read()), filename, "exec")
```

> ***Note***
>
> This allows optimizations to discover the file opening nature easily and apply file embedding or whatever we will have there one day.

This transformation is performed when the `execfile` built-in is detected as such during optimization.

Generator expressions with `yield`

These are converted at tree building time into a generator function body that yields from the iterator given, which is the put into a for loop to iterate, created a lambda function of and then called with the first iterator.

That eliminates the generator expression for this case. It's a bizarre construct and with this trick needs no special code generation.

This is a complex example, demonstrating multiple cases of yield in unexpected cases:

```
x = ((yield i) for i in (1, 2) if not (yield))
# Basically the same as:
def x():
    for i in (1, 2):
        if not (yield):
            yield (yield i)
```

Function Decorators

When one learns about decorators, you see that:

```
@decorator
def function():
    pass


# Is basically the same as:
def function():
    pass


function = decorator(function)
```

The only difference is the assignment to function. In the `@decorator` case, if the decorator fails with an exception, the name `function` is not assigned yet, but kept in a temporary variable.

Therefore in Nuitka this assignment is more similar to that of a lambda expression, where the assignment to the name is only at the end, which also has the extra benefit of not treating real function and lambda functions any different.

This removes the need for optimization and code generation to support decorators at all. And it should make the two variants optimize equally well.

Functions nested arguments

Nested arguments are a Python2 only feature supported by Nuitka. Consider this example:

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

We solve this, by kind of wrapping the function with another function that does the unpacking and gives the errors that come from this:

```
def function(a, _1):
    def _tmp(a, b, c):
        return a, b, c

    a, b = _1
    return _tmp(a, b, c)
```

The `.1` is the variable name used by CPython internally, and actually works if you use keyword arguments via star dictionary. So this is very compatible and actually the right kind of re-formulation, but it removes the need from the code that does parameter parsing to deal with these.

Obviously, there is no frame for `_tmp`, just one for `function` and we do not use local variables, but temporary functions.

In-place Assignments

In-place assignments are re-formulated to an expression using temporary variables.

These are not as much a reformulation of += to +, but instead one which makes it explicit that the assign target may change its value.

```
a += b
```

```
_tmp = a.__iadd__(b)

if a is not _tmp:
    a = _tmp
```

Using `__iadd__` here to express that for the +, the in-place variant `iadd` is used instead. The `is` check may be optimized away depending on type and value knowledge later on.

Complex Assignments

Complex assignments are defined as those with multiple targets to assign from a single source and are re-formulated to such using a temporary variable and multiple simple assignments instead.

```
a = b = c
```

```
_tmp = c
a = _tmp
b = _tmp
del _tmp
```

This is possible, because in Python, if one assignment fails, it can just be interrupted, so in fact, they are sequential, and all that is required is to not calculate `c` twice, which the temporary variable takes care of. Were `b` a more complex expression, e.g. `b.some_attribute` that might raise an exception, `a` would still be assigned.

Unpacking Assignments

Unpacking assignments are re-formulated to use temporary variables as well.

```
a, b.attr, c[ind] = d = e, f, g = h()
```

Becomes this:

```
_tmp = h()

_iter1 = iter(_tmp)
_tmp1 = unpack(_iter1, 3)
_tmp2 = unpack(_iter1, 3)
_tmp3 = unpack(_iter1, 3)
unpack_check(_iter1)
a = _tmp1
b.attr = _tmp2
c[ind] = _tmp3
d = _tmp
_iter2 = iter(_tmp)
_tmp4 = unpack(_iter2, 3)
_tmp5 = unpack(_iter2, 3)
_tmp6 = unpack(_iter2, 3)
unpack_check(_iter1)
e = _tmp4
f = _tmp5
g = _tmp6
```

That way, the unpacking is decomposed into multiple simple statements. It will be the job of optimizations to try and remove unnecessary unpacking, in case e.g. the source is a known tuple or list creation.

Note

The `unpack` is a special node which is a form of `next` that will raise a `ValueError` when it cannot get the next value, rather than a `StopIteration`. The message text contains the number of values to unpack, therefore the integer argument.

> ***Note***
>
> The `unpack_check` is a special node that raises a `ValueError` exception if the iterator is not finished, i.e. there are more values to unpack. Again the number of values to unpack is provided to construct the error message.

With Statements

The `with` statements are re-formulated to use temporary variables as well. The taking and calling of `__enter__` and `__exit__` with arguments, is presented with standard operations instead. The promise to call `__exit__` is fulfilled by `try/except` clause instead.

```
with some_context as x:
    something(x)
```

```
tmp_source = some_context

# Actually it needs to be "special look-up" for Python2.7, so attribute
# look-up won't be exactly what is there.
tmp_exit = tmp_source.__exit__

# This one must be held for the whole with statement, it may be assigned
# or not, in our example it is. If an exception occurs when calling
# ``__enter__``, the ``__exit__`` should not be called.
tmp_enter_result = tmp_source.__enter__()

# Indicator variable to know if "tmp_exit" has been called.
tmp_indicator = False

try:
    # Now the assignment is to be done, if there is any name for the
    # manager given, this may become multiple assignment statements and
    # even unpacking ones.
    x = tmp_enter_result

    # Then the code of the "with" block.
    something(x)
except Exception:
    # Note: This part of the code must not set line numbers, which we
    # indicate with special source code references, which we call "internal".
    # Otherwise the line of the frame would get corrupted.

    tmp_indicator = True

    if not tmp_exit(*sys.exc_info()):
        raise
finally:
    if not tmp_indicator:
        # Call the exit if no exception occurred with all arguments
        # as "None".
        tmp_exit(None, None, None)
```

> ***Note***
>
> We don't refer really to `sys.exc_info()` at all, instead, we have fast references to the current exception type, value and trace, taken directly from the caught exception object on the C level.
>
> If we had the ability to optimize `sys.exc_info()` to do that, we could use the same transformation, but right now we don't have it.

For Loops

The `for` loops use normal assignments and handle the iterator that is implicit in the code explicitly.

```
for x, y in iterable:
    if something(x):
        break
else:
    otherwise()
```

This is roughly equivalent to the following code:

```
_iter = iter(iterable)
_no_break_indicator = False

while 1:
    try:
        _tmp_value = next(_iter)
    except StopIteration:
        # Set the indicator that the else branch may be executed.
        _no_break_indicator = True

        # Optimization should be able to tell that the else branch is run
        # only once.
        break

    # Normal assignment re-formulation applies to this assignment of course.
    x, y = _tmp_value
    del _tmp_value

    if something(x):
        break

if _no_break_indicator:
    otherwise()
```

> ***Note***
>
> The `_iter` temporary variable is of course also in a `try/finally` construct, to make sure it releases after its used. The `x, y` assignment is of course subject to unpacking re-formulation.
>
> The `try/except` is detected to allow to use a variant of `next` that does not raise an exception, but to be fast check about the `NULL` return from `next` built-in. So no actual exception handling is happening in this case.

While Loops

Quoting the `nuitka.tree.ReformulationWhileLoopStatements` documentation:

Reformulation of while loop statements.

Loops in Nuitka have no condition attached anymore, so while loops are re-formulated like this:

```
while condition:
    something()
```

```
while 1:
    if not condition:
        break

    something()
```

This is to totally remove the specialization of loops, with the condition moved to the loop body in an initial conditional statement, which contains a `break` statement.

That achieves, that only `break` statements exit the loop, and allow for optimization to remove always true loop conditions, without concerning code generation about it, and to detect such a situation, consider e.g. endless loops.

> ***Note***
>
> Loop analysis (not yet done) can then work on a reduced problem (which `break` statements are executed under what conditions) and is then automatically very general.
>
> The fact that the loop body may not be entered at all, is still optimized, but also in the general sense. Explicit breaks at the loop start and loop conditions are the same.

End quoting the `nuitka.tree.ReformulationWhileLoopStatements` documentation:

Exception Handlers

Exception handlers in Python may assign the caught exception value to a variable in the handler definition. And the different handlers are represented as conditional checks on the result of comparison operations.

```
try:
    block()
```

```
except A as e:
    handlerA(e)
except B as e:
    handlerB(e)
else:
    handlerElse()
```

```
try:
    block()
except:
    # These are special nodes that access the exception, and don't really
    # use the "sys" module.
    tmp_exc_type = sys.exc_info()[0]
    tmp_exc_value = sys.exc_info()[1]

    # exception_matches is a comparison operation, also a special node.
    if exception_matches(tmp_exc_type, (A,)):
        e = tmp_exc_value
        handlerA(e)
    elif exception_matches(tmp_exc_type, (B,)):
        e = tmp_exc_value
        handlerB(e)
    else:
        handlerElse()
```

For Python3, the assigned e variables get deleted at the end of the handler block. Should that value be already deleted, that `del` does not raise, therefore it's tolerant. This has to be done in any case, so for Python3 it is even more complex.

```
try:
    block()
except:
    # These are special nodes that access the exception, and don't really
    # use the "sys" module.
    tmp_exc_type = sys.exc_info()[0]
    tmp_exc_value = sys.exc_info()[1]

    # exception_matches is a comparison operation, also a special node.
    if exception_matches(tmp_exc_type, (A,)):
        try:
            e = tmp_exc_value
            handlerA(e)
        finally:
            del e
    elif exception_matches(tmp_exc_type, (B,)):
        try:
            e = tmp_exc_value
            handlerB(e)
        finally:
            del e
    else:
        handlerElse()
```

Should there be no `else:` branch, a default re-raise statement is used instead.

And of course, the values of the current exception type and value, both use special references, that access the C++ and don't go via `sys.exc_info` at all, nodes called `CaughtExceptionTypeRef` and `CaughtExceptionValueRef`.

This means, that the different handlers and their catching run time behavior are all explicit and reduced the branches.

Statement `try/except` *with* `else`

Much like `else` branches of loops, an indicator variable is used to indicate the entry into any of the exception handlers.

Therefore, the `else` becomes a real conditional statement in the node tree, checking the indicator variable and guarding the execution of the `else` branch.

Class Creation (Python2)

Classes in Python2 have a body that only serves to build the class dictionary and is a normal function otherwise. This is expressed with the following re-formulation:

```
# in module "SomeModule"
# ...


class SomeClass(SomeBase, AnotherBase):
    """ This is the class documentation. """

    some_member = 3
```

```
def _makeSomeClass():
    # The module name becomes a normal local variable too.
    __module__ = "SomeModule"

    # The doc string becomes a normal local variable.
    __doc__ = """ This is the class documentation. """

    some_member = 3

    return locals()

    # force locals to be a writable dictionary, will be optimized away, but
    # that property will stick. This is only to express, that locals(), where
    # used will be writable to.
    exec("")


SomeClass = make_class("SomeClass", (SomeBase, AnotherBase), _makeSomeClass())
```

That is roughly the same, except that `_makeSomeClass` is *not* visible to its child functions when it comes to closure taking, which we cannot express in Python language at all.

Therefore, class bodies are just special function bodies that create a dictionary for use in class creation. They don't really appear after the tree building stage anymore. The type inference will of course have to become able to understand `make_class` quite well, so it can recognize the created class again.

Class Creation (Python3)

In Python3, classes are a complicated way to write a function call, that can interact with its body. The body starts with a dictionary provided by the metaclass, so that is different, because it can `__prepare__` a non-empty locals for it, which is hidden away in "prepare_class_dict" below.

What's noteworthy, is that this dictionary, could e.g. be an `OrderDict`. I am not sure, what `__prepare__` is allowed to return.

```
# in module "SomeModule"
# ...

class SomeClass(SomeBase, AnotherBase, metaclass = SomeMetaClass):
    """ This is the class documentation. """

    some_member = 3
```

```
# Non-keyword arguments, need to be evaluated first.
tmp_bases = (SomeBase, AnotherBase)

# Keyword arguments go next, __metaclass__ is just one of them. In principle
# we need to forward the others as well, but this is ignored for the sake of
# brevity.
tmp_metaclass = select_metaclass(tmp_bases, SomeMetaClass)

tmp_prepared = tmp_metaclass.__prepare__("SomeClass", tmp_bases)

# The function that creates the class dictionary. Receives temporary variables
# to work with.
def _makeSomeClass():
    # This has effect, currently I don't know how to express that in Python3
    # syntax, but we will have a node that does that.
    locals().replace(tmp_prepared)

    # The module name becomes a normal local variable too.
    __module__ = "SomeModule"

    # The doc string becomes a normal local variable.
    __doc__ = """ This is the class documentation. """

    some_member = 3

    # Create the class, share the potential closure variable "__class__"
    # with others.
    __class__ = tmp_metaclass("SomeClass", tmp_bases, locals())

    return __class__


# Build and assign the class.
SomeClass = _makeSomeClass()
```

Generator Expressions

There are re-formulated as functions.

Generally they are turned into calls of function bodies with (potentially nested) for loops:

```
gen = (x * 2 for x in range(8) if cond())
```

```
def _gen_helper(__iterator):
    for x in __iterator:
        if cond():
            yield x * 2


gen = _gen_helper(range(8))
```

List Contractions

The list contractions of Python2 are different from those of Python3, in that they don't actually do any closure variable taking, and that no function object ever exists.

```
list_value = [x * 2 for x in range(8) if cond()]
```

```
def _listcontr_helper(__iterator):
    result = []

    for x in __iterator:
        if cond():
            result.append(x * 2)

    return result


list_value = _listcontr_helper(range(8))
```

The difference is that with Python3, the function "_listcontr_helper" is really there and named `<listcontraction>` (or `<listcomp>` as of Python3.7 or higher), whereas with Python2 the function is only an outline, so it can readily access the containing name space.

Set Contractions

The set contractions of Python2.7 are like list contractions in Python3, in that they produce an actual helper function:

```
set_value = {x * 2 for x in range(8) if cond()}
```

```
def _setcontr_helper(__iterator):
    result = set()

    for x in __iterator:
        if cond():
            result.add(x * 2)

    return result
```

```
set_value = _setcontr_helper(range(8))
```

Dictionary Contractions

The dictionary contractions of are like list contractions in Python3, in that they produce an actual helper function:

```
dict_value = {x: x * 2 for x in range(8) if cond()}
```

```
def _dictcontr_helper(__iterator):
    result = {}

    for x in __iterator:
        if cond():
            result[x] = x * 2

    return result


set_value = _dictcontr_helper(range(8))
```

Boolean expressions `and` *and* `or`

The short circuit operators `or` and `and` tend to be only less general that the `if`/`else` expressions, but have dedicated nodes. We used to have a re-formulation towards those, but we now do these via dedicated nodes too.

These new nodes, present the evaluation of the left value, checking for its truth value, and depending on it, to pick it, or use the right value.

Simple Calls

As seen below, even complex calls are simple calls. In simple calls of Python there is still some hidden semantic going on, that we expose.

```
func(arg1, arg2, named1=arg3, named2=arg4)
```

On the C-API level there is a tuple and dictionary built. This one is exposed:

```
func(*(arg1, arg2), **{"named1": arg3, "named2": arg4})
```

A called function will access this tuple and the dictionary to parse the arguments, once that is also re-formulated (argument parsing), it can then lead to simple in-lining. This way calls only have 2 arguments with constant semantics, that fits perfectly with the C-API where it is the same, so it is actually easier for code generation.

Although the above looks like a complex call, it actually is not. No checks are needed for the types of the star arguments and it's directly translated to `PyObject_Call`.

Complex Calls

The call operator in Python allows to provide arguments in 4 forms.

- Positional (or normal) arguments
- Named (or keyword) arguments
- Star list arguments
- Star dictionary arguments

The evaluation order is precisely that. An example would be:

```
something(pos1, pos2, name1=named1, name2=named2, *star_list, **star_dict)
```

The task here is that first all the arguments are evaluated, left to right, and then they are merged into only two, that is positional and named arguments only. for this, the star list argument and the star dictionary arguments, are merged with the positional and named arguments.

What's peculiar, is that if both the star list and dictionary arguments are present, the merging is first done for star dictionary, and only after that for the star list argument. This makes a difference, because in case of an error, the star argument raises first.

```
something(*1, **2)
```

This raises "TypeError: something() argument after ** must be a mapping, not int" as opposed to a possibly more expected "TypeError: something() argument after * must be a sequence, not int."

That doesn't matter much though, because the value is to be evaluated first anyway, and the check is only performed afterwards. If the star list argument calculation gives an error, this one is raised before checking the star dictionary argument.

So, what we do, is we convert complex calls by the way of special functions, which handle the dirty work for us. The optimization is then tasked to do the difficult stuff. Our example becomes this:

```
def _complex_call(called, pos, kw, star_list_arg, star_dict_arg):
    # Raises errors in case of duplicate arguments or tmp_star_dict not
    # being a mapping.
    tmp_merged_dict = merge_star_dict_arguments(
        called, tmp_named, mapping_check(called, tmp_star_dict)
    )

    # Raises an error if tmp_star_list is not a sequence.
    tmp_pos_merged = merge_pos_arguments(called, tmp_pos, tmp_star_list)

    # On the C-API level, this is what it looks like.
    return called(*tmp_pos_merged, **tmp_merged_dict)


returned = _complex_call(
    called=something,
    pos=(pos1, pos2),
    named={"name1": named1, "name2": named2},
    star_list_arg=star_list,
    star_dict_arg=star_dict,
)
```

The call to `_complex_call` is be a direct function call with no parameter parsing overhead. And the call in its end, is a special call operation, which relates to the `PyObject_Call` C-API.

Assignment Expressions

In Python 3.8 or higher, you assign inside expressions.

```
if (x := cond()):
   do_something()
```

this is the same as:

```
# Doesn't exist with that name, and it is not really taking closure variables,
# it just shares the execution context.
def _outline_func():
   nonlocal x
   x = cond()

   return x

if (_outline_func()):
   do_something
```

When we use this outline function, we are allowed statements, even assignments, in expressions. For optimization, they of course pose a challenge to be removed ever, only happens when it becomes only a return statement, but they do not cause much difficulties for code generation, since they are transparent.

Match Statements

In Python 3.10 or higher, you can write things like this:

```
match something():
    case [x] if x:
        z = 2
    case _ as y if y == x and y:
        z = 1
    case 0:
        z = 0
```

This is the same as

```
tmp_match_subject = something()

# Indicator variable, once true, all matching stops.
tmp_handled = False

# First branch
x = tmp_match_subject

if sequence_check(x)
   if x:
      z = 2
      tmp_handled = True

if tmp_handled is False:
   y = tmp_match_subject
```

```
   if x == y and y:
      z = 1
      tmp_handled = True

if tmp_handled is False:
   z = 0
```

Print Statements

The `print` statement exists only in Python2. It implicitly converts its arguments to strings before printing them. In order to make this accessible and compile time optimized, this is made visible in the node tree.

```
print arg1, "1", 1
```

This is in Nuitka converted so that the code generation for `print` doesn't do any conversions itself anymore and relies on the string nature of its input.

```
print str(arg1), "1", str(1)
```

Only string objects are spared from the `str` built-in wrapper, because that would only cause noise in optimization stage. Later optimization can then find it unnecessary for certain arguments.

Additionally, each `print` may have a target, and multiple arguments, which we break down as well for dumber code generation. The target is evaluated first and should be a file, kept referenced throughout the whole print statement.

```
print >> target_file, str(arg1), "1", str(1)
```

This is being reformulated to:

```
try:
   tmp_target = target_file

   print >>tmp_target, str(arg1), print >>tmp_target, "1", print
   >>tmp_target, str(1), print >>tmp_target

finally:
   del tmp_target
```

This allows code generation to not deal with arbitrary amount of arguments to `print`. It also separates the newline indicator from the rest of things, which makes sense too, having it as a special node, as it's behavior with regards to soft-space is different of course.

And finally, for `print` without a target, we still assume that a target was given, which would be `sys.stdout` in a rather hard-coded way (no variable look-ups involved).

Reformulations during Optimization

Builtin `zip` *for Python2*

```
def _zip(a, b, c):  # Potentially more arguments.
   # First assign, to preserve the order of execution, the arguments might be
   # complex expressions with side effects.
   tmp_arg1 = a
   tmp_arg2 = b
   tmp_arg3 = c
   # could be more
   ...

   # Creation of iterators goes first.
   try:
      tmp_iter_1 = iter(tmp_arg1)
   except TypeError:
      raise TypeError("zip argument #1 must support iteration")
   try:
      tmp_iter_2 = iter(tmp_arg2)
   except TypeError:
      raise TypeError("zip argument #2 must support iteration")
   try:
      tmp_iter_3 = iter(tmp_arg3)
   except TypeError:
      raise TypeError("zip argument #3 must support iteration")

   # could be more
   ...

   tmp_result = []
   try:
      while 1:
         tmp_result.append(
               (
                  next(tmp_iter_1),
                  next(tmp_iter_2),
                  next(tmp_iter_3),
                  # more arguments here ...
               )
         )
   except StopIteration:
      pass

   return tmp_result
```

Builtin `zip` *for Python3*

```
for x, y, z in zip(a, b, c):
    ...
```

```
def _zip_gen_object(a, b, c, ...):
    ...
    # See Python2
    ...

   # could be more
   ...
   while 1:
        yield (
            next(tmp_iter_1),
            next(tmp_iter_2),
            next(tmp_iter_3),
            ...
        )
    except StopIteration:
        break

for x, y, z in _zip_gen_object(a, b, c):
    ...
```

Builtin `map` *for Python2*

```
def _map():
    # TODO: Not done yet.
    pass
```

Builtin `min`

```
# TODO: keyfunc (Python2/3), defaults (Python3)
def _min(a, b, c):  # Potentially more arguments.
    tmp_arg1 = a
    tmp_arg2 = b
    tmp_arg3 = c
    # more arguments here ...

    result = tmp_arg1
    if keyfunc is None:  # can be decided during re-formulation
        tmp_key_result = keyfunc(result)
        tmp_key_candidate = keyfunc(tmp_arg2)
        if tmp_key_candidate < tmp_key_result:
            result = tmp_arg2
            tmp_key_result = tmp_key_candidate
        tmp_key_candidate = keyfunc(tmp_arg3)
        if tmp_key_candidate < tmp_key_result:
            result = tmp_arg3
            tmp_key_result = tmp_key_candidate
        # more arguments here ...
    else:
        if tmp_arg2 < result:
            result = tmp_arg2
        if tmp_arg3 < result:
            result = tmp_arg3
        # more arguments here ...

    return result
```

Builtin `max`

See `min` just with > instead of <.

Call to `dir` *without arguments*

This expression is reformulated to `locals().keys()` for Python2, and `list(locals.keys())` for Python3.

Calls to functions with known signatures

As a necessary step for inlining function calls, we need to change calls to variable references to function references.

```
def f(arg1, arg2):
    return some_op(arg1, arg2)


# ... other code

x = f(a, b + c)
```

In the optimization it is turned into

```
# ... other code

x = lambda arg1, arg2: some_op(arg1, arg2)(a, b + c)
```

> ***Note***
>
> The `lambda` stands here for a reference to the function, rather than a variable reference, this is the normal forward propagation of values, and does not imply duplicating or moving any code at all.

At this point, we still have not resolved the actual call arguments to the variable names, still a Python level function is created, and called, and arguments are parsed to a tuple, and from a tuple. For simplicity sake, we have left out keyword arguments out of the equation for now, but they are even more costly.

So now, what we want to do, is to re-formulate the call into what we call an outline body, which is a inline function, and that does the parameter parsing already and contains the function code too. In this inlining, there still is a function, but it's technically not a Python function anymore, just something that is an expression whose value is determined by control flow and the function call.

```
# ... other code


def _f():
    tmp_arg1 = arg1
    tmp_arg2 = b + c
    return tmp_arg1 + tmp_arg2


x = _f()
```

With this, a function is considered inlined, because it becomes part of the abstract execution, and the actual code is duplicated.

The point is, that matching the signature of the function to the actual arguments given, is pretty straight forward in many cases, but there are two forms of complications that can happen. One is default values, because they need to assigned or not, and the other is keyword arguments, because they allow to reorder arguments.

Let's consider an example with default values first.

```
def f(arg1, arg2=some_default()):
    return some_op(arg1, arg2)


# ... other code

x = f(a, b + c)
```

Since the point, at which defaults are taken, we must execute them at that point and make them available.

```
tmp_defaults = (some_default,)  # that was f.__defaults__

# ... other code


def _f():
    tmp_arg1 = arg1
    tmp_arg2 = tmp_defaults[0]
    return tmp_arg1 + tmp_arg2


x = _f()
```

Now, one where keyword arguments are ordered the other way.

```
def f(arg1, arg2):
    return some_op(arg1, arg2)


# ... other code

x = f(arg2=b + c, arg1=a)  # "b+c" is evaluated before "a"
```

The solution is an extra level of temporary variables. We remember the argument order by names and then assign parameters from it:

```
# ... other code


def _f():
    tmp_given_value1 = b + c
    tmp_given_value2 = a
    tmp_arg1 = tmp_given_value2
    tmp_arg2 = tmp_given_value1
    return tmp_arg1 + tmp_arg2


x = _f()
```

Obviously, optimization of Nuitka can decide, that e.g. should a or b+c not have side effects, to optimize these with standard variable tracing away.

Nodes that serve special purposes

Try statements

In Python, there is `try/except` and `try/finally`. In Nuitka there is only a `try`, which then has blocks to handle exceptions, `continue`, or `break`, or `return`. There is no `else` to this node type.

This is more low level and universal. Code for the different handlers can be different. User provided `finally` blocks become copied into the different handlers.

Releases

When a function exits, the local variables are to be released. The same applies to temporary variables used in re-formulations. These releases cause a reference to the object to the released, but no value change. They are typically the last use of the object in the function.

The are similar to `del`, but make no value change. For shared variables this effect is most visible.

Side Effects

When an exception is bound to occur, and this can be determined at compile time, Nuitka will not generate the code the leads to the exception, but directly just raise it. But not in all cases, this is the full thing.

Consider this code:

```
f(a(), 1 / 0)
```

The second argument will create a `ZeroDivisionError` exception, but before that `a()` must be executed, but the call to `f` will never happen and no code is needed for that, but the name look-up must still succeed. This then leads to code that is internally like this:

```
f(a(), raise_ZeroDivisionError())
```

which is then modeled as:

```
side_effect(a(), f, raise_ZeroDivisionError())
```

where we can consider `side_effect` to be a function that returns the last expression. Of course, if this is not part of another expression, but close to statement level, side effects, can be converted to multiple statements simply.

Another use case, is that the value of an expression can be predicted, but that the language still requires things to happen, consider this:

```
a = len((f(), g()))
```

We can tell that a will be 2, but the call to `f` and g must still be performed, so it becomes:

```
a = side_effects(f(), g(), 2)
```

Modelling side effects explicitly has the advantage of recognizing them easily and allowing to drop the call to the tuple building and checking its length, only to release it.

Caught Exception Type/Value References

When catching an exception, these are not directly put to `sys.exc_info()`, but remain as mere C variables. From there, they can be accessed with these nodes, or if published then from the thread state.

Hard Module Imports

These are module look-ups that don't depend on any local variable for the module to be looked up, but with hard-coded names. These may be the result of optimization gaining such level of certainty.

Currently they are used to represent `sys.stdout` usage for `print` statements, but other usages will follow.

Locals Dict Update Statement

For the `exec` re-formulation, we apply an explicit sync back to locals as an explicit node. It helps us to tell the affected local variable traces that they might be affected. It represents the bit of `exec` in Python2, that treats `None` as the locals argument as an indication to copy back.

Optimizing Attribute Lookups into Method Calls for Built-ins types

The attribute lookup node `ExpressionAttributeLookup` represents looking up an attribute name, that is known to be a string. That's already a bit more special, than say what `ExpressionBuiltinGetattr` does for `getattr`, where it could be any object being looked up. From the Python syntax however, these are what gets created, as it's not allowed in any other way. So, this is where this starts.

Then, when we are creating an attribute node with a *fixed* name, we dispatch it to generated node classes, e.g. `ExpressionAttributeLookupFixedAppend`. This will be the same, except that the attribute name is hardcoded.

There are generated, such that they can have code that is special for `.append` lookups. In their case, it makes sense to ask the source, if they are a `list` object exactly. It doesn't make sense to do this check for names that the `list` does not contain. So at that stage, we are saving both a bit of memory and time.

Should this question succeed, i.e. the expression the attribute values is looked up upon, is known to be a `list` exactly, we persist this knowledge in the also generated nodes that represent `list.append` and just that. It is called `ExpressionAttributeLookupListAppend` and only represents the knowledge gained so far.

We do not consider if `ExpressionAttributeLookupFixedAppend` is called, or not, passed as an argument, assigned somewhere, it doesn't matter yet, but for `ExpressionAttributeLookupListAppend` we know a hell of a lot more. We know its type, we know attributes for it, say `__name__`, as it is a compile time constant, therefore much optimization can follow for them, and code generation can specialize them too (not yet done).

Should these nodes then, and say this happens later after some inlining happens be seen as called, we can then turn them into method call nodes, checking the arguments and such, this is then `ExpressionListOperationAppend` and at this point, will raising errors with wrong argument counts.

And then we have this `ExpressionListOperationAppend` which will influence the tracing of `list` contents, i.e. it will be able to tell the `list` in question is no more empty after this `append`, and it will be able to at least predict the last element value, truth value of the list, etc.

Plan to add "ctypes" support

Add interfacing to C code, so Nuitka can turn a `ctypes` binding into an efficient binding as if it were written manually with Python C-API or better.

Goals/Allowances to the task

1. Goal: Must not directly use any pre-existing C/C++ language file headers, only generate declarations in generated C code ourselves. We would rather write or use tools that turn an existing a C header to some `ctypes` declarations if it needs to be, but not mix and use declarations from existing header code.

> ***Note***
>
> The "cffi" interface maybe won't have the issue, but it's not something we need to write or test the code for.

2. Allowance: May use `ctypes` module at compile time to ask things about `ctypes` and its types.
3. Goal: Should make use of `ctypes`, to e.g. not hard code in Nuitka what `ctypes.c_int()` gives on the current platform, unless there is a specific benefit.
4. Allowance: Not all `ctypes` usages must be supported immediately.
5. Goal: Try and be as general as possible.

   For the compiler, `ctypes` support should be hidden behind a generic interface of some sort. Supporting `math` module should be the same thing.

Type Inference - The Discussion

Main initial goal is to forward value knowledge. When you have `a = b`, that means that a and b now "alias". And if you know the value of b you can assume to know the value of a. This is called "aliasing".

When assigning a to something new, that won't change b at all. But when an attribute is set, a method called of it, that might impact the actual value, referenced by both. We need to understand mutable vs. immutable though, as some things are not affected by aliasing in any way.

```
a = 3
b = a

b += 4  # a is not changed

a = [3]
b = a

b += [4]  # a is changed indeed
```

If we cannot tell, we must assume that a might be changed. It's either b or what a was before. If the type is not mutable, we can assume the aliasing to be broken up, and if it is, we can assume both to be the same value still.

When that value is a compile time constant, we will want to push it forward, and we do that with "(Constant) Value Propagation", which is implemented already. We avoid too large constants, and we properly trace value assignments, but not yet aliases.

In order to fully benefit from type knowledge, the new type system must be able to be fully friends with existing built-in types, but for classes to also work with it, it should not be tied to them. The behavior of a type `long`, `str`, etc. ought to be implemented as far as possible with the built-in `long`, `str` at compiled time as well.

> ***Note***
>
> This "use the real thing" concept extends beyond builtin types, e.g. `ctypes.c_int()` should also be used, but we must be aware of platform dependencies. The maximum size of `ctypes.c_int` values would be an example of that. Of course that may not be possible for everything.
>
> This approach has well proven itself with built-in functions already, where we use real built-ins where possible to make computations. We have the problem though that built-ins may have problems to execute everything with reasonable compile time cost.

Another example, consider the following code:

```
len("a" * 1000000000000)
```

To predict this code, calculating it at compile time using constant operations, while feasible, puts an unacceptable burden on the compilation.

Esp. we wouldn't want to produce such a huge constant and stream it, the C++ code would become too huge. So, we need to stop the `*` operator from being used at compile time and cope with reduced knowledge, already here:

```
"a" * 100000000000000
```

Instead, we would probably say that for this expression:

- The result is a `str` or a C level `PyStringObject *`.
- We know its length exactly, it's `100000000000000`.
- Can predict every of its elements when sub-scripted, sliced, etc., if need be, with a function we may create.

Similar is true for this horrible (in Python2) thing:

```
range(100000000000000)
```

So it's a rather general problem, this time we know:

- The result is a `list` or C level `PyListObject *`.
- We know its length exactly, `100000000000000`.
- Can predict every of its elements when index, sliced, etc., if need be, with a function.

Again, we wouldn't want to create the list. Therefore Nuitka avoids executing these calculation, when they result in constants larger than a threshold of e.g. 256 elements. This concept has to be also applied to large integers and more CPU and memory traps.

Now lets look at a more complete use case:

```
for x in range(100000000000000):
    doSomething()
```

Looking at this example, one traditional way to look at it, would be to turn `range` into `xrange`, and to note that `x` is unused. That would already perform better. But really better is to notice that `range()` generated values are not used at all, but only the length of the expression matters.

And even if `x` were used, only the ability to predict the value from a function would be interesting, so we would use that computation function instead of having an iteration source. Being able to predict from a function could mean to have Python code to do it, as well as C code to do it. Then code for the loop can be generated without any CPython library usage at all.

> ***Note***
>
> Of course, it would only make sense where such calculations are "O(1)" complexity, i.e. do not require recursion like "n!" does.

The other thing is that CPython appears to at - run time - take length hints from objects for some operations, and there it would help too, to track length of objects, and provide it, to outside code.

Back to the original example:

```
len("a" * 1000000000000)
```

The theme here, is that when we can't compute all intermediate expressions, and we sure can't do it in the general case. But we can still, predict some of properties of an expression result, more or less.

Here we have `len` to look at an argument that we know the size of. Great. We need to ask if there are any side effects, and if there are, we need to maintain them of course. This is already done by existing optimization if an operation generates an exception.

> ***Note***
>
> The optimization of `len` has been implemented and works for all kinds of container creation and ranges.

Applying this to "ctypes"

The *not so specific* problem to be solved to understand `ctypes` declarations is maybe as follows:

```
import ctypes
```

This leads to Nuitka in its tree to have an assignment from a `__import__` expression to the variable `ctypes`. It can be predicted by default to be a module object, and even better, it can be known as `ctypes` from standard library with more or less certainty. See the section about "Importing".

So that part is "easy", and it's what will happen. During optimization, when the module `__import__` expression is examined, it should say:

- `ctypes` is a module
- `ctypes` is from standard library (if it is, might not be true)
- `ctypes` then has code behind it, called `ModuleFriend` that knows things about it attributes, that should be asked.

The later is the generic interface, and the optimization should connect the two, of course via package and module full names. It will need a `ModuleFriendRegistry`, from which it can be pulled. It would be nice if

we can avoid `ctypes` to be loaded into Nuitka unless necessary, so these need to be more like a plug-in, loaded only if necessary, i.e. the user code actually uses `ctypes`.

Coming back to the original expression, it also contains an assignment expression, because it re-formulated to be more like this:

```
ctypes = __import__("ctypes")
```

The assigned to object, simply gets the type inferred propagated as part of an SSA form. Ideally, we could be sure that nothing in the program changes the variable, and therefore have only one version of that variable.

For module variables, when the execution leaves the module to unknown code, or unclear code, it might change the variable. Therefore, likely we will often only assume that it could still be `ctypes`, but also something else.

Depending on how well we control module variable assignment, we can decide this more of less quickly. With "compiled modules" types, the expectation is that it's merely a quick C == comparison check. The module friend should offer code to allow a check if it applies, for uncertain cases.

Then when we come to uses of it:

```
ctypes.c_int()
```

At this point, using SSA, we are more of less sure, that `ctypes` is at that point the module, and that we know what it's `c_int` attribute is, at compile time, and what it's call result is. We will use the module friend to help with that. It will attach knowledge about the result of that expression during the SSA collection process.

This is more like a value forward propagation than anything else. In fact, constant propagation should only be the special case of it, and one design goal of Nuitka was always to cover these two cases with the same code.

Excursion to Functions

In order to decide what this means to functions and their call boundaries, if we propagate forward, how to handle this:

```
def my_append(a, b):
    a.append(b)

    return a
```

We annotate that a is first a "unknown but defined parameter object", then later on something that definitely has an `append` attribute, when returned, as otherwise an exception occurs.

The type of a changes to that after `a.append` look-up succeeds. It might be many kinds of an object, but e.g. it could have a higher probability of being a `PyListObject`. And we would know it cannot be a `PyStringObject`, as that one has no `append` method, and would have raised an exception therefore.

> ***Note***
>
> If classes, i.e. other types in the program, have an `append` attribute, it should play a role too, there needs to be a way to plug-in to this decisions.

> ***Note***
>
> On the other hand, types without `append` attribute can be eliminated.

Therefore, functions through SSA provide an automatic analysis on their return state, or return value types, or a quick way to predict return value properties, based on input value knowledge.

So this could work:

```
b = my_append([], 3)

assert b == [3]  # Could be decided now
```

Goal: The structure we use makes it easy to tell what `my_append` may be. So, there should be a means to ask it about call results with given type/value information. We need to be able to tell, if evaluating `my_append` makes sense with given parameters or not, if it does impact the return value.

We should e.g. be able to make `my_append` tell, one or more of these:

- Returns the first parameter value as return value (unless it raises an exception).
- The return value has the same type as a (unless it raises an exception).
- The return value has an `append` attribute.
- The return value might be a `list` object.
- The return value may not be a `str` object.
- The function will raise if first argument has no `append` attribute.

The exactness of statements may vary. But some things may be more interesting. If e.g. the aliasing of a parameter value to the return value is known exactly, then information about it need to all be given up, but some can survive.

It would be nice, if `my_append` had sufficient information, so we could specialize with `list` and `int` from the parameters, and then e.g. know at least some things that it does in that case. Such specialization would have to be decided if it makes sense. In the alternative, it could be done for each variant anyway, as there won't be that many of them.

Doing this "forward" analysis appears to be best suited for functions and therefore long term. We will try it that way.

Excursion to Loops

```
a = 1

while 1:  # think loop: here
    b = a + 1
    a = b

    if cond():
        break

print(a)
```

The handling of loops (both `for` and `while` are re-formulated to this kind of loops with `break` statements) has its own problem. The loop start and may have an assumption from before it started, that a is constant, but that is only true for the first iteration. So, we can't pass knowledge from outside loop forward directly into the for loop body.

So the collection for loops needs to be two pass for loops. First, to collect assignments, and merge these into the start state, before entering the loop body. The need to make two passes is special to loops.

For a start, it is done like this. At loop entry, all pre-existing, but written traces, are turned into loop merges. Knowledge is not completely removed about everything assigned or changed in the loop, but then it's not trusted anymore.

From that basis, the `break` exits are analysed, and merged, building up the post loop state, and `continue` exits of the loop replacing the unknown part of the loop entry state. The loop end is considered a `continue` for this purpose.

Excursion to Conditions

```
if cond:
    x = 1
else:
    x = 2

b = x < 3
```

The above code contains a condition, and these have the problem, that when exiting the conditional block, a merge must be done, of the `x` versions. It could be either one. The merge may trace the condition under which a choice is taken. That way, we could decide pairs of traces under the same condition.

These merges of SSA variable "versions", represent alternative values. They pose difficulties, and might have to be reduced to commonality. In the above example, the `<` operator will have to check for each version, and then to decide that both indeed give the same result.

The trace collection tracks variable changes in conditional branches, and then merges the existing state at conditional statement exits.

> ***Note***
>
> A branch is considered "exiting" if it is not abortive. Should it end in a `raise`, `break`, `continue`, or `return`, there is no need to merge that branch, as execution of that branch is terminated.
>
> Should both branches be abortive, that makes things really simple, as there is no need to even continue.
>
> Should only one branch exist, but be abortive, then no merge is needed, and the collection can assume after the conditional statement, that the branch was not taken, and continue.

When exiting both the branches, these branches must both be merged, with their new information.

In the above case:

- The "yes" branch knows variable `x` is an `int` of constant value `1`
- The "no" branch knows variable `x` is an `int` of constant value `2`

That might be collapsed to:

- The variable `x` is an integer of value in `(1,2)`

Given this, we then should be able to pre-compute the value of this:

```
b = x < 3
```

The comparison operator can therefore decide and tell:

- The variable `b` is a boolean of constant value `True`.

Were it unable to decide, it would still be able to say:

- The variable `b` is a boolean.

For conditional statements optimization, it's also noteworthy, that the condition is known to pass or not pass the truth check, inside branches, and in the case of non-exiting single branches, after the statement it's not true.

We may want to take advantage of it. Consider e.g.

```
if type(a) is list:
    a.append(x)
else:
    a += (x,)
```

In this case, the knowledge that `a` is a list, could be used to generate better code and with the definite knowledge that `a` is of type list. With that knowledge the `append` attribute call will become the `list` built-in type operation.

Excursion to `return` statements

The `return` statement (like `break`, `continue`, `raise`) is "aborting" to control flow. It is always the last statement of inspected block. When there statements to follow it, optimization will remove it as "dead code".

If all branches of a conditional statement are "aborting", the statement is decided "aborting" too. If a loop doesn't abort with a break, it should be considered "aborting" too.

Excursion to `yield` expressions

The `yield` expression can be treated like a normal function call, and as such invalidates some known constraints just as much as they do. It executes outside code for an unknown amount of time, and then returns, with little about the outside world known anymore, if it's accessible from there.

Mixed Types

Consider the following inside a function or module:

```
if cond is not None:
    a = [x for x in something() if cond(x)]
else:
    a = ()
```

A programmer will often not make a difference between `list` and `tuple`. In fact, using a `tuple` is a good way to express that something won't be changed later, as these are mutable.

Note

Better programming style, would be to use this:

```
if cond is not None:
    a = tuple(x for x in something() if cond(x))
else:
    a = ()
```

People don't do it, because they dislike the performance hit encountered by the generator expression being used to initialize the tuple. But it would be more consistent, and so Nuitka is using it, and of course one day Nuitka ought to be able to make no difference in performance for it.

To Nuitka though this means, that if `cond` is not predictable, after the conditional statement we may either have a `tuple` or a `list` type object in `a`. In order to represent that without resorting to "I know nothing about it", we need a kind of `min`/`max` operating mechanism that is capable of say what is common with multiple alternative values.

Note

At this time, we don't really have that mechanism to find the commonality between values.

Back to "ctypes"

```
v = ctypes.c_int()
```

Coming back to this example, we needed to propagate `ctypes`, then we can propagate "something" from `ctypes.int` and then known what this gives with a call and no arguments, so the walk of the nodes, and diverse operations should be addressed by a module friend.

In case a module friend doesn't know what to do, it needs to say so by default. This should be enforced by a base class and give a warning or note.

Now to the interface

The following is the intended interface:

- Iteration with node methods `computeStatement` and `computeExpression`.

  These traverse modules and functions (i.e. scopes) and visit everything in the order that Python executes it. The visiting object is `TraceCollection` and pass forward. Some node types, e.g. `StatementConditional` new create branch trace collections and handle the SSA merging at exit.

- Replacing nodes during the visit.

  Both `computeStatement` and `computeExpression` are tasked to return potential replacements of themselves, together with "tags" (meaningless now), and a "message", used for verbose tracing.

  The replacement node of + operator, may e.g. be the pre-computed constant result, wrapped in side effects of the node, or the expression raised, again wrapped in side effects.

- Assignments and references affect SSA.

  The SSA tree is initialized every time a scope is visited. Then during traversal, traces are built up. Every assignment and merge starts a new trace for that matter. References to a given variable version are traced that way.

- Value escapes are traced too.

  When an operation hands over a value to outside code, it indicates so to the trace collection. This is for it to know, when e.g. a constant value, might be mutated meanwhile.

- Nodes can be queried about their properties.

  There is a type shape and a value shape that each node can be asked about. The type shape offers methods that allow to check if certain operations are at all supported or not. These can always return `True` (yes), `False` (no), and `None` (cannot decide). In the case of the later, optimizations may not be able do much about it. Lets call these values "tri-state".

  There is also the value shape of a node. This can go deeper, and be more specific to a given node.

  The default implementation will be very pessimistic. Specific node types and shapes may then declare, that they e.g. have no side effects, will not raise for certain operations, have a known truth value, have a known iteration length, can predict their iteration values, etc.

- Nodes are linked to certain states.

  During the collect, a variable reference, is linked to a certain trace state, and that can be used by parent operations.

  ```
  a = 1
  b = a + a
  ```

  In this example, the references to `a`, can look-up the `1` in the trace, and base value shape response to `+` on it. For compile time evaluation, it may also ask `isCompileTimeConstant()` and if both nodes will respond `True`, then "getCompileTimeConstant()" will return `1`, which will be be used in computation.

  Then `extractSideEffects()` for the a reference will return `()` and therefore, the result `2` will not be wrapped.

  An alternative approach would be `hasTypeSlotAdd()` on the both nodes, and they both do, to see if the selection mechanism used by CPython can be used to find which types `+` should be used.

- Class for module import expression `ExpressionImportModule`.

  This one just knows that something is imported, but not how or what it is assigned to. It will be able in a recursive compile, to provide the module as an assignment source, or the module variables or submodules as an attribute source when referenced from a variable trace or in an expression.

- Base class for module friend `ModuleFriendBase`.

  This is intended to provide something to overload, which e.g. can handle `math` in a better way.

- Module `ModuleFriendRegistry`

  Provides a register function with `name` and instances of `ValueFriendModuleBase` to be registered. Recursed to modules should integrate with that too. The registry could well be done with a metaclass approach.

- The module friends should each live in a module of their own.

With a naming policy to be determined. These modules should add themselves via above mechanism to `ModuleFriendRegistry` and all shall be imported and register. Importing of e.g. `ctypes` should be delayed to when the friend is actually used. A meta class should aid this task.

The delay will avoid unnecessary blot of the compiler at run time, if no such module is used. For "qt" and other complex stuff, this will be a must.

- The walk should initially be single pass, and not maintain history.

  Instead optimization that needs to look at multiple things, e.g. "unused assignment", will look at the whole SSA collection afterwards.

Discussing with examples

The following examples:

```
# Assignment, the source decides the type of the assigned expression
a = b

# Operator "attribute look-up", the looked up expression "ctypes" decides
# via its trace.
ctypes.c_int

# Call operator, the called expressions decides with help of arguments,
# which have been walked, before the call itself.
called_expression_of_any_complexity()

# import gives a module any case, and the "ModuleRegistry" may say more.
import ctypes

# From import need not give module, "x" decides what it is.
from x import y

# Operations are decided by arguments, and CPython operator rules between
# argument states.
a + b
```

The optimization is mostly performed by walking of the tree and performing trace collection. When it encounters assignments and references to them, it considers current state of traces and uses it for `computeExpression`.

> ***Note***
>
> Assignments to attributes, indexes, slices, etc. will also need to follow the flow of `append`, so it cannot escape attention that a list may be modified. Usages of `append` that we cannot be sure about, must be traced to exist, and disallow the list to be considered known value again.

Code Generation Impact

Right now, code generation assumes that everything is a `PyObject *`, i.e. a Python object, and does not take knowledge of `int` or other types into consideration at all, and it should remain like that for some time to come.

Instead, `ctypes` value friend will be asked give `Identifiers`, like other codes do too. And these need to be able to convert themselves to objects to work with the other things.

But Code Generation should no longer require that operations must be performed on that level. Imagine e.g. the following calls:

```
c_call(other_c_call())
```

Value returned by "other_c_call()" of say `c_int` type, should be possible to be fed directly into another call. That should be easy by having a `asIntC()` in the identifier classes, which the `ctypes` Identifiers handle without conversions.

Code Generation should one day also become able to tell that all uses of a variable have only `c_int` value, and use `int` instead of `PyObjectLocalVariable` more or less directly. We could consider `PyIntLocalVariable` of similar complexity as `int` after the C++ compiler performed its in-lining.

Such decisions would be prepared by finalization, which then would track the history of values throughout a function or part of it.

Initial Implementation

The basic interface will be added to *all* expressions and a node may override it, potentially using trace collection state, as attached during `computeExpression`.

Goal 1 (Reached)

Initially most things will only be able to give up on about anything. And it will be little more than a tool to do simple look-ups in a general form. It will then be the first goal to turn the following code into better performing one:

```
a = 3
b = 7
c = a / b
print(c)
```

to:

```
a = 3
b = 7
c = 3 / 7
print(c)
```

and then:

```
a = 3
b = 7
c = 0
print(c)
```

and then:

```
a = 3
b = 7
```

```
c = 0
print(0)
```

This depends on SSA form to be able to tell us the values of a, b, and c to be written to by constants, which can be forward propagated at no cost.

Goal 2 (Reached)

The assignments to a, b, and c shall all become prey to "unused" assignment analysis in the next step. They are all only assigned to, and the assignment source has no effect, so they can be simply dropped.

```
print(0)
```

In the SSA form, these are then assignments without references. These assignments, can be removed if the assignment source has no side effect. Or at least they could be made "anonymous", i.e. use a temporary variable instead of the named one. That would have to take into account though, that the old version still needs a release.

The most general form would first merely remove assignments that have no impact, and leave the value as a side effect, so we arrive at this first:

```
3
7
0
print(0)
```

When applying the removal of expression only statements without effect, this gives us:

```
print(0)
```

which is the perfect result. Doing it in one step would only be an optimization at the cost of generalization.

In order to be able to manipulate nodes related to a variable trace, we need to attach the nodes that did it. Consider this:

```
if cond():
    x = 1
elif other():
    x = 3

# Not using "x".
print(0)
```

In the above case, the merge of the value traces, should say that x may be undefined, or one of 1 or 3, but since x is not used, apply the "dead value" trick to each branch.

The removal of the "merge" of the 3 x versions, should exhibit that the other versions are also only assigned to, and can be removed. These merges of course appear as usages of the x versions.

Goal 3

Then third goal is to understand all of this:

```
def f():
    a = []

    print(a)

    for i in range(1000):
        print(a)

        a.append(i)

    return len(a)
```

> ***Note***
>
> There are many operations in this, and all of them should be properly handled, or at least ignored in safe way.

The first goal code gave us that the `list` has an annotation from the assignment of `[]` and that it will be copied to a until the for loop in encountered. Then it must be removed, because the `for` loop somehow says so.

The a may change its value, due to the unknown attribute look-up of it already, not even the call. The for loop must be able to say "may change value" due to that, of course also due to the call of that attribute too.

The code should therefore become equivalent to:

```
def f():
    a = []

    print([])

    for i in range(1000):
        print(a)

        a.append(i)

    return len(a)
```

But no other changes must occur, especially not to the `return` statement, it must not assume a to be constant "[]" but an unknown a instead.

With that, we would handle this code correctly and have some form constant value propagation in place, handle loops at least correctly, and while it is not much, it is important demonstration of the concept.

Goal 4

The fourth goal is to understand the following:

```
def f(cond):
    y = 3
```

```
    if cond:
        x = 1
    else:
        x = 2

    return x < y
```

In this we have a branch, and we will be required to keep track of both the branches separately, and then to merge with the original knowledge. After the conditional statement we will know that "x" is an "int" with possible values in `(1,2)`, which can be used to predict that the return value is always `True`.

The forth goal will therefore be that the "ValueFriendConstantList" knows that append changes a value, but it remains a list, and that the size increases by one. It should provide an other value friend "ValueFriendList" for "a" due to that.

In order to do that, such code must be considered:

```
a = []

a.append(1)
a.append(2)

print(len(a))
```

It will be good, if `len` still knows that a is a list object, but not the constant list anymore.

From here, work should be done to demonstrate the correctness of it with the basic tests applied to discover undetected issues.

Fifth and optional goal: Extra bonus points for being able to track and predict `append` to update the constant list in a known way. Using `list.append` that should be done and lead to a constant result of `len` being used.

The sixth and challenging goal will be to make the code generation be impacted by the value friends types. It should have a knowledge that `PyList_Append` does the job of append and use `PyList_Size` for `len`. The "ValueFriends" should aid the code generation too.

Last and right now optional goal will be to make `range` have a value friend, that can interact with iteration of the for loop, and `append` of the `list` value friend, so it knows it's possible to iterate 5000 times, and that "a" has then after the "loop" this size, so `len(a)` could be predicted. For during the loop, about a the range of its length should be known to be less than 5000. That would make the code of goal 2 completely analyzed at compile time.

Limitations for now

- Aim only for limited examples. For `ctypes` that means to compile time evaluate:

  ```
  print(ctypes.c_int(17) + ctypes.c_long(19))
  ```

  Later then call to "libc" or something else universally available, e.g. "strlen()" or "strcmp()" from full blown declarations of the callable.

- We won't have the ability to test that optimization are actually performed, we will check the generated code by hand.

With time, we will add XML based checks with "xpath" queries, expressed as hints, but that is some work that will be based on this work here. The "hints" fits into the "ValueFriends" concept nicely or so the hope is.

- No inter-function optimization functions yet

  Of course, once in place, it will make the `ctypes` annotation even more usable. Using `ctypes` objects inside functions, while creating them on the module level, is therefore not immediately going to work.

- No loops yet

  Loops break value propagation. For the `ctypes` use case, this won't be much of a difficulty. Due to the strangeness of the task, it should be tackled later on at a higher priority.

- Not too much.

  Try and get simple things to work now. We shall see, what kinds of constraints really make the most sense. Understanding `list` subscript/slice values e.g. is not strictly useful for much code and should not block us.

> ***Note***
>
> This design is not likely to be the final one.

How to make Features Experimental

Every experimental feature needs a name. We have a rule to pick a name with lower case and _ as separators. An example of with would be the name `jinja_generated_add` that has been used in the past.

Command Line

Experimental features are enabled with the command line argument

```
nuitka --experimental=jinja_generated_add ...
```

In C code

In Scons, all experimental features automatically are converted into C defines, and can be used like this:

```
#ifdef _NUITKA_EXPERIMENTAL_JINJA_GENERATED_ADD
#include "HelpersOperationGeneratedBinaryAdd.c"
#else
#include "HelpersOperationBinaryAdd.c"
#endif
```

The C pre-processor is the only thing that makes an experimental feature usable.

In Python

You can query experimental features using `Options.isExperimental()` with e.g. code like this:

```
if Options.isExperimental("use_feature"):
    experimental_code()
else:
    standard_code()
```

When to use it

Often we need to keep feature in parallel because they are not finished, or need to be tested after merge and should not break. Then we can do code changes that will not make a difference except when the experimental flag is given on the command line to Nuitka.

The testing of Nuitka is very heavy weight when e.g. all Python code is compiled, and very often, it is interesting to compare behavior with and without a change.

When to remove it

When a feature becomes default, we might choose to keep the old variant around, but normally we do not. Then we remove the `if` and `#if` checks and drop the old code.

At this time, large scale testing will have demonstrated the viability of the code.

Adding dependencies to Nuitka

First of all, there is an important distinction to make, runtime or development time. The first kind of dependency is used when Nuitka is executing.

Adding a Runtime Dependency

This is the kind of dependency that is the most scrutinized. As we want Nuitka to run on latest greatest Python as well as relatively old ones, we have to be very careful with these ones.

There is also a distinction of optional dependencies. Right now e.g. the `lxml` package is relatively optional, and Nuitka can work without it being installed, because e.g. on some platforms it will not be easy to do so. That bar has lifted somewhat, but it means e.g. that XML based optimization tests are not run with all Python versions.

The list of runtime dependencies is in `requirements.txt` and it is for those the case, that they are not really required to be installed by the user, consider this snippet:

```
# Folders to use for cache files.
appdirs

# Scons is the backend building tool to turn C files to binaries.
scons
```

For both these dependencies, there is either an inline copy (Scons) that we handle to use in case, if Scons is not available (in fact we have a version that works with Python 2.6 and 2.7 still), and also the same for appdirs and every dependency.

But since inline copies are against the rules on some platforms that still do not contain the package, we often even have our own wrapper which provides a minimal fallback or exposes a sane interface for the subset of functionality that we use.

> ***Note***
>
> Therefore, please if you consider adding one of these, get in touch with `@Nuitka-pushers` first and get a green light.

Adding a Development Dependency

A typical example of a development dependency is `black` which is used by our autoformat tool, and then in turn by the git pre-commit hook. It is used to format source code, and doesn't have a role at run time of the actual compiler code of Nuitka.

Much less strict rules apply to these in comparison to runtime dependencies. Generally please take care that the tool must be well maintained an available on newer Pythons. Then we can use it, no problem normally. But if it's really big, say all of SciPy, we might want to justify it a bit better.

The list of development dependencies is in `requirements-devel.txt` and it is for example like this:

```
# Autoformat needs this
rstfmt == 0.0.10 ; python_version >= '3.7'
```

We always add the version, so that when tests run on as old versions as Python 2.6, the installation would fail with that version, so we need to make a version requirement. Sometimes we use older versions for Python2 than for Python3, `Jinaj2` being a notable candidate, but generally we ought to avoid that. For many tools only being available for currently 3.7 or higher is good enough, esp. if they are run as development tools, like `autoformat-nuitka-source` is.

Idea Bin

This an area where to drop random ideas on our minds, to later sort it out, and out it into action, which could be code changes, plan changes, issues created, etc.

- Make "SELECT_METACLASS" meta class selection transparent.

  Looking at the "SELECT_METACLASS" it should become an anonymous helper function. In that way, the optimization process can remove choices at compile time, and e.g. in-line the effect of a meta class, if it is known.

  This of course makes most sense, if we have the optimizations in place that will allow this to actually happen.

- Keeping track of iterations

  The trace collection trace should become the place, where variables or values track their use state. The iterator should keep track of the "next()" calls made to it, so it can tell which value to given in that case.

  That would solve the "iteration of constants" as a side effect and it would allow to tell that they can be removed.

  That would mean to go back in the tree and modify it long after.

  ```
  a = iter((2, 3))
  b = next(a)
  c = next(a)
  del a
  ```

It would be sweet if we could recognize that as:

```
a = iter((2, 3))
b = side_effect(next(a), 2)
c = side_effect(next(a), 3)
del a
```

That trivially becomes:

```
a = iter((2, 3))
next(a)
b = 2
next(a)
c = 3
del a
```

When the `del a` is examined at the end of scope, or due to another assignment to the same variable, ending the trace, we would have to consider of the `next` uses, and retrofit the information that they had no effect.

```
a = iter((2, 3))
b = 2
b = 3
del a
```

- Aliasing

  Each time an assignment is made, an alias is created. A value may have different names.

```
a = iter(range(9))
b = a
c = next(b)
d = next(a)
```

  If we fail to detect the aliasing nature, we will calculate d wrongly. We may incref and decref values to trace it.

  Aliasing is automatically traced already in SSA form. The b is assigned to version of a. So, that should allow to replace it with this:

```
a = iter(range(9))
c = next(a)
d = next(a)
```

  Which then will be properly handled.

- Tail recursion optimization.

  Functions that return the results of calls, can be optimized. The Stackless Python does it already.

- Integrate with "upx" compression.

  Calling "upx" on the created binaries, would be easy.

- In-lining constant "exec" and "eval".

  It should be possible to re-formulate at least cases without "locals" or "globals" given.

```
def f():
    a = 1
    b = 2

    exec("""a+=b;c=1""")

    return a, c
```

  Should become this here:

```
def f():
    a = 1
    b = 2

    a += b  #
    c = 1  # MaybeLocalVariables for everything except known local ones.

    return a, c
```

  If this holds up, inlining `exec` should be relatively easy.

- Original and overloaded built-ins

  This is about making things visible in the node tree. In Nuitka things that are not visible in the node tree tend to be wrong. We already pushed around information to the node tree a lot.

  Later versions, Nuitka will become able to determine it has to be the original built-in at compile time, then a condition that checks will be optimized away, together with the slow path. Or the other path, if it won't be. Then it will be optimized away, or if doubt exists, it will be correct. That is the goal.

  Right now, the change would mean to effectively disable all built-in call optimization, which is why we don't immediately do it.

  Making the compatible version, will also require a full listing of all built-ins, which is typing work merely, but not needed now. And a way to stop built-in optimization from optimizing built-in calls that it used in a wrap. Probably just some flag to indicate it when it visits it to skip it. That's for later.

  But should we have that both, I figure, we could not raise a `RuntimeError` error, but just do the correct thing, in all cases. An earlier step may raise `RuntimeError` error, when built-in module values are written to, that we don't support.

Prongs of Action

In this chapter, we keep track of prongs of action currently ongoing. This can get detailed and shows things we strive for.

Builtin optimization

Definitely want to get built-in names under full control, so that variable references to module variables do not have a twofold role. Currently they reference the module variable and also the potential built-in as a fallback.

In terms of generated code size and complexity for modules with many variables and uses of them that is horrible. But `some_var` (normally) cannot be a built-in and therefore needs no code to check for that each time.

This is also critical to getting to whole program optimization. Being certain what is what there on module level, will enable more definitely knowledge about data flows and module interfaces.

Class Creation Overhead Reduction

This is more of a meta goal. Some work for the metaclass has already been done, but that is Python2 only currently. Being able to to decide built-ins and to distinguish between global only variables, and built-ins more clearly will help this a lot.

In the end, empty classes should be able to be statically converted to calls to `type` with static dictionaries. The inlining of class creation function is also needed for this, but on Python3 cannot happen yet.

Memory Usage at Compile Time

We will need to store more and more information in the future. Getting the tree to be tight shaped is therefore an effort, where we will be spending time too.

The mix-ins prevent slots usage, so lets try and get rid of those. The "children having" should become more simple and faster code. I am even thinking of even generating code in the meta class, so it's both optimal and doesn't need that mix-in any more. This is going to be ugly then.

Coverage Testing

And then there is coverage, it should be taken and merged from all Python versions and OSes, but I never managed to merge between Windows and Linux for unknown reasons.

Python3 Performance

The Python3 lock for thread state is making it slower by a lot. I have only experimental code that just ignores the lock, but it likely only works on Linux, and I wonder why there is that lock in the first place.

Ignoring the locks cannot be good. But what updates that thread state pointer ever without a thread change, and is this what ABI flags are about in this context, are there some that allow us to ignore the locks.

An important bit would be to use a thread state once acquired for as much as possible, currently exception helpers do not accept it as an argument, but that ought to become an option, that way saving and restoring an exception will be much faster, not to mention checking and dropping non interesting, or rewriting exceptions.

Caching of Python level compilation

While the C compilation result is already cached with *ccache* and friends now, we need to also cover our bases and save the resulting node tree of potential expensive optimization on the module level.

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 source under: https://github.com/Nuitka/Nuitka/blob/develop/Developer_Manual.rst

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