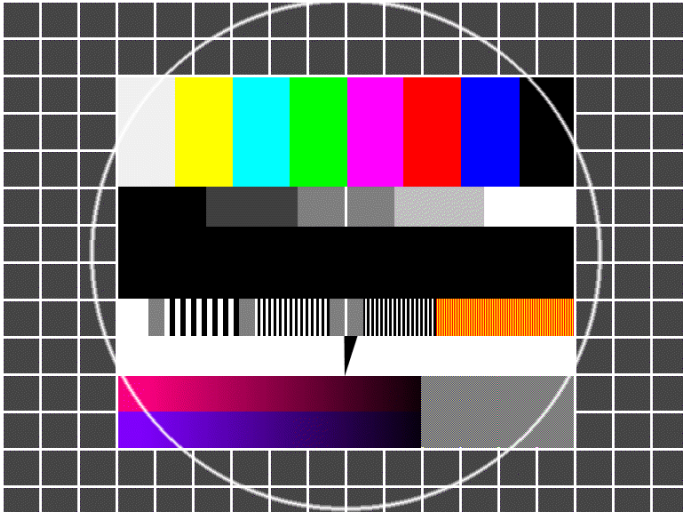
Contents

Nuitka

Nuitka

Nuitka

Nuitka



Topics (1 of 6)

- Intro
  - Who am I? / What is Nuitka?
  - System Requirements
- Demo
  - Compiling a simple program
  - Compiling a simple module
  - Compiling full blown program (Mercurial)
  - Follow the design in the source

Topics (2 of 6)

- Nuitka goals Quick run through
  - Faster than before, no new language
  - No language or compatibility limits
  - Same error messages
  - All extension modules work

Topics (3 of 6)

- Status Update
  - Project Plan
  - All Python versions supported
  - All major platforms supported
  - Threading is there
  - Join me

Topics (4 of 6)

- Reformulations
  - Python in simpler Python
  - New lessons learned
- But - what if
  - Changes to builtins module
  - Values escape in Python

Topics (5 of 6)

- Optimization (so far)
  - Peephole visitor
  - Dead code elimination
  - Frame stack avoidance

Topics (6 of 6)

- Optimization (coming)
  - Trace collection
  - Merges in the trace
  - Dead assignment elimination
  - Escape analysis
  - Shape analysis

Intro - Nuitka (1 of 3)

- Created explicitly to achieve a fully compatible Python compiler that does not invent a new language, and opens up whole new usages.
- Thinking out of box. Python is not *only* for scripting, do the other things with Python too.
- 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

Intro - Nuitka (2 of 3)

- Major milestones achieved, basically working as an accelerator.
- Nuitka is known to work under:
  - Linux, NetBSD, FreeBSD
  - Windows 32 and 64 bits
  - MacOS X
  - Crosscompile from Linux to Windows
- Android and iOS need work, but should be possible

Nuitka - Intro (3 of 3)

Starting Europython last year, Nuitka was released under Apache License 2.0.

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

System Requirements

- Nuitka needs:
  - Python 2.6 or 2.7, 3.2, or 3.3 (new)
  - C++ compiler:
    - g++ 4.5 or higher
    - clang 3.0, 3.2, or 3.3
    - Visual Studio 2008, Visual Studio 2010
    - MinGW for Win32
  - Your Python code 😉
  - That is it ✔

Demo Time (1 of 3)

Simple program

```
def somefunc( a ):
   b = 7
   return a + b

def somegenerator( x ):
   yield 1
   yield x

if __name__ == "__main__":
    print somefunc( 8 )
    print list( somegenerator( 42 ) )
```

Demo Time (2 of 3)

Simple module, same as before.

- Generated code doesn't change much, but `__name__` does, and the conditional code guarded by a test on it is optimized away.

Demo Time (3 of 3)

Complex program.

- Compile without recursion gives warnings about "mercurial" module
- Compile with embedded module `--recurse-to=mercurial`.
- Compile with plugins found as well.

  Using `--recurse-directory=/usr/share/pyshared/hgext`.

Generated Code (1 of 2)

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

```
PyObjectTempKeeper0 op1;
PyObjectTempKeeper1 op3;
PRINT_ITEM_TO( NULL, _python_str_digest_2c9fbb02f98767c025af8ac4a1461a18 ); //                                    #
PRINT_ITEM_TO( NULL,
   PyObjectTemporary(
     TO_STR(
        PyObjectTemporary(
           ( op3.assign(
              ( op1.assign( _mvar___main___a.asObject0() ),
                BINARY_OPERATION(
                   PyNumber_And,
                   op1.asObject0(),
                   _mvar___main___b.asObject0()
                )
              )
             ),
             BINARY_OPERATION(
                PyNumber_And,
                op3.asObject0(),
                _mvar___main___d.asObject0()
             )
           )
        ).asObject()
     )
   ).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.

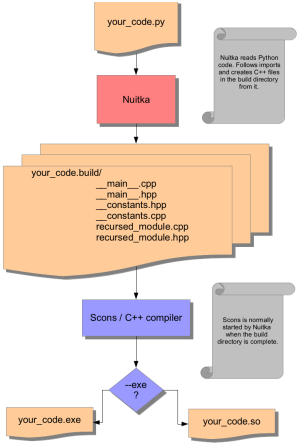
To aid it, we have `PyObjectTemporary`, `PyObjectTempKeeper` and their destructors.

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

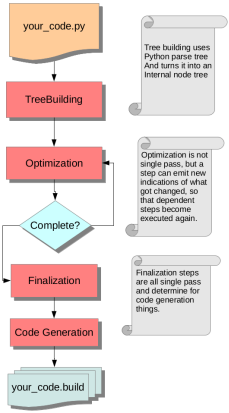
The `BINARY_OPERATION` is a wrapper for the CPython C-API, that throws a C++ Exception, should an error be indicated (`NULL` return).

Within generated C++ code return codes are not checked, a C++ exception would be raised. That allows C++ compiler to manage the release of references in `PyObjectTemporary` or `PyObjectLocalVariable`, `PyObjectSharedVariable`.

Nuitka Design - Outside View



Nuitka Design - Inside View



Nuitka

Goal: Pure Python - only faster (1 of 2)

- 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 (2 of 2)

- 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 )
```

News: Target Language Change

- C++11 -> C++03
  - Evaluation order not there with variadic templates.
  - Raw strings not as perfect anyway.
- Allows to support Android, Microsoft Visual Studio, etc.
- More portable code, was more work, but now we support much more platforms out of the box.

Function calls arguments (1 of 2)

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
PyObject *_tmp1 = a();
PyObject *_tmp2 = b();
PyObject *_tmp3 = c();

f( tmp1, tmp2, tmp3 );
```

Function calls arguments (2 of 2)

Therefore we have complex "ordered evaluation", supplying arguments to C++ functions with "sequence operator" usages.

Improvement since 2012: The ordered evaluation is now a general solution, does no longer rely on using compiler specifics.

The MSVC compiler could not be tricked at all, it takes liberty to arrange parameters for calls, in the optimal way, no matter what. So this had to be solved to ensure long term viability.

Nuitka the Project - git (1 of 2)

- Master (current release 0.4.4)

  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 might be done.

- Develop

  Future possible release, that is supposed to be fully correct, but it isn't supported as much, and at times may have problems or inconsistencies that will be removed before release.

Nuitka the Project - git (2 of 2)

- Factory

  Frequently re-based, staging here commits for develop, until they feel robust, my goal is to be perfectly suitable for `git bisect`. Orignally the develop branch was re-based to achieve that, but that's no longer done.

- Feature branches

  Not used anymore really. Nuitka is relatively feature complete, and can advance multiple things, in logical steps concurrently on a relatively stable basis.

Nuitka the Project - Plan (1 of 4)

1. Feature Parity with CPython ✔

   Understand the whole language and be fully compatible. Compatibility is amazingly high. Python 2.6, 2.7, 3.2, and 3.3 all perfect.

   All kinds of inspections now like Nuitka compiled code. Functions have proper flags, locals/globals identity or not, run time patched `inspect` module to be more tolerant.

   Patched and upstream integrated patches (wxpython, PyQt, PySide) which hard coded non-compiled functions/methods.

   New: Threading is somewhat supported.

Nuitka the Project - Plan (2 of 4)

2. Generate efficient C++ code ✔

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

   2013: We are now making faster function calls than before.

   2013: More built-ins covered.

   Open: Apply knowledge of variable usage patterns more often. Already doing it for parameter variables with or without `del` on them. Variables that cannot be used unassigned should be clearer too.

   Open: 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 - Plan (3 of 4)

3. "Constant Propagation"

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

4. "Type Inference"

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

   Only starting to exist. 🚫

Nuitka the Project - Plan (4 of 4)

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.

   Does not exist yet.

6. `hints` Module

   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.

   Does not yet exist. 

Nuitka - Interfacing to C/C++ (1 of 2)

The inclusion of headers files 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.

```
strchr = libc.strchr
strchr.restype = c_char_p
strchr.argtypes = [c_char_p, c_char]

strchr("abcdef", "d")
```

Nuitka - Interfacing to C/C++ (2 of 2)

```
typedef char * (*strchr_t)( char *, char );
strchr_t strchr = LOAD_FUNC( "libc", "strchr" );

strchr("abcdef", "d");
```

- Native speed calls, with *no overhead*.
- The use of `ctypes` interface in Python replaces them fully.
- Tools may translate headers into `ctypes` declarations.

Nuitka the Project - Activities

Current:

- SSA based optimization (removal of "value friends") 
- Proper handling of SSA for exception handlers, finally blocks
- Some actual type inference
- CPython3.3 tests also a as git repository with documented commits per diff 

Maybe this year:

- Making direct calls to known or suspected functions, removing argument parsing inside programs.
- More performance data on http://nuitka.net/pages/performance.html 

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 optimization. New: Important new ones.

Conversion: The class statement

Classes are functions with "early variable closure". Class bodies build a dictionary, then the meta-class is determined and asked to build a class type object, which is then assigned to an object.

```
# 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() )
```

- Python3 is more complex, but same idea.
- For Nuitka there are only functions.

Conversion: Boolean expressions and and or

The short circuit operators `or` and `and` tend to be only less general that the `if`/`else` expressions and are therefore re-formulated as such:

```
expr1() or expr2()
```

```
_tmp if ( _tmp = expr1() ) else expr2()
```

```
expr1() and expr2()
```

```
expr2() if ( _tmp = expr1() ) else _tmp
```

- Only switching sides.
- We only have conditional expressions left as "short-circuit".

Conversion: Many others (1 of 2)

- Details of this can be found in Developer Manual.
- For/ while loops -> loops with breaks, explicit iterator handling
- With statements -> trying a block of code, with special exception handling
- Decorators -> simple function calls with temporary variables
- Inplace / complex / unpacking assignments -> assignments with temporary variables.

Conversion: Many others (2 of 2)

- Details of this can be found in Developer Manual.
- No "elif" -> nested if statements
- no "else:" in "try:" blocks -> conditional statements with temporary variables
- Contractions -> functions too
- Generator expressions -> generator functions
- Complex calls * * or * -> simple calls that merge with other parameters in helper functions

Nuitka the Project - But what if builtins change

- Changes to __builtin__ module
  - Changes to builtins module
  - Values escape in Python

Nuitka the Project - But anything could change

- Almost anything may change behind Nuitka's back when calling unknown code
- But guards can check or trap these changes

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 released source archives, does 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 about major stuff.

Optimization 1 of 6

- Peephole visitor
  - Visit every module and function used
  - In each scope, visit each statements `computeStatement`.

    Example: `StatementAssignmentAttribute` (`AssignNodes.py`)
  - In each statement, visit each expression

    Example: `ExpressionAttributeLookup` (`AttributeNodes.py`)
  - In the order of execution call sub-expressions

Optimization 2 of 6

- Dead code elimination
  - Statements can be abortive (`return`, `raise`, `continue`, `break`)
  - Subsequent statements are unreachable
- Dead variables needs SSA
  - Coming. Variables only written to, are dead.

Optimization 3 of 6

- Frame stack avoidance
  - Frames in Python are mandatory for functions and modules.
  - Nuitka generates a frame for bodies of them, but then it shrinks their size to cover only things that can raise.
  - Example:

```
def f():
   a = 42   # cannot fail
   return a # cannot fail
```

  - For methods, that assign to self attributes, they may not raise as well, depends on base class though.

Optimization 4 of 6

- Trace collection
  - Setup variables at entry of functions or modules as:
    - "Uninit" (will raise when used)
    - "Unknown" (no idea what it is)
    - "Init" (known to be init, but unknown what it is)
  - During visit of all assignments and references to variables
    - Add references when used
    - Start a new "version" when an assignment takes place
    - "Merge" at code paths joining

Optimization 5 of 6

- Using traces, we can make dead assignment elimination
- Teaches code generation about access that cannot raise, more efficient code.

Optimization 6 of 6

- Escape analysis
  - When people say "but Python is too dynamic, everything may change at any time".
  - Need to trace the escape of values.
  - Guards to detect escape, ideally when it happens.

    Writes to global variables e.g. should trigger value generation flags. Very fast to detect if a global is changed.
  - Unescaped values, esp. lists, might see better

Discussion

- Will be here for all of PyCON-EU and Sprint, I have put up a Nuitka sprint, you are free to come and bring your software, and we can try Nuitka on it together. I *welcome* questions and ideas in person. Questions also welcome via Email to kay.hayen@gmail.com or on the mailing list.
- My hope is:

  1. More contributions (there are some, but not enough).
  2. To fund my travel, donations
  3. A *critical* review of Nuitka design and source code, would be great.
  4. 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-2013.pdf
- Diagrams were created with OOo Draw
- Icons taken from visualpharm.com (License requires link).
- For presentation on PyCon EU