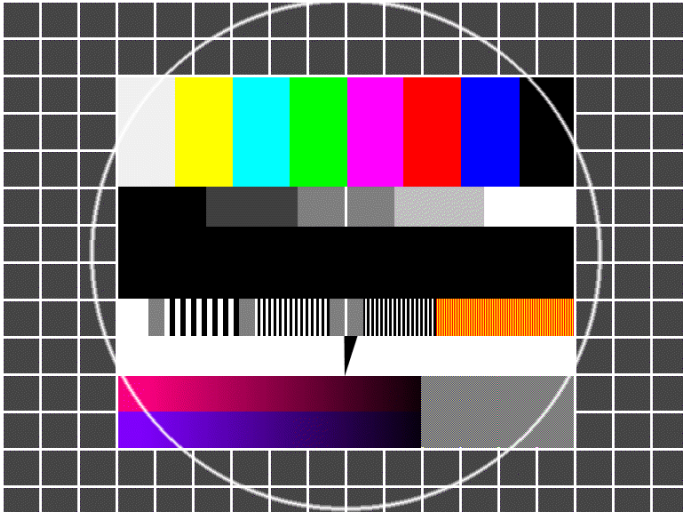


Topics (1 of 5)

- Intro
  - Who am I? / What is Nuitka?
  - System Requirements
- Demo
  - Compiling a simple program
  - Compiling full blown program (Mercurial)

Nuitka

Topics (2 of 5)

- Nuitka goals
  - Faster than before, no new language
  - No language, nor compatibility limits
  - Same or better error messages
  - All extension modules work
- How do we get there
  - Project Plan
  - Python versions / platforms
  - Join me

Nuitka

Topics (3 of 5)

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

Topics (4 of 5)

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

Topics (5 of 5)

- Optimization (coming)
  - Inlining function calls
  - Escape analysis
  - Shape analysis
  - Type inference

Intro - Nuitka (1 of 3)

- Started with goal to be *fully* compatible Python compiler that does not invent a new language, and opens up whole new Python usages.
- Thinking out of the box. Python is not *only* for scripting, do all 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)

- Nuitka is under Apache License 2.0
- Very liberal license
- Allows Nuitka to be used with all software


Intro - Nuitka (3 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
- Android and iOS need work, but should be possible


System Requirements

- Nuitka needs:
  - Python 2.6 or 2.7, 3.2, 3.3, 3.4, (3.5 beta3 works like 3.4)
  - C++ compiler:
    - g++ 4.4 or higher
    - clang 3.2 or higher
    - Visual Studio 2008/2010/2014
    - MinGW for Win32, MinGW 64 bit for Win64
  - Your Python code 😉
  - That is it ✔

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 `long x` (with `PyObject *` fallback in case that overflows.)

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

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

- Type hints as proposed for Python 3.5 may work too.

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, 3.3 and 3.4 all near 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: Access to `__closure__` works too. New: Support for `.pth` files. New: Call argument deletion order. New: Much more...

Nuitka the Project - Plan (2 of 4)

2. Generate efficient C-ish code ✔

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

   2015: Value propagation should start to remove code.

   2015: Using trace knowledge to determine need for releases. ✔

   2015: Exceptions are now fast. ✔

Nuitka

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` declarations to the point, where it can avoid using it, and make direct calls and accesses, based on those declarations.

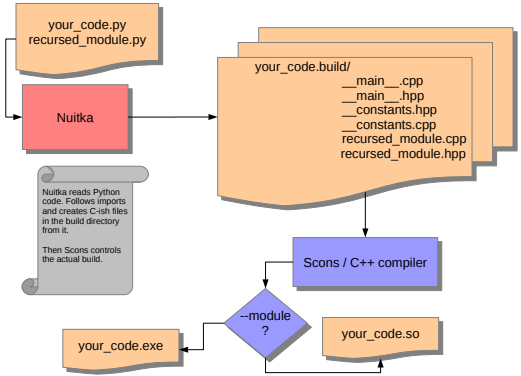
   Does not exist yet.

6. `hints` Module

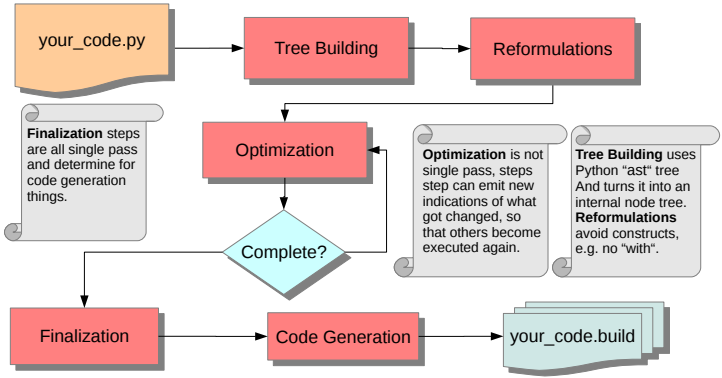
   Should check under CPython and raise errors, just like under Nuitka. Ideally, the code simply allows Nuitka to detect, what they do, and make conclusions based on that, which may be too ambitious though.

   Does not yet exist. 

Nuitka Design - Outside View



Nuitka Design - Inside View



Demo Time (1 of 2)

Simple program:

```
def f():
    def g(x, y):
        return x, y

    x = 2
    y = 1

    x, y = g(x, y) # can be inlined

    return x, y
```

Demo Time (2 of 2)

Nuitka --verbose output (part 1):

```
7  : new_statements : Removed useless frame object of 'g.'
14 : new_statements : Reduced scope of tried block.
6  : new_statements : Reduced scope of tried block.
7  : new_statements : Dropped propagated assignment statement to 'g'.
10 : new_statements : Dropped propagated assignment statement to 'x'.
11 : new_statements : Dropped propagated assignment statement to 'y'.
14 : new_expression : Value propagated for 'g' from '7'.
14 : new_expression : Value propagated for 'x' from '10'.
14 : new_expression : Value propagated for 'y' from '11'.
14 : new_constant   : Tuple with constant arguments. Constant result.
14 : new_statements : Replaced call to function 'g' with direct functio
14 : new_statements : Function call inlined.
```

Demo Time (3 of 4)

Nuitka --verbose output (part 2):

```
6  : new_statements : Uninitialized variable 'g' is not released.
6  : new_statements : Uninitialized variable 'g' is not released.
7  : new_statements : Dropped propagated assignment statement to 'inlin
7  : new_statements : Dropped propagated assignment statement to 'inlin
8  : new_expression : Value propagated for 'inline_1__x' from '14'.
8  : new_expression : Value propagated for 'inline_1__y' from '14'.
8  : new_constant   : Tuple with constant arguments. Constant result.
7  : new_statements : Uninitialized variable 'inline_1__x' is not relea
7  : new_statements : Uninitialized variable 'inline_1__y' is not relea
7  : new_statements : Removed all try handlers.
7  : new_expression : Outline is now simple expression, use directly.
```

Demo Time (3 of 4)

- Mercurial is a complex Python 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 4)

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

```
tmp_left_name_2 = par_a;
tmp_right_name_1 = par_b;

tmp_left_name_1 = BINARY_OPERATION_ADD(
    tmp_left_name_2,
    tmp_right_name_1
);
```

Generated Code (2 of 4)

```
if ( tmp_left_name_1 == NULL )
{
    FETCH_ERROR_OCCURRED(
        &exception_type,
        &exception_value,
        &exception_tb
    );

    exception_lineno = 4;
    goto frame_exception_exit_1;
}
```

Generated Code (3 of 4)

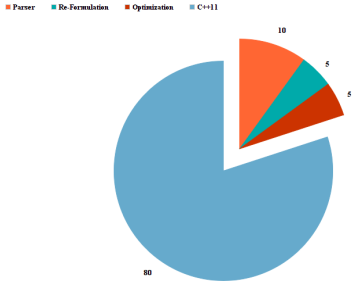
- News: Target Language Changes
- Last time: C++11 -> C++03
- This time: C++03 -> C-ish C++
- Next time maybe: C-ish C++ -> C89/C99
- Exceptions in C++ suck performance wise.
- In-place operations steal objects from behind the back, no way of doing that in C++ in a sane way.
- Faster end result, but need more knowledge code generation.

Generated Code (4 of 4)

- Before `C-ish` Nuitka used to generate code that was using C++ objects to hide technical details of object release.
- Now we control it all, and the code generation "manually" releases temporary values on error exits.
- The `C-ish` evolution changed code generation entirely. Going from C++ to C took a large amount of work. 
- Improved performance, but only slightly. 
- Some cases, in-place string operations, or exceptions couldn't be fast without it though, so it was necessary.

Evolution of Nuitka (1 of 3)

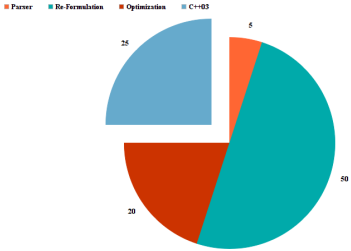
Knowledge used to be largely in how compatible code is generated.



This was largely only a proof that a compatible compile is feasible and can be faster all around.

Evolution of Nuitka (2 of 3)

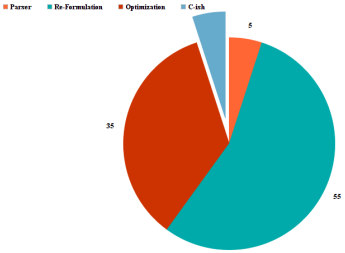
Then incrementally extracting to re-formulation and optimization



For a long time couldn't make optimization understand sufficiently.

Evolution of Nuitka (3 of 3)

Now optimization is carrying the day



Yet... until now, roughly same performance... that can now change. 😜

Nuitka the Project - Availability (1 of 2)

- Stable / git master (current release 0.5.13)

  The stable version should be perfect at all times and is fully supported. As soon as bugs are known and have fixes there will be minimal change hot-fix releases.

- Develop / git develop (pre-releases 0.5.14pre4)

  Next release building up, supposed to be correct, but at times may have some problems or inconsistencies.

Nuitka the Project - Availability (2 of 2)

- Factory / git factory

  Frequently re-based, staging here commits for develop, until they are robust, the goal is to be perfectly suitable for `git bisect`.

- Not just git
  - Repos with RPMs CentOS, RHEL, openSUSE, Fedora
  - Repos with DEBs Ubuntu and Debian
  - Debian Stable
  - Arch
- PyPI nuitka

Nuitka the Project - Join

You are *welcome*.

Accepting patches as ...

- Whatever `diff -ru` outputs for you
- Patch queues as output by `git format-patch`
- pull requests

I will do necessary integration work is *mine* based on anything that ever lived on `master` or `develop`,

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

Nuitka Testing Correctness

- Correctness tests can be as simple as they get

```
a = something() + other()
print a
```

- We have a perfect "oracle" in CPython. Nuitka and CPython are supposed to behave the same. So we just compare outputs.
- Doing some replacements in regular expressions to filter addresses, etc.

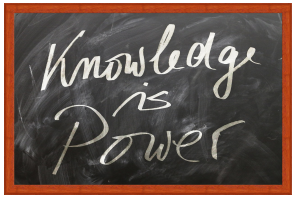
Nuitka Testing Correctness

- Correctness is also tested by compiling applications and then running your test suite. For example Mercurial. ✔
- And there CPython test suites, which if they don't falsely say "OK", will find the right kind of bugs. ✔
- Running tests with "wrong" Python versions give nice test coverage of exception stack traces. ✔
- A Buildbot instances makes sure errors hardly go unnoticed. ✔

Nuitka Performance

- Performance illustration is damn bloody harder
- For Development
  - Heavy usage of "valgrind" and comparison of ticks
  - Compare small constructs ✔
  - Doing it for selected Python constructs.
  - Not documented yet... 😟
- Currently up on Nuitka Speedcenter

Nuitka Performance



- For *Users* there is nothing yet
- Really bad situation, no way of knowing 🤔
- Any statement, say 258% is to be wrong for any program or code.
- Performance depends on constructs used, data used, applicable optimization.
- Bad bad, we need to find a way out of this.

Nuitka Performance

- The Rescue
- You browse the code sorted by ticks used.
- Highlight parts that are slower than CPython (sure to exist).
- Highlight parts that cost the most.
- Link to Nuitka knowledge about the program.
- Help and create that. Now.

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 and code generation 
- Function inlining generally working 
- CPython3.5 except for coroutines and `async wait`. Beta 3 otherwise works. 

Maybe this year:

- More performance data on http://nuitka.net/pages/performance.html 
- Type specific code generation
- Shape analysis

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. They make Nuitka's life easier, and Nuitka more dependent on powerful analysis.

Language Conversions: Many of them (1 of 2)

- Details of this can be found in Developer Manual.
- `for`/ `while` -> loops with explicit `break` statements
- `with` statements -> `try` a block of code, special exception handling
- Decorators -> simple function calls temporary objects
- `class` becomes call of metaclass with dictionary.
- Inplace / complex / unpacking assignments -> assignments with temporary variables, all simple.

Language Conversions: Many of them (2 of 2)

- Details of this can be found in Developer Manual.
- The `elif` -> nested `if` statements
- no `else` in `try` blocks -> checks on indicator variables
- Contractions -> are functions too
- Generator expressions -> generator functions too
- Complex calls `**` or `*` -> simple calls that merge in helper functions

Nuitka the Project - But what if builtins change

- Changes to `__builtin__` module
  - We can trap them: See `Nuitka_BuiltinModule_SetAttr`
  - Could raise `RuntimeError` or may unset a flag that allows for quick code paths using the assumption it's the original one.
  - Already traping changes to `__import__`, `open`, etc.

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
- Compiled module objects can trap module attribute writes
- Locals maybe as well.

Optimization 1 of 5

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

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

    Example: `ExpressionAttributeLookup` (`AttributeNodes.py`)
  - Visitation strongly follows execution order.

Optimization 2 of 5

- Dead code elimination
  - Statements can be abortive (`return`, `raise`, `continue`, `break`)
  - Subsequent statements are unreachable
- Dead variables
  - New in next release. Variables only written to, are dead and need not be such.
  - Can be removed (no side-effect for creation/del), or replaced with a temporary variable instead.

Optimization 3 of 5

- Frame stack avoidance
  - Frames in CPython are mandatory
  - Nuitka puts up a (cached) frame with minimal scope
  - Example:

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

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

Optimization 4 of 5

- Trace collection (SSA)
  - Setup variables at entry of functions or modules as:
    - "Uninit" (no value, will raise when used, local variable)
    - "Unknown" (no idea if even assigned, module or closure variable)
    - "Init" (known to be init, but unknown value, parameter variable)
    - "Merge" (one of multiple branches)
  - During visit of all assignments and references to variables
    - Using information from last pass to look ahead.

Optimization 5 of 5

- 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 even better code


Discussion

- Will be here for all of EuroPython and Sprint. I *welcome* questions and ideas in person.

  Questions also on the mailing list.

- My hope is:

  1. More contributions (there are some, but not enough).
  2. To fund my travel, `*donations <http://nuitka.net/pages/donations.html>*`_.
  3. A *critical* review of Nuitka design and source code, would be great.

Nuitka

This Presentation

- Created with rst2pdf
- Download the PDF http://nuitka.net/pr/Nuitka-Presentation-PyCON-EU-2015.pdf
- Diagrams were created with Libreoffice Draw
- Icons taken from visualpharm.com (License requires link).
- For presentation on EuroPython 2015