

Themen (1 von 3)

- Ziele für einen echten Python Compiler
 - Nur schneller als vorher, keine neue Sprache
 - Keine Einschränkungen
 - Originale Fehlermeldungen
 - Alle Erweiterungsmodule sollen funktionieren
- Wahl der Zielsprache
 - Übersetzen von Python nach was
 - Wichtige Entscheidung

Themen (2 von 3)

- Problematische Unterschiede C++ und Python
 - Operatoren "or" und "and" sind anders
 - Kein try/finally in C++
 - Auswertung der Parameter bei Aufrufen nicht garantiert
 - Strings
- Verworfenen Alternativen
 - PyPy / RPython (limitierter Sprachsatz)
 - PyPy / JIT (nur JIT)
 - Pyrex / Cython (andere Sprachen als Python, inkompatibel)

Themen (3 von 3)

- Nuitka Design
 - Äußerer Überblick
 - Innerer Überblick
- Nuitka das Projekt
 - Der "git flow"
 - Projektplan
 - Status
 - Mitmachen
 - Lizenz

Ziele für einen echten Python Compiler (1 von 2)

- Reines Python nur schneller

Spezielle Sprachkonstrukte sollten außen vor sein, auch das Annotieren in einer anderen Datei sollte tabu sein.

- Besserer Code auch wenn Performance wichtig ist

Ein Python-Compiler sollte den Programmierer dazu bringen, besseren Python Code zu schreiben, nicht schlechteren.

1. Es sollte nicht mehr nötig sein, benamste Konstanten zu vermeiden, weil diese langsamer sind.
2. Funktionen so aufrufen, wie es am lesbarsten ist, nicht wie es die Python-Laufzeit am schnellsten bearbeiten kann.

Ziele für einen echten Python Compiler (2 von 2)

- Probleme der Laufzeit zur Compilezeit melden

Wenn eine Variable gelesen wird, die nie gesetzt wird, dann merkt man das in Python nur sehr spät. Das kostet Entwicklungszeit, Testzeit, jedenfalls sehr viel. Der Compiler sollte da die Rolle von PyLint weitgehend übernehmen können, und evtl. sogar noch mehr leisten als dieses.

- Keine Einschränkungen


Was immer Python kann, muss der Compiler können. Egal wie sehr es weh tut. Und Python erlaubt sehr viel.

- Originale Fehlermeldungen

Bei Syntax-Fehlern möglichst die originalen Meldungen. Natürlich dürfen gerne Warnungen dazu kommen über Laufzeitfehler.


Ziel: Reines Python nur schneller


- Typisierungen von Python

1. Wenn eine Variable den Typ "cint" hat, wie wird dann der Überlauf funktionieren, so wie in C, oder wie in Python garnicht?
2. Wenn eine Variable den Typ "char *" hat, ist der String dann veränderlich oder so wie in Python garnicht?
3. Wie ist das wenn konvertiert wird, Exceptions ja und welche?
4. Eine komplett neue Semantik ist schlecht. 

Stattdessen sollte der Python Compiler erkennen, ob ein Performance-Vorteil möglich ist, und entweder zur Compilezeit oder zur Laufzeit anderen Code zum Einsatz bringen, wenn Überläufe passieren.

Ziel: Reines Python nur schneller (1 von 4)

- Typenannotation im separater Datei
 1. Nur effektiv wenn vom Compiler ausgeführt. Schade! 
 2. Tatsächlich sind Aussagen wie "muss ein Integer sein" wichtige Information.

Diese Verbessern die Code-Qualität generell. Sie sind u.U. wie Assertions und von daher *nicht* auszulagern. 
 3. Zwei Dateien zu pflegen hat seine eigenen Schwierigkeiten, ich persönlich bin oft schon von einer Datei überfordert.

Ziel: Reines Python nur schneller (2 von 4)

- Wer will schon eine neue Sprache lernen.

Jedenfalls nicht alle. 🚫

Manche bestimmt, neue Sprachen lernen oder gar erfinden ist so spannend. Habe ich vor fast 30 Jahren schon hinter mich gebracht. 😊

Manche tun es nur aus der Not heraus, dass reines Python bisher nicht schnell geht, mit der Ausnahme vielleicht PyPy JIT, aber die kann auch nicht alles.

Der Compiler sollte diese Leute "befreien".

Ziel: Reines Python nur schneller (3 von 4)

- Neue Sprache verliert alle Tools

1. IDE Autovervollständigung (emacs python-mode, Eclipse, etc.) 😞
2. IDE Syntax-Highlighting 😞
3. PyLint Checks 😞
4. Dependency Graphen 😞
5. Kein einfacher Fallback auf CPython, Jython, PyPy, IronPython, etc. 😞

Aua, ohne das will/kann ich nicht auskommen. 🚫

Ziel: Reines Python nur schneller (4 von 4)

- Lösungsplan ohne neue Sprache:

Ein Modul "hints" enthält in Python implementierte Checks, Assertions, etc.

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

Der Compiler erkennt diese Hinweise und legt "x" dann entweder gleich als "int x" oder "PyObjectOrInt" an.

Idealerweise würden Nuitka solche Hints durch Inlinen von "mustbeint" umsetzen, und so zu Schlüssen kommen, die teilweise auch aus dem hier folgen:

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

Ziel: Code nicht der Performance opfern (1 von 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
```

Ziel: Code nicht der Performance opfern (2 von 5)

```
def someFunction():  
    len = len  
  
    # some len using code follows
```

- Auch so eine "Perle" der Python Optimierung 😞
- Globale Variablen in den lokalen Namensraum importieren, macht alles zunächst mal nur unlesbarer und ist nicht universell. Ohne dies ist aber z.B. len() extrem viel langsamer aufzurufen.

Ziel: Code nicht der Performance opfern (3 von 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-Argumente sind extrem teuer. Für den Aufruf muss ein Dictionary erstellt werden, was vom Aufgerufenen für jeden Argumentnamen geprüft werden muss, auf überzählige Argumente getestet werden muss, etc.

Ziel: Code nicht der Performance opfern (4 von 5)

- Optimierungen bedeuten bei CPython fast immer:

1. Schlechter lesbarer Code als nötig 😞

2. Schlechteres Design als nötig 😞

- Alle diese Dinge kann ein Compiler viel besser handhaben

Ohne den Quellcode zu verändern und wenn doch durch Annotationen, die zusätzliche Prüfungen bedeuten

Ziel: Code nicht der Performance opfern (5 von 5)

- Performance-Optimierungen im Quellcode sollen sich nicht lohnen
 1. Der Entwickler soll nicht "int" sagen müssen, wenn es aus dem Programm hervorgeht
 2. Manuelles Inlinen, lokalisieren von Identifiern, etc. soll alles unnötig sein
- Wir nutzen doch Python oft wegen des lesbaren Code. "Schnell" ist ein Feature, das *jemand anders* lösen sollte.
- Damit ist im Grunde auch gesagt, warum *ich* Nuitka mache. 😊

Verworfen Alternative: PyPy - RPython (1 von 2)

- Ich habe Patches für PyPy/RPython entwickelt:

Die wurden akzeptiert, allesamt Trivialitäten. Die Community von PyPy ist sehr freundlich und kooperativ. Alles war im IRC-Channel mit Patches möglich, hat mir gut gefallen. 😊

- Mit RPython experimentiert

Die Performance-relevanten Teile in RPython umzuschreiben war möglich und hat auch teilweise Spaß gemacht. Kleinigkeiten konnte man leicht dazufügen, aber...

Verworfen Alternative: PyPy - RPython (2 von 2)

- Ein reduzierter Compiler

Leider verlangt RPython nur ein "reduced" - also reduziertes - Python zu verwenden. 😞

Erzwingt damit Design-Änderungen, die mir nicht gefallen haben

- So kann ich nicht alle Ziele erreichen 🚫

Verworfen Alternative: PyPy - JIT

- Aber es ist kein Compiler

Der JIT weiß nie so genau, wie weit er gucken kann, was er verfolgen darf, ohne dass ein Overhead für alles entsteht. Deshalb muss sich ein JIT in seiner Analyse zwangsläufig beschränken.

- Zu komplex

Das Design ist sehr beeindruckend, aber eben auch zu komplex.

- Viele Ziele sind nur fast erreichbar

Was der JIT erkennt, ist gelöst, was nicht, das nicht. 🚫

Verworfen Alternative: Pyrex / Cython

- Ich habe Patches für Cython entwickelt:

Die wurden nicht akzeptiert, weil sie Cython von Pyrex entfernt hätten, was zumindest damals wichtig war. 😞

- Und:

Die Richtung stimmte für mich nicht. Zu viele Entwickler/User haben produktiven Code in einer Sprache, die absolut nicht Python ist.

Das hauptsächliche Ziel von Cython ist m.E. Python-Module und C/C++ anzubinden. Ganze Programme scheinen garnicht auf der Rechnung zu sein. Ausserdem steht Optimierbarkeit von Hand im Vordergrund.

- Keine Übereinstimmung bei den Zielen. 🚫

Die Zielsprache (1 von 4)

- Wahl der Zielsprache, eine echte Entscheidung
 - Die Portabilität von Nuitka entscheidet sich hier.
 - Wie schwer ist es den Code zu generieren.
 - Gibt es die Python C-API als Binding
 - Kenne ich die Sprache.
 - Hilft die Sprache Bugs zu finden?
- Die Kandidaten
 - C++03, C++11, Ada

Die Zielsprache (2 von 4)

Sprachen Matrix :

Anforderung\Sprache	C++03	C++11	Ada
Portabel	Ja	Nein ¹	Ja
Kenntnisse	Ja	Nein ²	Ja
Python C-API	Ja	Ja	Nein ³
Laufzeitchecks	Nein	Nein	Ja ⁴
Generierung	Schwer	Leicht	Schwer

Die Zielsprache (3 von 4)

Die Nummern beziehen sich auf die Links in der vorherigen Tabelle.

- 1:: C++11 wird von noch keinem Compiler vollständig unterstützt (temporäres Problem)
- 2:: Nicht viele haben C++11 Kenntnisse. Mein *einzig*er C++11 Code ist der in Nuitka vorhandene und von Nuitka generierte.
- 3:: Die Python C-API für Ada wäre selbst zu erstellen, möglich aber lästig.
- 4:: Runtime Checks hat nur Ada in der Qualität. Ich vermisse die "CONSTRAINT_ERROR", wenn z.B. manche Datenstrukturen ein "valid" Feld haben, und ich selbst dieses in anderen Sprachen prüfen muss.

Die Zielsprache (4 von 4)

Die Entscheidung für C++11 ist letztlich eine gegen Portabilität, gegen die Kenntnis der Sprache, was schwerwiegende Nachteile sind. Daher fiel es leicht, noch die GNU Extension dazu zu nehmen, da ohnehin nur ein C++ Compiler aktuell geht.

Für C++11 spricht eigentlich nur die leichtere Code-Generierung, da hier Templates durch die variadischen Funktionen leicht zu machen sind. Klassen für Variablen müssen nur einheitliche Interfaces haben, nicht aber eine gemeinsame Basisklasse, und ich muss nicht vorab wissen, wieviele Parameter es werden, etc.

Bei C++03 hätte ich dafür Boost benötigt, womit auch viel von C++11 möglich wäre. Aber dann gibt es noch Sachen wie "Raw Strings", die einfach viel Arbeit sparen.

Für Ada hätte gesprochen, dass die Laufzeitprüfungen, mir manches an Debugging deutlich verkürzt hätten. Dafür hätte ich die Python C-API selbst binden müssen.

Generierter Code (1 von 2)

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

```
PRINT_ITEMS( true, NULL, _python_str_digest_2c9fbb02f98767c025af8ac4a1461a18,
PyObjectTemporary(
  BINARY_OPERATION(
    PyNumber_And,
    PyObjectTemporary(
      BINARY_OPERATION(
        PyNumber_And, _mvar__main__a.asObject0(), _mvar__main__b.asObject0()
      )
    ).asObject(),
    _mvar__main__d.asObject0()
  )
).asObject() );
```

Generierter Code (2 von 2)

Ein wichtiges Ziel für die Code-Generierung war es nicht selbst mit temporären "PyObject *" Variablen hantieren zu müssen, da müsste die Code-Generierung dem Ziel-Compiler bei allen möglichen Exits selbst helfen. Dafür gibt es "PyObjectTemporary" und dessen Destruktor.

Der String enthält Sonderzeichen, daher die md5-Summe im Identifier, sonst sein Inhalt.

Das BINARY_OPERATION ist ein Wrapper der Python C-API, der eine C++ Exception werfen würde, sollte NULL returned werden. In Nuitka generiertem C++ wird nie ein Return-Wert geprüft, im Fehlerfall gibt es immer eine Exception.

Der C++ Compiler gibt im Falle der Exceptions oder auch so über die "PyObjectTemporary" oder auch "PyObjectLocalVariable", "PyObjectSharedVariable" vorhandenen Referenzen frei.

Python nach C++ Lücke (1 von 8)

Boolsche Operationen in Python sind Selektionen :

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

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

Lösung ohne Temp-Variable gesucht. Mit GNU Erweiterung "?:" :

```
SELECT_IF_TRUE( _mvar__main__a.asObject() ) ?: _mvar__main__b.asObject()
```

Der "?:" Operator ist "short-circuit", d.h. die rechte Seite wird nur evaluiert, wenn nicht SELECT_IF_TRUE den Wert "NULL" returned. Auf diese Weise kann das Verhalten von Python erreicht werden. Mit Standard-Mitteln geht es (als Expression) nach meinem Kenntnisstand nicht.

Python nach C++ Lücke (2 von 8)

Comparison Chains gehen in C++ auch nicht recht:

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

In C++ gibt es kein echtes try/finally:

Die C++ Götter glauben fest an die Vorteile von RAII, sie werden ihre Gründe haben, die hier nicht interessieren.

So muss try/finally emuliert werden, durch Fangen der Exception, Speichern in einer Variable, und durch erneutes Werfen der gespeicherten Exception.

Python nach C++ Lücke (3 von 8)

Komplikationen durch "break", "continue" und "return":

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

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

Python nach C++ Lücke (4 von 6)

- Der Aufruf von "cleanup" geschieht in jedem Fall.

Das Verhalten von "break", "continue" und "return" ist wie eine Exception zu sehen. Daher auch die Umsetzung als Exception für diese Fälle in Loops. Der Fall "loop with try-finally" wird gesondert behandelt und bekommt langsameren Code. Schöner ist natürlich, wenn Python "continue" mit C++ "continue;" übersetzt werden kann.

An RAI ist nicht wirklich zu denken, da der Zugriff auf lokale Variablen fehlt, und man Klassen nicht mit Zugriff auf diese ohne viel Aufwand versorgen kann.

Python nach C++ Lücke (5 von 8)

- Funktionsaufrufe

In Python ist die Reihenfolge der Auswertung von Parametern *garantiert*. In C++ nicht.

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

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

Python nach C++ Lücke (6 von 8)

- Funktionsaufrufe

ARM und Intel machen es verschieden bei C/C++:

- von links nach rechts (ARM, Register für Parameter)
- von rechts nach links (Intel, Stack für Parameter)

In C++ nicht lösbar. Durch geschickte Verwendung von Macros, kann jede Funktion so definiert werden, dass die Parameter in C++ richtig ankommen:

```
#define RICH_COMPARE_LT( operand1, operand2 ) _RICH_COMPARE_LT( EVAL_ORDERED_2( operand1, operand2 ) )  
  
static PyObject *_RICH_COMPARE_LT( EVAL_ORDERED_2( PyObject *operand1, PyObject *operand2 ) );
```


Python nach C++ Lücke (7 von 8)

- Strings

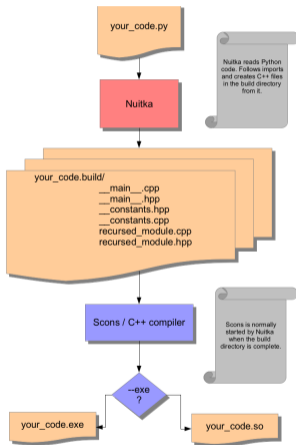
Python hat sehr elegante raw strings. Damit kann man praktisch jeden "Blob" im Code unterbringen.

C++11 hat "raw strings", aber die brauchen zumindest im g++ solchen Code:

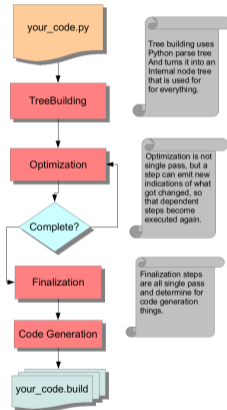
Python nach C++ Lücke (8 von 8)

```
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 - Äußerer Überblick



Nuitka Design - Innerer Überblick



Nuitka das Projekt - git flow

- Wurde für das letzte Release und Hotfixes danach erstmals genutzt.

- Stable

Die stabile Version, sollte vollständig tun und wird unterstützt. Soweit Fehler bekannt werden, werden Hotfixes gemacht.

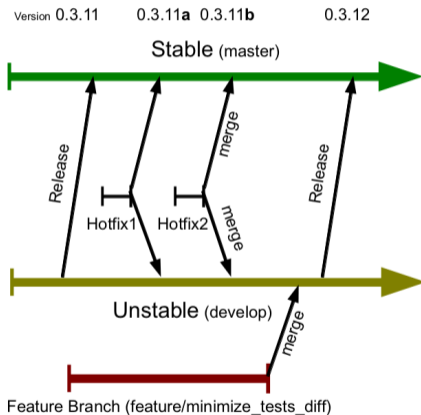
- Develop

Ein mögliches zukünftiges Release, sollte im Grunde auch vollständig tun, wird aber nicht unterstützt, und es kann Probleme oder Inkonsistenzen geben.

- Feature Branches

Hier passieren länger laufende, ein Thema betreffende Entwicklungen, die noch nicht fertig sind, aber schon mal öffentlich. Muss nicht funktionieren.

Nuitka das Projekt - git flow (2 von 2)



Nuitka das Projekt - Projektplan (1 von 2)

1. Feature Gleichstand mit CPython

Den gesamten Sprachumfang verstehen und absolut kompatibel verhalten.

2. Effizienten Code generieren

Unter Verwendung von nur "PyObject *" das Verhalten effizient umsetzen und daraus einen Gewinn erzielen.

3. "Constant Propagation"

Bestimmung von möglichst vielen Werten und Einschränkungen und auf dieser Basis noch effizienteren Code erstellen.

Nuitka das Projekt - Projektplan (2 von 2)

4. "Type Inference"

Erkennung und Sonderbehandlung von "Strings", "Integern", "Listen", etc. in dem Programm.

5. Schnittstelle zu C-Code

Nuitka soll "ctypes" bindings erkennen und deren Aufrufe so durchführen, als wären diese in C deklariert worden.

6. "hints" Modul

Damit kann der Entwickler Hinweise an Nuitka und CPython geben kann, etwa, dass ein Parameterwert immer ein Integer ist.

Nuitka das Projekt - Status (1 von 5)

1. Feature Gleichstand mit CPython

Erreicht. ✓

Im "feature/minimize_CPython26_tests_diff" Branch kommen auch obskure Features wie z.B. Unterstützung von "sys.getframe()" dazu kommen. ⚙️

2. Effizienten Code generieren

Bei pystone gibt es einen satten Sprung um 258%. ✓

Bei pybench ist der Sprung oft "inf", d.h. Nuitka braucht keine messbare Zeit mehr, oder die Faktoren sind zumindest sehr hoch. ✓

Nur mit den Exceptions kann ich noch nicht ganz zufrieden sein, diese sind in C++ zu langsam und müssten öfter vermieden werden. Hier ist noch was zu tun. ⚙️

Nuitka das Projekt - Status (2 von 5)

3. "Constant Propagation"

Weitgehend erreicht. ✓

4. "Type Inference"

Gibt es noch nicht. ✗

Da sie zuverlässig funktionieren muss, kann man sich nicht etwa bei PyLint bedienen, wo Fehler eher verzeihlich sind.

Nuitka das Projekt - Status (3 von 5)

5. Schnittstelle zu C-Code

Gibt es noch nicht. 

Die Einbindung von C-Header Dateien oder Syntax ist tabu.

Ich habe die Vorstellung, dass es Nuitka möglich sein sollte, aus den ctypes Deklarationen einer C-Schnittstelle, einen Aufruf zu generieren, der ctypes nicht benutzt. Damit könnte man dann portable Bindings schreiben, die überall funktionieren, und bei Nuitka ganz rasant wären.

Nuitka das Projekt - Status (4 von 5)

6. hints Modul

Gibt es noch nicht. 

Sollte unter CPython auch prüfen und Fehler geben, genauso wie unter Nuitka. Im Idealfall erlauben die Prüfungen Nuitka einfach nur zu erkennen, was sie tun, und daraus zu folgern, was allerdings zu ambitioniert sein könnte.





Es wäre auch toll, wenn man da eine gemeinsame Basis finden könnte mit anderen Projekten.

Nuitka das Projekt - Status (5 von 5)

- Nuitka läuft unter:
 - Linux x86/x64
 - Linux ARM
 - Crosscompile nach Windows
 - Windows nativ mit MinGW
- Nuitka braucht:
 - Python 2.6 oder 2.7
 - g++ 4.5 oder g++ 4.6

Nuitka das Projekt - Aktivitäten

Aktuelle:

- Schöneres README, richtiges User Manual als PDF 
- Speedcenter revival auf nuitka.net 
- CPython2.6 tests als git repository mit beschreibenden commits per diff 
- XML Regression Tests 

Dieses Jahr vielleicht:

- Funktionsaufrufe Parameter-Parsen in Programmen wegoptimieren
- CPython2.7, PyPy, ShedSkin tests als git repo....

Nuitka - XML Dump

Python:

```
tryBreakFinallyTest()
```

Auszug aus dem XML-Dump:

```
<node line="27" kind="StatementExpressionOnly">  
  <role name="expression">  
    <node line="27" kind="ExpressionFunctionCall">  
      <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 - Lektionen

```
assert type(a) is float
a == a                                # wie kann das falsch sein?
```

Gleichheit von Referenzen mit sich selbst ist nicht selbstverständlich

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

Verschachtelte Comprehensions sind erlaubt.

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

Yield ist wirklich eine Expression, die Werte ausser None liefern kann.

Nuitka das Projekt - Mitmachen

Sehr gerne. 😊

Ich akzeptiere Patches als...

- was immer "diff -u" brauchbares ausspuckt
- git formatierte "patch queues"
- git pull Requests.

Ich mache die Integrationsarbeit. Ob auf "master" oder "develop" basierend, ich werde ihre Arbeit integrieren und Ihnen zuschreiben.

Es gibt eine Mailingliste [nuitka-dev](#) auf der ich die meisten Bekanntmachungen mache.

Und es gibt die RSS Feeds auf <http://nuitka.net>, wo ich das wichtigste aufschreibe.

Nuitka das Projekt - Lizenz

Noch will ich Kontrolle über Nuitka, deshalb GPLv3 mit Copyright-Assignment. Was ich später machen möchte, ist mir noch unklar.

- Meine Entscheidung
- So ist Nuitka für Closed Source nicht tauglich

Mir erstmal egal, soll aber kein Dauerzustand werden.

Diskussion

- Ich stehe noch während der gesamten PyCon-DE zur Verfügung für Fragen und Anregungen. Fragen gerne an meine Email kayhayen@gmx.de oder auf der Mailingliste.
- Insbesondere hoffe ich:
 1. Jemand zeigt mir vielleicht, wie ich die *PyPy Tests* benutzen kann. Der Test-Runner scheint PyPy spezifisch zu sein.
 2. Ein *kritisches* Review vom Nuitka Quellcode, seinem Design würde mich freuen.
 3. Ideen von C++ Leuten, wie Nuitka besseren C++ Code generieren könnte.

Diese Präsentation

- Erstellt mit [rst2pdf](#)
- Download des PDF <http://nuitka.net/pr/Nuitka-Presentation-PyCON-DE-2011.pdf>
- Diagramme mit [OOo Draw](#) erstellt
- Icons von [visualpharm.com](#) (Lizenz verlangt Link)
- Für den Vortrag auf der [PyCon DE](#)