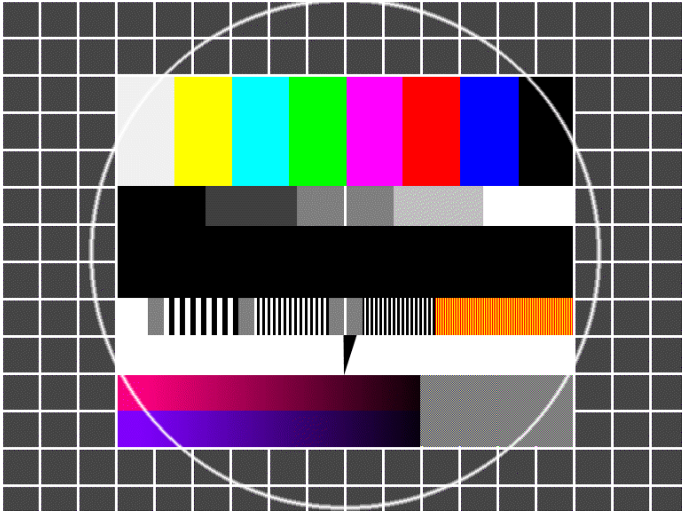
Contents



Topics (1 of 3)

- Goals for a true Python Compiler
  - Only faster than before, no new language
  - Impose no language limits
  - Same error messages
  - All extension modules should work
- Evaluated Alternatives
  - PyPy / RPython (limited language)
  - PyPy / JIT (JIT only)
  - Pyrex / Cython (different language than Python, incompatible)

Topics (2 of 3)

- Problematic Differences C++ and Python
  - Operators `or` und `and` do not match
  - No `try` .. `finally` in C++
  - Evaluation of function call arguments
  - Strings

Topics (3 of 3)

- Lessons Learned
    - Surprises
    - Reformulations
- Nuitka the Project
    - The `git flow`
    - Project Plan
    - Status
    - Join
    - License

Goals for a true Python Compiler (1 of 2)

- Pure Python, just faster

  Special language constructs are a "No go", also annotation in a separate file should not be done.

- Better code, even where performance matters

  The Python-Compiler shall make programmers writer better code, not worse

  1. It shouldn't be necessary to avoid named constants, only because those are slower.
  2. Functions calls should be written in the most readable way, no the most CPython run time efficient way.

Goals for a true Python Compiler (2 of 2)

- Report problems at run time at compile time.

  When a variable is read, that has never been set to a value, this is noticed in Python only late. That costs development time, test time, a whole lot.

  The compiler should assume the role of PyLint largely, and eventually achieve even more than it.

- No Limits

  Everything that Python has, the compiler must do it too. No matter how much it hurts. And Python allows a lot.

- Original error messages
  For syntax errors, type errors for function calls, etc. as far as possible the original error messages. Of course, new warnings will be welcome.

Goal: Pure Python - only faster

- Adding Types to Python

1. If a variable has the type `cint`, how will overflow work? As in C, or not at all as in Python?
2. If a variable has the type `char \*`, is that string then mutable? As in C, or not at all, as in Python?
3. What happens during conversions, Exceptions raise them, which?
4. A whole new semantics is bad. 🚫

   Instead the Python Compiler should recognize, if a performance gain is possible, and either at compile time or run time use code that handles overflows so the result will be same as in CPython.

Goal: Pure Python - only faster (1 of 4)

- Typeannotations in a separate file

1. Only effective when executed by the compiler. Pity! 🚫
2. Indeed, statements like "must be an integer" are important information.

   These improve the code quality generally, they are often like assertions, and as such should *not* be external. ✓
3. Maintaining two files has its own difficulties. I personally am happy to get along with a single one.

Goal: Pure Python - only faster (2 of 4)

- Who wants to learn a new language.

  Certainly not all of us.

  Some of us surely, learning or even inventing new languages can be exciting. I invented a language 30 years ago.

  Some do it out of necessity, that pure Python currently it not as fast, with the exception of PyPy JIT maybe, but that one doesn't do everything as well.

  The compiler should liberate these people.

Goal: Pure Python - only faster (3 of 4)

- New language means loosing all the tools

1. IDE auto completion (emacs python-mode, Eclipse, etc.) ☹
2. IDE syntax highlighting ☹
3. PyLint checks ☹
4. Dependency graphs ☹
5. No simple fallback to CPython, Jython, PyPy, IronPython, etc. ☹

That hurts, I don't want to/can't live without these. 🚫

Goal: Pure Python - only faster (4 of 4)

- Proposed Solution without new language

  A module `hints` to contain checks implemented in Python, assertions, etc.

  ```
  x = hints.mustbeint( x )
  ```

  The compiler recognizes these hints and `x` in C++ may become `int x` or `PyObjectOrInt`.

  Ideally, these hints will be recognized by inlining and understanding `mustbeint` consequences, that follow as well from this:

  ```
  x = int( x )
  ```

Goal: Code is not Performance Victim (1 of 5)

```
meters_per_nautical_mile = 1852

def convertMetersToNauticalMiles( meters ):
    return meters / meters_per_nautical_mile

def convertNauticalMilesToMeters( miles ):
    return miles * meters_per_nautical_mile
```

```
def convertMetersToNauticalMiles( meters ):
    return meters / 1852

def convertNauticalMilesToMeters( miles ):
    return miles * 1852
```

Nuitka

Goal: Code is not Performance Victim (2 of 5)

```
def someFunction():
    len = len

    # some len using code follows
```

- Another evil code optimization ☹
- Copying globale variables into local namespace makes things less readable, and less readable. But without it, `len()` is much slower to call.

Goal: Code is not Performance Victim (3 of 5)

```
return Generator.getFunctionCallCode(
    function_identifier   = function_identifier,
    argument_tuple        = positional_args_identifier,
    argument_dictionary   = kw_identifier,
    star_list_identifier = star_list_identifier,
    star_dict_identifier = star_dict_identifier,
)
```

```
return Generator.getFunctionCallCode(
    function_identifier, argument_tuple, kw_identifier, star_list_identifier, star_dict_identifier
)
```

Keyword arguments are extremly costly. For each call, a new dictionary is created and passed, where the called function needs to make tests per argument name, extra argument, argument name types, etc.

Goal: Code is not Performance Victim (4 of 5)

- Performance Optimization for CPython almost always means:
  1. Less readable code than necessary ☹
  2. Worse design than necessary ☹
- All of these, a compiler can handle better

  Without changing the source code, except possibly to add extra checks.

Nuitka

Goal: Code is not Performance Victim (5 of 5)

- Performance optimizations in source code need to no longer have a point
  1. The developer shouldn't write `int` where it's clear by looking at the source
  2. Manual inlining, copying identifiers to local scope, etc. shall become unnecessary
- We are into Python for the readable code, aren't we? So to us `fast` is just a feature, that somebody else should provide.
- That also gives my reason as for why *I* work on Nuitka.

Discarded Alternative: PyPy - RPython (1 of 2)

- I have developed patches for PyPy/RPython:

  These patches were accepted, all of them trivial stuff that came up during a trial usage of RPython. The community of PyPy was very friendly and cooperative. Everything worked via IRC channel and patches. I liked it a lot. 😉

- Experimented with RPython

  Rewriting the performance critical parts in RPython was possible and partially was fun. Little things could be added easily, but ...

Discarded Alternative: PyPy - RPython (2 of 2)

- A reduced Compiler

  Unfortunately RPython means "reduced" Python.

  That forced design changes in the application that I didn't like, and ultimately made me not persue that path.

- That way, I can't reach my goals

Discarded Alternative: PyPy - JIT

- But it's not a compiler

  A JIT never really knows, how far to look, what to consider, without creating too much pointless overhead. Therefore a JIT must constrain itself in its analysis.

- Too complex

  The design is really, really impressive, but that also means it's too complex.

- Many goals are reachable only partially

  Whatever the JIT recognizes, that is solved, what not, is not solved. 🚫

Discarded Alternative: Pyrex / Cython

- I have developed patches for Cython:

  These were not accepted, because they would have moved Cython away from Pyrex, which was important at the time. ☹

- And:

  The direction of the project was totally in mismatch. A lot of developers and users have productive code in a language that is absolutely not Python.

  The main goal of Cython is in my humble opinion to connect Python modules to C++, i.e. bindings. Whole Python programs appears to not be a major goal. Also, optimizable by hand is the top priority.

- Ultimately that means no overlap in goals. 🚫

Nuitka - Tailored to Goals

- Created explicitly to achieve all stated goals
- No time pressure, need not be fast immediately

  Can do things the *correct* /TM/ way, no stop gap is needed
- Named after my wife Anna

  Anna - Annuitka - Nuitka

Target Language

The decision for C++11 is ultimately:

- against portability (gcc only at the time, clang not, no MSVC yet)
- against language knowledge

All of these are important drawbacks, yet for C++11 spoke easier code generation:

- variadic templates (helped initially)
- raw strings

With C++03 that would have required Boost, which also achieves a lot of C++11. But then there are still things like "raw strings", who save a lot of work.

Generated Code (1 of 2)

```
print "& (3)", a & b & d
```

```
PRINT_ITEM_TO( NULL, _python_str_digest_2c9fbb02f98767c025af8ac4a1461a18 );
PRINT_ITEM_TO( NULL,
   PyObjectTemporary(
      BINARY_OPERATION(
         PyNumber_And,
         PyObjectTemporary(
            BINARY_OPERATION(
               PyNumber_And, _mvar___main___a.asObject0(), _mvar___main___b.asObject0()
            )
         ).asObject(),
         _mvar___main___d.asObject0()
      )
   ).asObject()
);
PRINT_NEW_LINE_TO( NULL );
```

Generated Code (2 of 2)

An important design choice for generated code, was to avoid having to manage temporary `PyObject \*` within code. Instead, Python expressions, should translated to C++ expressions. Otherwise generated code would have to handle release.

To aid it, we have `PyObjectTemporary` and its destructor.

The string `&` containing, cannot become a C++ identifier, therefore a hash code is used. A string `"value"` would become `_python_str_plain_value`.

The `BINARY_OPERATION` is a wrapper for the CPython C-API, that throws a C++ Exception, should an error be indicated (`NULL` return).

Within Nuitka generated C++ return codes are not checked, in error case, a C++ exception is raised. That allows the C++ compile to manage the release of `PyObjectTemporary` or `PyObjectLocalVariable`, `PyObjectSharedVariable` references.

Python to C++ gap (1 of 10)

Boolean operators in Python are actually sections:

```
a or b # Really either "a" or "b" as that value.
```

```
a || b // Won't work, is "true" or "false"
```

Solution without temporary variables. The GNU extension `?:` :

```
SELECT_IF_TRUE( _mvar___main___a.asObject() ) ?: _mvar___main___b.asObject()
```

Python to C++ gap (2 of 10)

The `?:` operator is "short-circuit", that means, the right hand side, is only evaluated, when `SELECT_IF_TRUE` didn't return `NULL`. This way, the behavior of Python can be achieved.

Using a temporary variable, it *could* be done this like:

```
(tmp_object = SELECT_IF_TRUE( _mvar___main___a.asObject() )) ? (tmp_object): _mvar___main___b.asObject()
```

It's not done yet, as there is currently *no* C++11 compiler that doesn't support the GNU extension.

The ternary operator according to standard has a sequence point, that makes sure the assignment is executed, before the values of `?` are allowed to be used.

Python to C++ gap (3 of 10)

Comparison chains don't work in C++:

```
f() < g() < h() # Calls "g()" only once
```

We need to re-write that, to non-Python as described in Developer Manual:

```
# With "temp variables" and "assignment expressions", absolutely the same as:
f() < ( tmp_g = g() ) and tmp_g < ( tmp_h = h() )
```

Assignment expressions are supported by C++ and the `||` and `&&` operators are sequence points to these assignments according to C++ standard.

Python to C++ gap (4 of 10)

In C++ there is no `try` .. `finally` construct:

The C++ gods are so convinced of the advantages of RAII (Resource Acquisition is Initialization), they don't offer it. They surely have their reasons, which are of no interest here.

So `try` .. `finally`` needs to be emulated. This is done by catching exceptions, saving them to a variable, executing the finally code, and the re-throwing the saved exception.

Python to C++ gap (5 of 10)

Complications occur because of `break`, `continue`, and `return`:

```
while something():
   try:
      needs_cleanup()

      if some_condition():
         break
      elif other_condition():
         continue
      else:
         return result
   finally:
      cleanup()
```

Python to C++ gap (6 of 10)

The call to `cleanup` is happening in all runs of the loop.

In fact, break, continue and return, need to be treated just like exceptions, therefore, they are implemented as C++ exceptions, dependent from where they occur:

Inside `try` .. `finally`:

```
throw BreakException()
```

Outside:

```
break
```

Unfortunately, g++ is not (yet?) clever enough to avoid the exception at run time.

Python to C++ gap (7 of 10)

Function Calls:

In Python the order of evaluation of parameters is *guaranteed*. In C++ it is *not in any way*:

```
# Python calls a, b, c, then f, in that exact order
f( a(), b(), c() )
```

```
// C++ has undefined evaluation order, may call a, b, c in any order
f( a(), b(), c() );
```

Python to C++ gap (8 of 10)

Function calls, continued:

ARM and Intel have a difference with g++, as does "clang" on Intel:

- left to right (ARM, registers for parameters)
- right to left (Intel, stack for parameters)

In C++ there is no general solution. By clever use of Macros, one can define every function is a way, that makes sure parameters are evaluated, in the compiler specific way:

```
#define RICH_COMPARE_LT( operand1, operand2 ) _RICH_COMPARE_LT( EVAL_ORDERED_2( operand1, operand2 ) )

PyObject *_RICH_COMPARE_LT( EVAL_ORDERED_2( PyObject *operand1, PyObject *operand2 ) );
```

Python to C++ gap (9 of 10)

Strings:

Python has very elegant raw strings. With them, you can include practically every "blob" in the code:

```
r'Anything goes here'
```

C++11 also has "raw strings":

```
R"'raw(Anything goes here)raw"
```

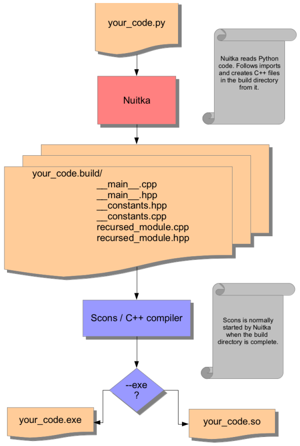
But for them to work properly, at least with g++, such code is needed:

Python to C++ gap (10 of 10)

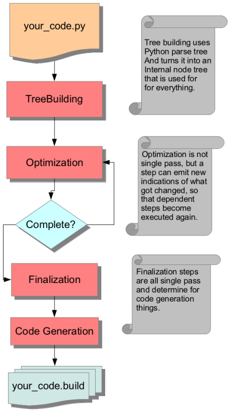
```
def decide( match ):
    if match.group(0) == "\n":
        return end + r' "\n" ' + start
    elif match.group(0) == "\r":
        return end + r' "\r" ' + start
    elif match.group(0) == "\0":
        return end + r' "\0" ' + start
    elif match.group(0) == "??":
        return end + r' "??" ' + start
    else:
        return end + r' "\\" ' + start

result = re.sub( "\n|\r|\0|\\\\|\\?\\?", decide, result )
```

Nuitka Design - Outside View



Nuitka Design - Inside View



Tree building uses Python parse tree And turns it into an Internal node tree that is used for for everything.

Optimization is not single pass, but a step can emit new indications of what got changed, so that dependent steps become executed again.

Finalization steps are all single pass and determine for code generation things.

Nuitka the Project - git flow (1 of 2)

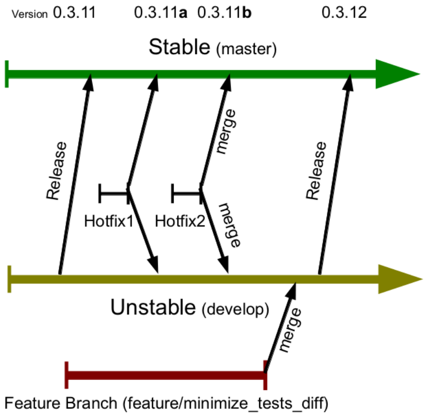
- First used for release 0.3.11, up to current release 0.3.23.
- Stable

  The stable version should be perfect at all times and is fully supported. As soon as bugs are known and have fixes, hotfix releases containing only these fixes will be done.
- Develop

  Future possible release, that is supposed to be fully correct, but it isn't supported as much, and can at times have problems or inconsistencies.
- Feature Branches

  Here, longer taking, single topic developments, that are not yet finished, are made public. Need not work at all.

Nuitka the Project - git flow (2 of 2)



Nuitka the Project - Plan (1 of 2)

1. Feature Parity with CPython

Understand the whole language and be fully compatible.

2. Generate efficient C++ code

With just using `PyObject *` implement the behaviour efficiently, and achieve a speed gain from that already.

3. "Constant Propagation"

Identify as much values and constraints at compile time. And on that basis, generate even more efficient code.

Nuitka the Project - Plan (2 of 2)

4. "Type Inference"

Detect and special case `str`, `int`, `list`, etc. in the program.

5. Interfacing with C code.

Nuitka should become able to recognize and understand `ctypes` and `cffi` bindings to the point, where it can avoid using ctypes, and make direct calls and accesses, based on thos declarations.

6. `hints` Module

The tool for the developer to provide extra information to Nuitka and CPython, e.g. that a parameter value has to be an `int` value.

Nuitka the Project - Status (1 of 5)

1. Feature Parity with CPython

Achieved. ✓

The frame stack, pickling of compiled functions, compatibility is a solved problem.

2. Generate efficient C++ code

The pystone benchmark gives a nice speedup by *258%*. ✓

Bei pybench the gain is often `inf`, that means, Nuitka takes no measurable time, or the factors are simply too high. ✓

Only exceptions are not yet fast enough. These are slow in C++, and should be avoided more often. Here, there we have more work to do.

Nuitka the Project - Status (2 of 5)

3. "Constant Propagation"

Largely achieved. ✓

4. "Type Inference"

Only starting to exist. 🚫

Note

Because it has to be reliable, we cannot borrow code from PyLint, which is allowed more optimistic assumptions.

Nuitka das Projekt - Status (3 of 5)

5. Interface to C code

Does not exist yet. 🚫

The inclusion of C-Headers and syntax is a taboo.

Vision for Nuitka, it should be possible, to generate direct calls and accesses from declarations of `ctypes` module.

That would be the base of portable bindings, that just work everywhere, and that these - using Nuitka - would be possible to become extremely fast.

Nuitka the Project - Status (4 of 5)

6. hints Module

Does not yet exist. 🚫

Should check under CPython and raise errors, just like under Nuitka. Ideally, the code simply allows Nuitka to detect, what they do, and make conclusions based on that, which may be too ambitious though.

It would be great, if there was found common ground with other projects.

Nuitka the Project - Status (5 of 5)

- Nuitka is known to work under:
  - Linux on x86/x64
  - Linux on ARM
  - Crosscompile to Windows
  - Windows native using MinGW
- Nuitka needs:
  - Python 2.6 or 2.7, 3.2 experimental
  - g++ 4.5, g++ 4.6, g++ 4.7 or clang 3.0

Nuitka the Project - Activities

Current:

- Type Inference
- Speedcenter revival on http://speedcenter.nuitka.net
- CPython2.7 tests also a as git repository with documented commits per diff
- XML based Regression Tests

Maybe this year:

- Making direct calls to known functions, removing argument parsing inside programs
- Go through CPython3.2 tests pass, what is missing
- ShedSkin tests to compare with

Nuitka - XML Dump

Python:

```
tryBreakFinallyTest()
```

Quote from XML-Dump:

```
<node line="27" kind="StatementExpressionOnly">
  <role name="expression">
    <node line="27" kind="ExpressionCall">
      <role name="called">
        <node variable="ModuleVariableReference to ModuleVariable 'tryBreakFinallyTest' of '__main__'"
        line="27" kind="ExpressionVariableRef" name="tryBreakFinallyTest"/>
      </role>
      <role name="positional_args"/>
      <role name="pairs"/>
      <role name="list_star_arg"/>
      <role name="dict_star_arg"/>
    </node>
  </role>
</node>
```

Nuitka - Lessions Learned

```
assert type(a) is float
a == a                           # how can this be possible False
```

Equality cannot be assumed even if a `is a` is true.

```
[ (x,y) for x in range(3) for y in range(4) ]
```

Nested comprehensions. Never saw them before, probably for a reason.

```
a = yield( value )
```

Never realized that `yield` has turned into an expression that could become not `None`. Didn't see much code doing that though.

Nuitka

Language Conversions to make things simpler

- There are cases, where Python language 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.
- These simplifications are very important for optimisation. Releases in the last months have mainly been about these.

Conversion: 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 handling in Nuitka is:

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

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

Conversion: 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 ) )
```

Conversion: The "comparison chain" expressions

```
a < b > c < d
# With "temp variables" and "assignment expressions", absolutely the same as:
a < ( tmp_b = b ) and tmp_b > ( tmp_c = c ) and ( tmp_c < d )
```

- This transformation is performed at tree building already.
- The temporary variables keep the value for the potential read in the same expression.
- The syntax is not Python, and only pseudo language to expression the internal structure of the node tree after the transformation.

Conversion: The "execfile" builtin

Handling is:

```
execfile( filename )
```

Basically the same as:

```
exec( compile( open( filename ).read() ), filename, "exec" )
```

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

Conversion: 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.

Important for Nuitka, it never sees decorator as special.

Conversion: Inplace Assignments

Inplace 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
```

Conversion: 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
b = _tmp
a = _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.

Conversion: 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 statementy. 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.

Conversion: 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 lookup" for Python2.7, so attribute lookup 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__()
```

```python
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.

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

Conversion: For Loops

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

```
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 True:
    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()
```

Conversion: While Loops

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

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

```python
while True:
    if not condition:
        break

    something()
```

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

Conversion: Exception Handler Values

Exception handlers may assign the caught exception value in the handler definition.

```
try:
    something()
except Exception as e:
    handle_it()
```

That is equivalent to the following:

```
try:
    something()
except Exception:
    e = sys.exc_info()[1]
    handle_it()
```

Of course, the value of the current exception, use special references for assignments, that access the C++ and don't go via "sys.exc_info" at all, these are called "CaughtExceptionValueRef".

Conversion: try.. except else branches

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.

Conversion: Classes Creation

Classes 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:

```
class SomeClass(SomeBase,AnotherBase)
    some_member = 3
```

```
def _makeSomeClass:
    some_member = 3

    return locals()

    # force locals to be a writable dictionary, will be optimized away, but that
    # property will stick.
    exec ""

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

Nuitka the Project - Join

You are *welcome*. 😀

Am accepting patches as ...

- whatever `diff -ru` outputs
- git formatted "patch queues"
- git pull requests

The integration work is *mine*. Based on git branches `master` or `develop`, or source archive, no matter, I will integrate your work and attribute it to you.

There is the mailing list nuitka-dev on which most of the announcements will be done too. Also there are RSS Feeds on http://nuitka.net, where you will be kept up to date.

Nuitka

Nuitka the Project - License

Starting now, I have released Nuitka under Apache License 2.0.

- Very liberal license
- Allows Nuitka to be used with practically all software

Discussion

- Will be here for all of PyCON-EU, and *welcome* questions and ideas in person. Questions also welcome via Email to kayhayen@gmx.de or on the mailing list.
- My hope is:

  1. Some exchange with PyPy developers, maybe I can use the *PyPy Tests* as well, the test runner appears to be specific.
  2. A *critical* Review of Nuitka design and source code, would be great.
  3. Ideas from C++ people, how Nuitka could produce better code.

This Presentation

- Created with rst2pdf
- Download the PDF http://nuitka.net/pr/Nuitka-Presentation-PyCON-EU-2012.pdf
- Diagrams created with OOo Draw
- Icons from visualpharm.com (License requires link)
- For presentation on PyCon EU