numba Documentation

Release tag: 0.11.1

Continuum Analytics (2012)

March 04, 2014

CONTENTS

1 Installation **3**
- 1.1 Use Anaconda . . . 3
- 1.2 Install from Source . . . 3

2 Quick Start **5**

3 Types **9**
- 3.1 Basic Types . . . 9
- 3.2 More Complex Types . . . 11

4 Arrays **13**
- 4.1 Array Creation & Loop-Jitting . . . 13
- 4.2 Loop Jitting Constraints . . . 14

5 UFuncs **19**
- 5.1 Ufuncs . . . 19
- 5.2 Generalized Ufuncs . . . 20

6 Examples **23**
- 6.1 A Simple Function . . . 23
- 6.2 Objects . . . 24
- 6.3 UFuncs . . . 24
- 6.4 Mandelbrot . . . 26
- 6.5 Filterbank Correlation . . . 27
- 6.6 Multi threading . . . 28

7 First Steps with numba **31**
- 7.1 Introduction to numba . . . 31
- 7.2 A simple example . . . 31
- 7.3 Compiling a function with numba.jit using an explicit function signature . . . 32
- 7.4 Signature . . . 33
- 7.5 Compiling a function without providing a function signature (autojit functionality) . . . 34
- 7.6 Some extra remarks . . . 34

8 Numba and types **37**
- 8.1 Introduction . . . 37
- 8.2 Type inference by example . . . 37
- 8.3 Supported types in *numba* . . . 41
- 8.4 Notes about changes in this tutorial . . . 44

9 Interfacing with C 47
9.1 CFFI . . . 47
9.2 ctypes . . . 47

10 Static Compilation (pycc) 49

11 Numba Debugging Tips 51

12 Release Notes 53
12.1 Version 0.12.2 . . . 53
12.2 Version 0.12.1 . . . 53
12.3 Version 0.12 . . . 53
12.4 Version 0.11 . . . 54
12.5 Version 0.10 . . . 54
12.6 Version 0.9 . . . 55
12.7 Version 0.8 . . . 55
12.8 Version 0.7.2 . . . 55
12.9 Version 0.7.1 . . . 55
12.10 Version 0.7 . . . 55
12.11 Version 0.6.1 . . . 56
12.12 Version 0.6 . . . 56
12.13 Version 0.5 . . . 56
12.14 Version 0.4 . . . 57
12.15 Version 0.3.2 . . . 57
12.16 Version 0.3 . . . 57
12.17 Version 0.2 . . . 57

13 Numba Language Specification 59
13.1 Native Types . . . 59
13.2 Boxed Types . . . 59
13.3 Values . . . 60
13.4 Control Flow . . . 60
13.5 Introspection . . . 61
13.6 Length . . . 61
13.7 Destruction . . . 61
13.8 Metaprogramming . . . 61
13.9 Pass . . . 61
13.10 System IO . . . 61
13.11 Formatting . . . 61
13.12 Iterators . . . 61
13.13 Comprehensions . . . 62
13.14 Builtins . . . 62
13.15 Slice . . . 64
13.16 Classes . . . 64
13.17 Casts . . . 65
13.18 Characters . . . 65
13.19 Closures . . . 65
13.20 Globals . . . 65
13.21 Arguments . . . 65
13.22 Assertions . . . 65
13.23 Operators . . . 65
13.24 Division . . . 66
13.25 Math Functions . . . 66
13.26 Floating Point Math . . . 67
13.27 Complex Math . . . 68

14 Numba Architecture **69**
14.1 Introduction . . . 69
14.2 Core Entry Points . . . 69
14.3 Translation Internals . . . 70
14.4 Terms and Definitions . . . 72
14.5 Appendix . . . 72

15 Numba Intermediate Representation Specification **73**
15.1 Numba IR Stages . . . 73
15.2 Numba Intermediate Representations . . . 74
15.3 Appendicies . . . 77

16 Numba Roadmap **85**
16.1 Numba Versions . . . 85
16.2 Thing we want . . . 86
16.3 Less intricate . . . 86
16.4 More intricate . . . 88

17 Development crash course **91**
17.1 Overview . . . 91

User Guide

CHAPTER

ONE

INSTALLATION

1.1 Use Anaconda

The easiest way to install numba and get updates is by using the Anaconda Distribution:

```
$ conda install numba
```

1.2 Install from Source

Numba main dependency is NumPy, LLVM and llvmpy. Please refer to http://www.llvmpy.org/ for instructions on how to install LLVM and llvmpy. Note that Numba now depends on LLVM 3.3.

1.2.1 Dependencies

- LLVM 3.3
- llvmpy (from llvmpy/llvmpy fork)
- numpy (version 1.6 or higher)
- argparse (for pycc in python2.6)

CHAPTER

TWO

QUICK START

Numba compiles Python code to LLVM IR which can be natively executed at runtime much faster than pure Python code. The first step to using Numba is becoming familiar with the jit decorator, which tells Numba which functions to compile:

```
from numba import jit

@jit
def sum(x, y):
    return x + y
```

The very basic example above is compiled for any compatible input types automatically when the sum1d function is called. The result is a new function with performance comparable to a compiled function written in C (assuming best case scenario; more on that later). To compile for specific input types, we can tell Numba what those input types are:

```
@jit('f8(f8)')
def sum(x, y):
    return x + y
```

The string above passed to the jit decorator tells Numba the return type is an 8 byte float, and the single argument passed in is also an 8 byte float. The string takes the form 'returntype(arg1type, arg2type, ...)'.

One of the main features of Numba is it's support for NumPy arrays. The following example shows how a function can be compiled that takes a NumPy array of floats as an input:

```
@jit('f8(f8[:])')
def sum1d(array):
    sum = 0.0
    for i in range(array.shape[0]):
        sum += array[i]
    return sum
```

There are two main things to notice in the example above. The input argument is specified by the string 'f8[:]', which means a 1d array of 8 byte floats. A 2d array would be specified as 'f8[:, :]', a 3d array as 'f8[:, :, :]', and so on. The other thing to take note of is the array indexing and shape method call, and the fact that we're iterating over a NumPy array using Python. Normally this would be terribly slow and would be cause for writing a NumPy ufunc in C, but the performance of the code above is the same as NumPy's sum method.

Numba can also infer the array type automatically like other elementary types:

```
@jit
def sum1d(array)
    ...
```

Numba's elementary built in types in are summarized in the table below and can be found in the `numba` namespace.

Type Name	Alias	Result Type
boolean	b1	uint8 (char)
bool_	b1	uint8 (char)
byte	u1	unsigned char
uint8	u1	uint8 (char)
uint16	u2	uint16
uint32	u4	uint32
uint64	u8	uint64
char	i1	signed char
int8	i1	int8 (char)
int16	i2	int16
int32	i4	int32
int64	i8	int64
float_	f4	float32
float32	f4	float32
double	f8	float64
float64	f8	float64
complex64	c8	float complex
complex128	c16	double complex

Native platform-dependent types are also available under names such as `int_`, `short`, `ulonglong`, etc.

Function signatures can also be expressed with the type objects directly as opposed to using strings. For example:

```
from numba import jit, f8

@jit(f8(f8[:]))
def sum1d(array):
    ...
```

In the example above, the argument type object is passed in to the return type object's constructor.

Numba attempts to compile everything down to LLVM IR, but in some cases this isn't (yet) possible. If Numba can't infer the type of a variable or doesn't support a particular data type, it falls back to using Python objects. This is of course much slower. If you're having performance issues and suspect Python objects are to blame, you can use the nopython flag to force Numba to abort if it can't avoid using Python objects:

```
@jit(nopython=True):
def sum1d(array):
    ...
```

Another useful debugging tool is Numba's new inspect_types method. This can be called for any Numba compiled function to get a listing of the Numba IR generated from the Python code as well as the inferred types of each variable:

```
>>> sum1d.inspect_types()
sum1d (array(float64, 1d, A),) -> float64
``-----------------------------------------------------------------------``
# --- LINE 5 ---

@jit('f8(f8[:])')

# --- LINE 6 ---

def sum1d(array):

    # --- LINE 7 ---
    # label 0
    #   $0.1 = const(<type 'float'>, 0.0)  :: float64
```

```
#   sum = $0.1  :: float64

sum = 0.0

# --- LINE 8 ---
#   jump 6
# label 6
#   $6.1 = global(range: <built-in function range>)  :: range
#   $6.2 = getattr(attr=shape, value=array)  :: (int64 x 1)
#   $6.3 = const(<type 'int'>, 0)  :: int32
#   $6.4 = getitem(index=$6.3, target=$6.2)  :: int64
#   $6.5 = call $6.1($6.4, )  :: (int64,) -> range_state64
#   $6.6 = getiter(value=$6.5)  :: range_iter64
#   jump 26
# label 26
#   $26.1 = iternext(value=$6.6)  :: int64
#   $26.2 = itervalid(value=$6.6)  :: bool
#   branch $26.2, 29, 50
# label 29
#   $29.1 = $26.1  :: int64
#   i = $29.1  :: int64

for i in range(array.shape[0]):

    # --- LINE 9 ---
    # label 49
    #   del $6.6
    #   $29.2 = getitem(index=i, target=array)  :: float64
    #   $29.3 = sum + $29.2  :: float64
    #   sum = $29.3  :: float64
    #   jump 26

    sum += array[i]

# --- LINE 10 ---
#   jump 50
# label 50
#   return sum

return sum
```

For get a better feel of what numba can do, see *Examples*.

CHAPTER

THREE

TYPES

3.1 Basic Types

The following table contains the elementary types currently defined by Numba.

Type Name	Alias	Result Type
boolean	b1	uint8 (char)
bool_	b1	uint8 (char)
byte	u1	unsigned char
uint8	u1	uint8 (char)
uint16	u2	uint16
uint32	u4	uint32
uint64	u8	uint64
char	i1	signed char
int8	i1	int8 (char)
int16	i2	int16
int32	i4	int32
int64	i8	int64
float_	f4	float32
float32	f4	float32
double	f8	float64
float64	f8	float64
complex64	c8	float complex
complex128	c16	double complex

Types can be used to specify the signature of a function:

```
@jit('f8(f8[:])')
def sum1d(array):
    sum = 0.0
    for i in range(array.shape[0]):
        sum += array[i]
    return sum
```

Types can also be used in Numba to declare local variables in a function:

```
@jit(locals=dict(array=double[:, :], scalar1=double))
def func(array):
    scalar1 = array[0, 0] # scalar is declared double
    scalar2 = double(array[0, 0])
```

Of course, declaring types in this example is unnecessary since the type inferencer knows the input type of `array`, and hence knows the type of `array[i, j]` to be the dtype of `array`.

Note: Type declarations or casts can be useful in cases where the type inferencer doesn't know the type, or if you want to override the type inferencer's rules (e.g. force 32-bit floating point precision).

Variables declared in the `locals` dict have a single type throughout the entire function. However, any variable not declared in locals can assume different types, just like in Python:

```
@jit
def variable_ressign(arg):
    arg = 1.0
    arg = "hello"
    arg = object()
    var = arg
    var = "world"
```

However, there are some restrictions, namely that variables must have a unifyable type at control flow merge points. For example, the following code will not compile:

```
@jit
def incompatible_types(arg):
    if arg > 10:
        x = 1+2j
    else:
        x = 3.3

    return x         # ERROR! Inconsistent type for x!
```

This code is invalid because strings and integers are not compatible. However, if we do not read `x` after the if block, the code will compile fine, since it does not need to unify the type:

```
@jit
def compatible_types(arg):
    if arg > 10:
        x = "hello"
    else:
        x = arg

    x = func()
    return x
```

The same goes for loop carried dependencies and variables escaping loops, e.g.:

```
@jit
def incompatible_types2(N):
    x = "hello"
    for i in range(N):
        print x      # ERROR! Inconsistent type for x!
        x = i

    return x

@jit
def incompatible_types3(N):
    x = "hello"
    for i in range(N):
        x = i
        print x

    return x         # ERROR! Inconsistent type for x if N <= 0
```

Cases where the type inferencer doesn't know the type is often when you call a Python function or method that is not a numba function and numba doesn't otherwise recognize.

Numba allows you to obtain the type of a expression or variable through the typeof function in a Numba function. This type can then be used for instance to cast other values:

```
type = numba.typeof(x + y)
value = type(value)
```

When used outside of a Numba function, it returns the type the type inferencer would infer for that value:

```
>>> numba.typeof(1.0)
double
>>> numba.typeof(cmath.sqrt(-1))
complex128
```

3.2 More Complex Types

Numba is in the process of being refactored to better define more complex types such as structs, pointers, strings and user defined classes. More on this soon...

CHAPTER

FOUR

ARRAYS

Support for NumPy arrays is a key focus of Numba development and is currently undergoing extensive refactorization and improvement. Most capabilities of NumPy arrays are supported by Numba in object mode, and a few features are supported in nopython mode too (with much more to come).

A few noteworthy limitations of arrays at this time:

- Arrays can be passed in to a function in nopython mode, but not returned. Arrays can only be returned in object mode.
- New arrays can only be created in object mode.
- Currently there are no bounds checking for array indexing and slicing, although negative indices will wrap around correctly.
- NumPy array ufunc support in nopython node is incomplete at this time. Most if not all ufuncs should work in object mode though.
- Array slicing only works in object mode.

4.1 Array Creation & Loop-Jitting

NumPy array creation is not supported in nopython mode. Numba mitigates this by automatically trying to jit loops in nopython mode. This allows for array creation at the top of a function while still getting almost all the performance of nopython mode. For example, the following simple function:

```
# compiled in object mode
@jit
def sum(x, y):
    array = np.arange(x * y).reshape(x, y)
    sum = 0
    for i in range(x):
        for j in range(y):
            sum += array[i, j]
    return sum
```

looks like the equivalent of the following after being compiled by Numba:

```
# compiled in nopython mode
@njit
def jitted_loop(array, x, y):
    sum = 0
    for i in range(x):
        for j in range(y):
```

```
            sum += array[i, j]
    return sum

# compiled in object mode
@jit
def sum(x, y):
    array = np.arange(x * y).reshape(x, y)
    return jitted_loop(array, x, y)
```

Another consequence of array creation being restricted to object mode is that NumPy ufuncs that return the result as a new array are not allowed in nopython mode. Fortunately we can declare an output array at the top of our function and pass that in to the ufunc to store our result. For example, the following:

```
@jit
def foo():
    # initialize stuff

    # slow loop in object mode
    for i in range(x):
        result = np.multiply(a, b)
```

should be rewritten like the following to take advantage of loop jitting:

```
@jit
def foo():
    # initialize stuff

    # create output array
    result = np.zeros(a.size)

    # jitted loop in nopython mode
    for i in range(x):
        np.multiply(a, b, result)
```

4.2 Loop Jitting Constraints

The current loop-jitting mechanism is very conservative. A loop must satisfy a set of constraints for loop-jitting to trigger. These constraints will be relaxed in further development.

Currently, a loop is rejected if:

- the loop contains return statements;
- the loop binds a value to a variable that is read outside of the loop.

The following is rejected due to a return statement in the loop:

```
@jit
def foo(n):
    result = np.zeros(n)

    # Rejected loop-jitting candidate
    for i in range(n):
        result[i] = i    # setitem is accepted
        if i > 10:
            return       # return is not accepted

    return result
```

The following is rejected due to an assigning to a variable read outside of the loop:

```
@jit
def foo(n):
    result = np.zeros(n)

    x = 1
    # Rejected loop-jitting candidate
    for i in range(n):
        x = result[i]           # assign to variable 'x'

    result += x                 # reading variable 'x'
    return result
```

The following is accepted:

```
@jit
def foo(n):
    result = np.zeros(n)
    x = 1
    # Accepted loop-jitting candidate
    for i in range(n):
        x = 2
    x = 3               # 'x' is only written to

    return result
```

The following is accepted:

```
@jit
def foo(n):
    result = np.zeros(n)
    x = 1
    # Accepted loop-jitting candidate
    for i in range(n):
        result[i] = x

    return result
```

User can inspect the loop-jitting by running *foo.inspect_types()*:

```
foo (int32,) -> pyobject
--------------------------------------------------------------------------------
# File: somefile.py
# --- LINE 1 ---

@jit

# --- LINE 2 ---

def foo(n):

    # --- LINE 3 ---
    # label 0
    #   $0.1 = global(numpy: <module 'numpy' from '.../numpy/__init__.py'>)
      :: pyobject
    #   $0.2 = getattr(value=$0.1, attr=zeros)  :: pyobject
    #   result = call $0.2(n, )  :: pyobject

    result = numpy.zeros(n)
```

```
    # --- LINE 4 ---
    #   x = const(<class 'int'>, 1)  :: pyobject

    x = 1

    # --- LINE 5 ---
    #   jump 58
    # label 58
    #   $58.1 = global(foo__numba__loop21__: LiftedLoop(<function foo at 0x107781710>))  :: pyobject
    #   $58.2 = call $58.1(n, result, x, )  :: pyobject
    #   jump 54

    for i in range(n):

        # --- LINE 6 ---

        result[i] = x

    # --- LINE 7 ---
    # label 54
    #   return result

    return result

# The function contains lifted loops
# Loop at line 5
# Has 1 overloads
# File: somefile.py
# --- LINE 1 ---

@jit

# --- LINE 2 ---

def foo(n):

    # --- LINE 3 ---

    result = numpy.zeros(n)

    # --- LINE 4 ---

    x = 1

    # --- LINE 5 ---
    # label 34
    #   $34.1 = iternext(value=$21.3)  :: int32
    #   $34.2 = itervalid(value=$21.3)  :: bool
    #   branch $34.2, 37, 53
    # label 21
    #   $21.1 = global(range: <class 'range'>)  :: range
    #   $21.2 = call $21.1(n, )  :: (int32,) -> range_state32
    #   $21.3 = getiter(value=$21.2)  :: range_iter32
    #   jump 34
    # label 37
    #   $37.1 = $34.1  :: int32
    #   i = $37.1  :: int32
```

```
    for i in range(n):

        # --- LINE 6 ---
        # label 53
        #   del $21.3
        #   jump 54
        # label 54
        #   $54.1 = const(<class 'NoneType'>, None)  :: none
        #   return $54.1
        #   result[i] = x  :: (array(float64, 1d, C), int64, float64) -> none
        #   jump 34

        result[i] = x

    # --- LINE 7 ---

    return result


================================================================================
```

CHAPTER

FIVE

UFUNCS

5.1 Ufuncs

Numba's vectorize allows Numba functions taking scalar input arguments to be used as NumPy ufuncs (see http://docs.scipy.org/doc/numpy/reference/ufuncs.html). Creating a traditional NumPy ufunc is not the most difficult task in the world, but it is also not the most straightforward process and involves writing some C code. Numba makes this easy though. Using the vectorize decorator, Numba can compile a Python function into a ufunc that operates over NumPy arrays as fast as traditional ufuncs written in C.

Ufunc arguments are scalars of a NumPy array. Function definitions can be arbitrary mathematical expressions. The vectorize decorator needs to know the argument and return types of the ufunc. These are specified much like the jit decorator:

```
import math

@vectorize(['float64(float64, float64)'])
def my_ufunc(x, y):
    return x+y+math.sqrt(x*math.cos(y))

a = np.arange(1.0, 10.0)
b = np.arange(1.0, 10.0)
# Calls compiled version of my_ufunc for each element of a and b
print(my_ufunc(a, b))
```

Multiple signatures can be specified to handle multiple input types:

```
@vectorize(['int32(int32, int32)',
            'float64(float64, float64)'])
def my_ufunc(x, y):
    return x+y+math.sqrt(x*math.cos(y))

a = np.arange(1.0, 10.0, dtype='f8')
b = np.arange(1.0, 10.0, dtype='f8')
print(my_ufunc(a, b))

a = np.arange(1, 10, dtype='i4')
b = np.arange(1, 10, dtype='i4')
print(my_ufunc(a, b))
```

The order of the signatures is important. Numba dispatches based on the input array types and uses the first ufunc signature that the input types can be safely cast to. In the example above, if the float64 signature had been listed first, the call to sum with int32 arrays would have produced a float64 array as the result.

An alternative syntax is to use the UFuncBuilder object to build a list of function signatures:

```
from numba.npyufunc.ufuncbuilder import UFuncBuilder

def my_ufunc(x, y):
    return x+y+math.sqrt(x*math.cos(y))

builder = UFuncBuilder(my_ufunc)
builder.add(restype=i4, argtypes=[i4, i4])
builder.add(restype=f8, argtypes=[f8, f8])
```

To compile our ufunc we call the build_ufunc method:

```
compiled_ufunc = builder.build_ufunc()

a = np.arange(1.0, 10.0, dtype='f8')
b = np.arange(1.0, 10.0, dtype='f8')
print(compiled_ufunc(a, b))
```

Since we defined a binary ufunc, we can use the various NumPy ufunc methods such as `reduce`, `accumulate`, etc:

```
a = np.arange(100)
print(compiled_ufunc.reduce(a))
print(compiled_ufunc.accumulate(a))
```

5.2 Generalized Ufuncs

Numba also provides support for generalized ufuncs with the guvectorize decorator. Traditional ufuncs perfom element-wise operations, whereas generalized ufuncs operate on entire sub-arrays. In addition to the argument and return types, the guvectorize decorator takes an additional signature which specifies the shapes of the inner arrays we want to operate on:

```
import math

@guvectorize(['void(int32[:,:], int32[:,:], int32[:,:])',
              'void(float64[:,:], float64[:,:], float64[:,:])'],
              '(x, y),(x, y)->(x, y)')
def my_gufunc(a, b, c):
    for i in range(c.shape[0]):
        for j in range(c.shape[1]):
            c[i, j] = a[i, j] + b[i, j]

a = np.arange(1.0, 10.0, dtype='f8').reshape(3,3)
b = np.arange(1.0, 10.0, dtype='f8').reshape(3,3)
# Calls compiled version of my_gufunc for each row of a and b
print(my_gufunc(a, b))
```

Notice that we don't have a third argument in the gufunc call but the generalized ufunc definition above has three arguments. The last argument of the generalized ufunc is the output, which is automatically allocated with the shape specified in the signature.

Generalized ufuncs also have an alternative syntax. We can use the GUFuncBuilder object to build a list of function signatures and specify the shape of the arguments:

```
from numba.npyufunc.ufuncbuilder import GUFuncBuilder

def my_gufunc(a, b, c):
    for i in range(c.shape[0]):
        for j in range(c.shape[1]):
```

```
            c[i, j] = a[i, j] + b[i, j]

builder = GUFuncBuilder(my_ufunc, '(x, y),(x, y)->(x, y)')
builder.add('void(int32[:,:], int32[:,:], int32[:,:])')
builder.add('void(float64[:,:], float64[:,:], float64[:,:])')
```

To compile our ufunc we call the build_ufunc method:

```
compiled_gufunc = builder.build_ufunc()

a = np.arange(1.0, 10.0, dtype='f8').reshape(3,3)
b = np.arange(1.0, 10.0, dtype='f8').reshape(3,3)
print(my_gufunc(a, b))
```

CHAPTER

SIX

EXAMPLES

6.1 A Simple Function

Suppose we want to write an image-processing function in Python. Here's how it might look.

```
import numpy

def filter2d(image, filt):
    M, N = image.shape
    Mf, Nf = filt.shape
    Mf2 = Mf // 2
    Nf2 = Nf // 2
    result = numpy.zeros_like(image)
    for i in range(Mf2, M - Mf2):
        for j in range(Nf2, N - Nf2):
            num = 0.0
            for ii in range(Mf):
                for jj in range(Nf):
                    num += (filt[Mf-1-ii, Nf-1-jj] * image[i-Mf2+ii, j-Nf2+jj])
            result[i, j] = num
    return result

# This kind of quadruply-nested for-loop is going to be quite slow.
# Using Numba we can compile this code to LLVM which then gets
# compiled to machine code:

from numba import double, jit

fastfilter_2d = jit(double[:,:](double[:,:], double[:,:]))(filter2d)

# Now fastfilter_2d runs at speeds as if you had first translated
# it to C, compiled the code and wrapped it with Python
image = numpy.random.random((100, 100))
filt = numpy.random.random((10, 10))
res = fastfilter_2d(image, filt)
```

Numba actually produces two functions. The first function is the low-level compiled version of filter2d. The second function is the Python wrapper to that low-level function so that the function can be called from Python. The first function can be called from other numba functions to eliminate all python overhead in function calling.

6.2 Objects

```
# -*- coding: utf-8 -*-
from __future__ import print_function, division, absolute_import
from numba import jit

class MyClass(object):
    def mymethod(self, arg):
        return arg * 2

@jit
def call_method(obj):
    print(obj.mymethod("hello"))  # object result
    mydouble = obj.mymethod(10.2) # native double
    print(mydouble * 2)           # native multiplication

call_method(MyClass())
```

6.3 UFuncs

```
from numba import vectorize
from numba import autojit, double, jit
import math
import numpy as np

@vectorize(['f8(f8)','f4(f4)'])
def sinc(x):
    if x == 0:
        return 1.0
    else:
        return math.sin(x*math.pi) / (x*math.pi)

@vectorize(['int8(int8,int8)',
            'int16(int16,int16)',
            'int32(int32,int32)',
            'int64(int64,int64)',
            'f4(f4,f4)',
            'f8(f8,f8)'])
def add(x,y):
    return x + y

@vectorize(['f8(f8)','f4(f4)'])
def logit(x):
    return math.log(x / (1-x))

@vectorize(['f8(f8)','f4(f4)'])
def expit(x):
    if x > 0:
        x = math.exp(x)
        return x / (1 + x)
    else:
        return 1 / (1 + math.exp(-x))

@jit('f8(f8,f8[:])')
def polevl(x, coef):
```

```
    N = len(coef)
    ans = coef[0]
    i = 1
    while i < N:
        ans = ans * x + coef[i]
        i += 1
    return ans

@jit('f8(f8,f8[:])')
def p1evl(x, coef):
    N = len(coef)
    ans = x + coef[0]
    i = 1
    while i < N:
        ans = ans * x + coef[i]
        i += 1
    return ans


PP = np.array([
  7.96936729297347051624E-4,
  8.28352392107440799803E-2,
  1.23953371646414299388E0,
  5.44725003058768775090E0,
  8.74716500199817011941E0,
  5.30324038235394892183E0,
  9.99999999999999997821E-1], 'd')

PQ = np.array([
  9.24408810558863637013E-4,
  8.56288474354474431428E-2,
  1.25352743901058953537E0,
  5.47097740330417105182E0,
  8.76190883237069594232E0,
  5.30605288235394617618E0,
  1.00000000000000000218E0], 'd')

DR1 = 5.783185962946784521175995758455807035071
DR2 = 30.47126234366208639907816317502275584842

RP = np.array([
-4.79443220978201773821E9,
 1.95617491946556577543E12,
-2.49248344360967716204E14,
 9.70862251047306323952E15], 'd')

RQ = np.array([
    # 1.00000000000000000000E0,
 4.99563147152651017219E2,
 1.73785401676374683123E5,
 4.84409658339962045305E7,
 1.11855537045356834862E10,
 2.11277520115489217587E12,
 3.10518229857422583814E14,
 3.18121955943204943306E16,
 1.71086294081043136091E18], 'd')

QP = np.array([
```

```
-1.13663838898469149931E-2,
-1.28252718670509318512E0,
-1.95539544257735972385E1,
-9.32060152123768231369E1,
-1.77681167980488050595E2,
-1.47077505154951170175E2,
-5.14105326766599330220E1,
-6.05014350600728481186E0], 'd')

QQ = np.array([
    # 1.00000000000000000000E0,
  6.43178256118178023184E1,
  8.56430025976980587198E2,
  3.88240183605401609683E3,
  7.24046774195652478189E3,
  5.93072701187316984827E3,
  2.06209331660327847417E3,
  2.42005740240291393179E2], 'd')

NPY_PI_4 = .78539816339744830962
SQ2OPI   = .79788456080286535587989

@jit('f8(f8)')
def j0(x):
    if (x < 0):
        x = -x

    if (x <= 5.0):
        z = x * x
        if (x < 1.0e-5):
            return (1.0 - z / 4.0)
        p = (z-DR1) * (z-DR2)
        p = p * polevl(z, RP) / polevl(z, RQ)
        return p

    w = 5.0 / x
    q = 25.0 / (x*x)
    p = polevl(q, PP) / polevl(q, PQ)
    q = polevl(q, QP) / p1evl(q, QQ)
    xn = x - NPY_PI_4
    p = p*math.cos(xn) - w * q * math.sin(xn)
    return p * SQ2OPI / math.sqrt(x)


x = np.arange(10000, dtype='i8')
y = np.arange(10000, dtype='i8')
print(sum(x, y))
```

6.4 Mandelbrot

```
# -*- coding: utf-8 -*-
from __future__ import print_function, division, absolute_import
from numba import jit
import numpy as np
from pylab import imshow, jet, show, ion
```

```
@jit
def mandel(x, y, max_iters):
    """
    Given the real and imaginary parts of a complex number,
    determine if it is a candidate for membership in the Mandelbrot
    set given a fixed number of iterations.
    """
    i = 0
    c = complex(x,y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 4:
            return i

    return 255

@jit
def create_fractal(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            image[y, x] = color

    return image

image = np.zeros((500, 750), dtype=np.uint8)
imshow(create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20))
jet()
ion()
show()
```

6.5 Filterbank Correlation

```
# -*- coding: utf-8 -*-

"""
This file demonstrates a filterbank correlation loop.
"""
from __future__ import print_function, division, absolute_import
import numpy as np
import numba
from numba.utils import IS_PY3
from numba.decorators import jit
nd4type = numba.double[:,:,:,:]

if IS_PY3:
    xrange = range
```

```python
@jit(argtypes=(nd4type, nd4type, nd4type))
def fbcorr(imgs, filters, output):
    n_imgs, n_rows, n_cols, n_channels = imgs.shape
    n_filters, height, width, n_ch2 = filters.shape

    for ii in range(n_imgs):
        for rr in range(n_rows - height + 1):
            for cc in range(n_cols - width + 1):
                for hh in xrange(height):
                    for ww in xrange(width):
                        for jj in range(n_channels):
                            for ff in range(n_filters):
                                imgval = imgs[ii, rr + hh, cc + ww, jj]
                                filterval = filters[ff, hh, ww, jj]
                                output[ii, ff, rr, cc] += imgval * filterval

def main ():
    imgs = np.random.randn(10, 64, 64, 3)
    filt = np.random.randn(6, 5, 5, 3)
    output = np.zeros((10, 60, 60, 6))

    import time
    t0 = time.time()
    fbcorr(imgs, filt, output)
    print(time.time() - t0)

if __name__ == "__main__":
    main()
```

6.6 Multi threading

```python
# -*- coding: utf-8 -*-
"""
Example of multithreading by releasing the GIL through ctypes.
"""
from __future__ import print_function, division, absolute_import

from timeit import repeat
import threading
from ctypes import pythonapi, c_void_p
from math import exp

import numpy as np
from numba import jit, void, double

nthreads = 2
size = 1e6

def timefunc(correct, s, func, *args, **kwargs):
    print(s.ljust(20), end=" ")
    # Make sure the function is compiled before we start the benchmark
    res = func(*args, **kwargs)
    if correct is not None:
        assert np.allclose(res, correct)
    # time it
    print('{:>5.0f} ms'.format(min(repeat(lambda: func(*args, **kwargs),
```

```
                                              number=5, repeat=2)) * 1000))
    return res

def make_singlethread(inner_func):
    def func(*args):
        length = len(args[0])
        result = np.empty(length, dtype=np.float64)
        inner_func(result, *args)
        return result
    return func

def make_multithread(inner_func, numthreads):
    def func_mt(*args):
        length = len(args[0])
        result = np.empty(length, dtype=np.float64)
        args = (result,) + args
        chunklen = (length + 1) // numthreads
        chunks = [[arg[i * chunklen:(i + 1) * chunklen] for arg in args]
                  for i in range(numthreads)]

        # You should make sure inner_func is compiled at this point, because
        # the compilation must happen on the main thread. This is the case
        # in this example because we use jit().
        threads = [threading.Thread(target=inner_func, args=chunk)
                   for chunk in chunks[:-1]]
        for thread in threads:
            thread.start()

        # the main thread handles the last chunk
        inner_func(*chunks[-1])

        for thread in threads:
            thread.join()
        return result
    return func_mt

savethread = pythonapi.PyEval_SaveThread
savethread.argtypes = []
savethread.restype = c_void_p

restorethread = pythonapi.PyEval_RestoreThread
restorethread.argtypes = [c_void_p]
restorethread.restype = None

def inner_func(result, a, b):
    threadstate = savethread()
    for i in range(len(result)):
        result[i] = exp(2.1 * a[i] + 3.2 * b[i])
    restorethread(threadstate)

signature = void(double[:], double[:], double[:])
inner_func_nb = jit(signature, nopython=True)(inner_func)
func_nb = make_singlethread(inner_func_nb)
func_nb_mt = make_multithread(inner_func_nb, nthreads)

def func_np(a, b):
    return np.exp(2.1 * a + 3.2 * b)
```

```
a = np.random.rand(size)
b = np.random.rand(size)
c = np.random.rand(size)

correct = timefunc(None, "numpy (1 thread)", func_np, a, b)
timefunc(correct, "numba (1 thread)", func_nb, a, b)
timefunc(correct, "numba (%d threads)" % nthreads, func_nb_mt, a, b)
```

Tutorials

CHAPTER

SEVEN

FIRST STEPS WITH NUMBA

```
0.12.0
```

7.1 Introduction to numba

Numba allows the compilation of selected portions of Python code to native code, using llvm as its backend. This allows the selected functions to execute at a speed competitive with code generated by C compilers.

It works at the function level. We can take a function, generate native code for that function as well as the wrapper code needed to call it directly from Python. This compilation is done on-the-fly and in-memory.

In this notebook I will illustrate some very simple usage of numba.

7.2 A simple example

Let's start with a simple, yet time consuming function: a Python implementation of bubblesort. This bubblesort implementation works on a NumPy array.

Now, let's try the function, this way we check that it works. First we'll create an array of sorted values and randomly shuffle them:

Now we'll create a copy and do our bubble sort on the copy:

```
True
```

Let's see how it behaves in execution time:

```
1 loops, best of 3: 328 ms per loop
```

Note that as execution time may depend on its input and the function itself is destructive, I make sure to use the same input in all the timings, by copying the original shuffled array into the new one. %timeit makes several runs and takes the best result, if the copy wasn't done inside the timing code the vector would only be unsorted in the first iteration. As bubblesort works better on vectors that are already sorted, the next runs would be selected and we will get the time when running bubblesort in an already sorted array. In our case the copy time is minimal, though:

```
1000000 loops, best of 3: 1.17 µs per loop
```

7.3 Compiling a function with numba.jit using an explicit function signature

Let's get a numba version of this code running. One way to compile a function is by using the *numba.jit* decorator with an explicit signature. Later, we will see that we can get by without providing such a *signature* by letting *numba* figure out the *signatures* by itself. However, it is useful to know what the signature is, and what role it has in *numba*.

First, let's start by peeking at the numba.jit string-doc:

```
jit([signature_or_function, [locals={}, [target='cpu',
            [**targetoptions]]]])

    The function can be used as the following versions:

    1) jit(signature, [target='cpu', [**targetoptions]]) -> jit(function)

        Equivalent to:

            d = dispatcher(function, targetoptions)
            d.compile(signature)

        Create a dispatcher object for a python function and default
        target-options.  Then, compile the funciton with the given signature.

        Example:

            @jit("void(int32, float32)")
            def foo(x, y):
                return x + y

    2) jit(function) -> dispatcher

        Same as old autojit.  Create a dispatcher function object that
        specialize at call site.

        Example:

            @jit
            def foo(x, y):
                return x + y

    3) jit([target='cpu', [**targetoptions]]) -> configured_jit(function)

        Same as old autojit and 2).  But configure with target and default
        target-options.


        Example:

            @jit(target='cpu', nopython=True)
            def foo(x, y):
                return x + y

    Target Options
    ---------------
    The CPU (default target) defines the following:
```

```
- nopython: [bool]

    Set to True to disable the use of PyObjects and Python API
    calls.  The default behavior is to allow the use of PyObjects and
    Python API.  Default value is False.

- forceobj: [bool]

    Set to True to force the use of PyObjects for every value.  Default
    value is False.
```

So let's make a compiled version of our bubblesort:

At this point, **bubblesort_jit** contains the compiled function -wrapped so that is directly callable from Python- generated from the original bubblesort function. Note that there is a fancy parameter *"void(f4[:])"* that is passed. That parameter describes the *signature* of the function to generate (more on this later).

Let's check that it works:

```
True
```

Now let's compare the time it takes to execute the compiled function compared to the original

```
1000 loops, best of 3: 1.25 ms per loop
```

```
1 loops, best of 3: 323 ms per loop
```

Bear in mind that numba.jit is a decorator, although for practical reasons in this tutorial we will be calling it like a function to have access to both, the original function and the jitted one. In many practical uses, the decorator syntax may be more appropriate. With the decorator syntax our sample will look like this:

7.4 Signature

In order to generate fast code, the compiler needs type information for the code. This allows a direct mapping from the Python operations to the appropriate machine instruction without any type check/dispatch mechanism. In numba, in most cases it suffices to specify the types for the parameters. In many cases, numba can deduce types for intermediate values as well as the return value using *type inference*. For convenience, it is also possible to specify in the signature the type of the *return value*

A *numba.jit* compiled function will only work when called with the right type of arguments (it may, however, perform some conversions on types that it considers equivalent).

A *signature* contains the return type as well as the argument types. One way to specify the signature is using a string, like in our example. The *signature* takes the form: `<return type> ( <arg1 type>, <arg2 type>, ... )`. The types may be scalars or arrays (NumPy arrays). In our example, `void(f4[:])`, it means a function with no return (return type is `void`) that takes as unique argument an one-dimensional array of 4 byte floats `f4[:]`. Starting with numba version 0.12 the result type is optional. In that case the signature will look like the following: `<arg1 type>, <arg2 type>, ...`. When the signature doesn't provide a type for the return value, the type is *inferred*.

One way to specify the signature is by using such a string, the type for each argument being based on NumPy dtype strings for base types. Array types are also supported by using [:] type notation, where [:] is a one-dimensional strided array, [::1] is a one-dimensional contiguous array, [:,:] a bidimensional strided array, [:,:,:] a tridimiensional array, and so on. There are other ways to build the signature, you can find more details on signatures in its documentation page.

Some sample signatures follow:

signature	meaning
`void(f4[:], u8)`	a function with no return value taking a one-dimensional array of single precision floats and a 64-bit unsigned integer.
`i4(f8)`	a function returning a 32-bit signed integer taking a double precision float as argument.
`void(f4[:,:],f4[:,:])`	a function with no return value taking two 2-dimensional arrays as arguments.

For a more in-depth explanation on supported types you can take a look at the "Numba types" notebook tutorial.

7.5 Compiling a function without providing a function signature (autojit functionality)

Starting with numba version 0.12, it is possible to use *numba.jit* without providing a type-signature for the function. This functionality was provided by *numba.autojit* in previous versions of *numba*. The old *numba.autojit* hass been deprecated in favour of this signature-less version of *numba.jit*.

When no *type-signature* is provided, the decorator returns wrapper code that will automatically create and run a *numba* compiled version when called. When called, resulting function will infer the types of the arguments being used. That information will be used to generated the *signature* to be used when compiling. The resulting compiled function will be called with the provided arguments.

For performance reasons, functions are cached so that code is only compiled once for a given signature. It is possible to call the function with different signatures, in that case, different native code will be generated and the right version will be chosen based on the argument types.

For most uses, using jit without a signature will be the simplest option.

```
1000 loops, best of 3: 1.25 ms per loop
```

7.6 Some extra remarks

There is no magic, there are several details that is good to know about numba.

First, compiling takes time. Luckily enough it will not be a lot of time, specially for small functions. But when compiling many functions with many specializations the time may add up. Numba tries to do its best by caching compilation as much as possible though, so no time is spent in spurious compilation. It does its best to be *lazy* regarding compilation, this allows not paying the compilation time for code that is not used.

Second, not all code is compiled equal. There will be code that *numba* compiles down to an efficient native function. Sometimes the code generated has to fallback to the Python object system and its dispatch semantics. Other code may not compile at all.

When targeting the "cpu" target (the default), *numba* will either generate:

- Fast native code -also called 'nopython'-. The compiler was able to infer all the types in the function, so it can translate the code to a fast native routine without making use of the Python runtime.
- Native code with calls to the Python run-time -also called object mode-. The compiler was not able to infer all the types, so that at some point a value was typed as a generic 'object'. This means the full native version can't be used. Instead, numba generates code using the Python run-time that should be faster than actual interpretation but quite far from what you could expect from a full native function.

By default, the 'cpu' target tries to compile the function in 'nopython' mode. If this fails, it tries again in object mode.

This example shows how falling back to Python objects may cause a slowdown in the generated code:

```
10000 loops, best of 3: 31.9 µs per loop
```

```
1 loops, best of 3: 283 ms per loop
```

It is possible to force a failure if the *nopython* code generation fails. This allows getting some feedback about whether it is possible to generate code for a given function that doesn't rely on the Python run-time. This can help when trying to write fast code, as object mode can have a huge performance penalty.

On the other hand, *test2* fails if we pass the *nopython* keyword:

```
---------------------------------------------------------------------------
TypingError                               Traceback (most recent call last)

<ipython-input-19-6038b783c49c> in <module>()
----> 1 @numba.jit("void(i1[:])", nopython=True)
      2 def test2(value):
      3     for i in xrange(len(value)):
      4         value[i] = i % Decimal(100)


/Users/jayvius/Projects/numba/numba/decorators.pyc in wrapper(func)
    125         disp = dispatcher(py_func=func,  locals=locals,
    126                           targetoptions=targetoptions)
--> 127         disp.compile(sig)
    128         disp.disable_compile()
    129         return disp


/Users/jayvius/Projects/numba/numba/dispatcher.pyc in compile(self, sig, locals, **targetoptions)
    107             cres = compiler.compile_extra(typingctx, targetctx, self.py_func,
    108                                           args=args, return_type=return_type,
--> 109                                           flags=flags, locals=locs)
    110
    111             # Check typing error if object mode is used


/Users/jayvius/Projects/numba/numba/compiler.pyc in compile_extra(typingctx, targetctx, func, args, r
     77                                                                           args,
     78                                                                           return_type,
---> 79                                                                           locals)
     80         except Exception as e:
     81             if not flags.enable_pyobject:


/Users/jayvius/Projects/numba/numba/compiler.pyc in type_inference_stage(typingctx, interp, args, ret
    156         infer.seed_type(k, v)
    157
--> 158     infer.build_constrain()
    159     infer.propagate()
    160     typemap, restype, calltypes = infer.unify()


/Users/jayvius/Projects/numba/numba/typeinfer.pyc in build_constrain(self)
    271         for blk in utils.dict_itervalues(self.blocks):
    272             for inst in blk.body:
--> 273                 self.constrain_statement(inst)
    274
    275     def propagate(self):
```

```
/Users/jayvius/Projects/numba/numba/typeinfer.pyc in constrain_statement(self, inst)
    368     def constrain_statement(self, inst):
    369         if isinstance(inst, ir.Assign):
--> 370             self.typeof_assign(inst)
    371         elif isinstance(inst, ir.SetItem):
    372             self.typeof_setitem(inst)


/Users/jayvius/Projects/numba/numba/typeinfer.pyc in typeof_assign(self, inst)
    390                                              src=value.name, loc=inst.loc))
    391         elif isinstance(value, ir.Global):
--> 392             self.typeof_global(inst, inst.target, value)
    393         elif isinstance(value, ir.Expr):
    394             self.typeof_expr(inst, inst.target, value)


/Users/jayvius/Projects/numba/numba/typeinfer.pyc in typeof_global(self, inst, target, gvar)
    470             except KeyError:
    471                 raise TypingError("Untyped global name '%s'" % gvar.name,
--> 472                                   loc=inst.loc)
    473             self.assumed_immutables.add(inst)
    474             self.typevars[target.name].lock(gvty)


TypingError: Untyped global name 'Decimal'
File "<ipython-input-19-6038b783c49c>", line 4

numba version: 0.12.0
NumPy version: 1.7.1
llvm version: 0.12.1
```

CHAPTER

EIGHT

NUMBA AND TYPES

8.1 Introduction

Numba translates *Python* code into fast executing native code. In order to generate fast native code, many dynamic features of *Python* need to be translated into static equivalents. This includes dynamic typing as well as polymorphism. The approach taken in *numba* is using *type inference* to generate *type information* for the code, so that it is possible to translate into native code. If all the values in a *numba* compiled function can be translated into native types, the resulting code will be competitive with that generated with a low level language.

The objective of *type inference* is assigning a *type* to every single value in the function. The *type* of a value can either be:

- *Implicit*, in the case of providing an object that will provide its *type*. This happens, for example, in literals.
- *Explicit*, in the case of the programmer explicitly writing the *type* of a given value. This happens, for example, when a signature is given to *numba.jit*. That signature explicitly *types* the arguments.
- *Inferred*, when the *type* is deduced from an operation and the types of the its operands. For example, inferring that the type of *a + b*, when *a* and *b* are of type *int* is going to be an *int*

Type inference is the process by which all the *types* that are neither *implicit* nor *explicit* are deduced.

8.2 Type inference by example

Let's take a very simple sample function to illustrate these concepts:

When translating to native code it is needed to provide *type information* for every value involved in the sample function. This will include:

- The *literals* **4** and **3j**. These two have an implicit type.
- The argument **n**. In the function, as is, it is yet untyped.
- Some intermediate values, like **tmp** and the **return value**. Their type is not known yet.

8.2.1 Finding out the *types* of values

You can use the function *numba.typeof* to find out the *numba type* associated to a value.

```
Get the type of a variable or value.

Used outside of Numba code, infers the type for the object.
```

Bear in mind that, when used from the *Python* interpreter, *numba.typeof* will return the *numba type* associated to the object passed as parameter. For example, let's try using it on the *literals* found in our sample function:

```
int32
```

```
complex128
```

Also note that the types of the results are *numba types*:

```
numba.types.Integer
```

As a note, when used inside *numba* compiled code, *numba.typeof* will return the type as inferred during *type inference*. This may be a more general *type* than the one which would be returned when evaluating using the *Python interpreter*.

8.2.2 *Type inference* in *numba.jit*

Let's illustrate how type inference works with *numba.jit*. In order to illustrate this, we will use the *inspect_types* method of a compiled function and prints information about the types being used while compiling. This will be the different native types when the function has been compiled successfully in *nopython* mode. If object mode has been used we will get plenty of *pyobjects*.

Note that *inspect_types* is new to *numba 0.12*. Note also that the behavior of object mode has changed quite a bit as well in this release.

```
jit_sample_1 (float64,) -> complex128
--------------------------------------------------------------------------------
# --- LINE 1 ---

def jit_sample_1(n):

    # --- LINE 2 ---
    # label 0
    #   $0.1 = const(<type 'int'>, 4)  :: int32
    #   $0.2 = n + $0.1  :: float64
    #   tmp = $0.2  :: float64

    tmp = n + 4;

    # --- LINE 3 ---
    #   $0.3 = const(<type 'complex'>, 3j)  :: complex128
    #   $0.4 = tmp + $0.3  :: complex128
    #   return $0.4

    return tmp + 3j;


================================================================================
```

The source code of the original function should be shown with lines annotated with the values involved in that lines with its type annotated following a couple of double periods. The form will look like "**value = expression** :: **type**".

In this case, the resulting function will get a float64 argument and return a complex128. The literal 4 will be of type int32 ($0.1), while the result of adding the argument (n) to that literal will be a float64 ($0.2). The variable in the source code named tmp will be just float64 (assigned from $0.2). In the same way we can trace the next expression and see how **tmp+3j** results in a complex128 value that will be used as return value. The values named _$0.*_ are intermmediate values for the expression, and do not have a named counterpart in the source code.

If we were in *object* mode we would get something quite different. In order to illustrate, let's add the *forceobj* keyword to *numba.jit*. This will force *numba* to use object mode when compiling. Usually you don't want to use *forceobj* as *object* mode is slower than *nopython* mode:

```
jit_sample_1 (pyobject,) -> pyobject
--------------------------------------------------------------------------------
# --- LINE 1 ---

def jit_sample_1(n):

    # --- LINE 2 ---
    # label 0
    #   $0.1 = const(<type 'int'>, 4)  :: pyobject
    #   tmp = n + $0.1  :: pyobject

    tmp = n + 4;

    # --- LINE 3 ---
    #   $0.3 = const(<type 'complex'>, 3j)  :: pyobject
    #   $0.4 = tmp + $0.3  :: pyobject
    #   return $0.4

    return tmp + 3j;


================================================================================
```

As can be seen, everything is now a *pyobject*. That means that the operations will be executed by the Python runtime in the generated code.

Going back to the *nopython* mode, we can see how changing the input types will produced a different annotation for the code (and result in different code generation):

```
jit_sample_1 (int8,) -> complex128
--------------------------------------------------------------------------------
# --- LINE 1 ---

def jit_sample_1(n):

    # --- LINE 2 ---
    # label 0
    #   $0.1 = const(<type 'int'>, 4)  :: int32
    #   $0.2 = n + $0.1  :: int64
    #   tmp = $0.2  :: int64

    tmp = n + 4;

    # --- LINE 3 ---
    #   $0.3 = const(<type 'complex'>, 3j)  :: complex128
    #   $0.4 = tmp + $0.3  :: complex128
    #   return $0.4

    return tmp + 3j;


================================================================================
```

In this case, the input is an int8, but tmp ends being and int64 as it is added to an int32. Note that integer overflow of int64 is not handled by *numba*. In case of overflow the int64 will wrap around in the same way that it would happen in C.

8.2.3 Providing hints to the type inferrer

In most cases, the type inferrer will provide a type for your code. However, sometimes you may want a given intermediate value to use a specific type. This can be achieved by using the *locals* keyword in *numba.jit*. In *locals* a dictionary can be passed that maps the name of different local variables to a numba type. The compiler will assign that type to that variable.

Let's make a version of out function where we force *tmp* to be a *float*:

```
jit_sample_1 (int8,) -> complex128
--------------------------------------------------------------------------------
# --- LINE 1 ---

def jit_sample_1(n):

    # --- LINE 2 ---
    # label 0
    #   $0.1 = const(<type 'int'>, 4)  :: int32
    #   $0.2 = n + $0.1  :: int64
    #   tmp = $0.2  :: float64

    tmp = n + 4;

    # --- LINE 3 ---
    #   $0.3 = const(<type 'complex'>, 3j)  :: complex128
    #   $0.4 = tmp + $0.3  :: complex128
    #   return $0.4

    return tmp + 3j;


================================================================================
```

Note that as of numba 0.12, any type inference or type hints are ignored if object mode ends being generated, as everything gets treated as an object using the python runtime. This behavior may change in future versions.

```
jit_sample_1 (pyobject,) -> pyobject
--------------------------------------------------------------------------------
# --- LINE 1 ---

def jit_sample_1(n):

    # --- LINE 2 ---
    # label 0
    #   $0.1 = const(<type 'int'>, 4)  :: pyobject
    #   tmp = n + $0.1  :: pyobject

    tmp = n + 4;

    # --- LINE 3 ---
    #   $0.3 = const(<type 'complex'>, 3j)  :: pyobject
    #   $0.4 = tmp + $0.3  :: pyobject
    #   return $0.4

    return tmp + 3j;


================================================================================
```

8.2.4 Importance of type inference

It must be emphasized how important it is type inference in *numba*. A function where type inference is unable to provide a specific type for a value (that is, any type other than the generic *pyobject*). Any function that has a value fallback to *pyobject* will force the numba compiler to use the object mode. Object mode is way less efficient thant the *nopython*.

It is possible to know if a *numba* compiled function has fallen back to object mode by calling *inspect_types* on it. If there are values typed as *pyobject* that means that the object mode was used to compile it.

8.3 Supported types in *numba*

Numba supports many different types. It also supports some composite types as well as structures. Starting with numba 0.12 there is a namespace for types (numba.types). The numba namespace also imports these types.

In this section you can find a set of basic types you can use in numba. Many of the types have a "short name" matching their equivalent NumPy dtype. The list is not exahustive.

8.3.1 Integral types

type

numba type

short name

python equivalent

boolean

numba.types.bool_

b1

bool

signed integer

numba.types.int_

int

signed integer (8 bit)

numba.types.int8

i1

signed integer (16 bit)

numba.types.int16

i2

signed integer (32 bit)

numba.types.int32

i4

signed integer (64 bit)

numba.types.int64

i8

unsigned integer

numba.types.uint

unsigned integer (16 bit)

numba.types.uint16

u2

unsigned integer (32 bit)

numba.types.uint32

u4

unsigned integer (64 bit)

numba.types.uint64

u8

8.3.2 Floating point types

type

numba type

short name

python equivalent

single precision floating point (32 bit)

numba.float32

f4

double precision floating point (64 bit)

numba.float64

f8

float

single precision complex (2 x 32 bit)

numba.complex64

c8

double precison complex (2 x 64 bit)

numba.complex128

c16

complex

8.3.3 Array types

Array types are supported. An array type is built from a base type, a number of dimensions and potentially a layout specification. Some examples follow:

A one-dimensional array of float32

```
array(float32, 1d, A)
```

```
array(float32, 1d, A)
```

A two dimensional array of integers

```
array(uint32, 2d, A)
```

```
array(int32, 2d, A)
```

A two dimensional array of type 'c8' (complex64) in C array order

```
array(complex64, 2d, C)
```

```
array(complex64, 2d, C)
```

A two dimensional array of type uint16 in FORTRAN array order

```
array(uint16, 2d, F)
```

```
array(uint16, 2d, F)
```

Notice that the arity of the dimensions is not part of the types, only the number of dimensions. In that sense, an array with a shape (4,4) has the same numba type as another array with a shape (10, 12)

```
True
```

8.3.4 Some extra types

A type signature for a function (also known as a *function prototype*) that returns a float64, taking a two dimensional float64 array as first argument and a float64 argument

```
float64(array(float64, 2d, A), float64)
```

As can be seen the signature is just a type specification. In many places that a *function signature* is expected a string can be used instead. That string is in fact evaluated inside the numba.types namespace in order to build the actual type. This allows specifying the types in a compact way (as there is no need to fully qualify the base types) without polluting the active namespace (as it would happen by adding a __from numba.types import *__.

In *numba* 0.12 this is performed by the *numba.sigutils.parse_signature* function. Note that this function is likely to change or move in next versions, as it is just an implementation detail, but it can be used to show how the string version matches the other one, while keeping the syn

```
float64(array(float64, 2d, A), float64)
```

A generic Python object

```
pyobject
```

8.4 Notes about changes in this tutorial

In *numba* 0.12 there have been internal changes that have made material previously found in this tutorial obsolete.

- Some of the types previously supported in the *numba* type system have been dropped to be handled as *pyobjects*.
- The numba command line tool is no longer supported, but its functionality to get insights on how type inference works is now present in the form of the *inspect_types* method in the generated jitted function. This method is used in this tutorials to illustrate type inference.
- In 0.12 the object mode of *numba* has been greatly modified. Before it was using a mix of Python run-time and native code. In 0.12 object mode forces all values into *pyobjects*. As conversion to a string forces *numba* into object mode, the approach used in the previous version of this tutorial to print from inside the compiled function is no longer useful, as it will not print the staticly inferred types.

A sample of the this last point follows:

```
arg n: int32
literal 4: int32
tmp: int32
literal 3j:complex128

complex128

arg n: int32
literal 4: int32
tmp: int32
literal 3j:complex128

complex128
```

As can be seen, in both cases, Python and numba.jit, the results are the same. This is because *numba.typeof* is being evaluated with using the Python run-time.

If we use the inspect_types method on the jitted version, we will see that everything is in fact a *pyobject*

```
old_style_sample (pyobject,) -> pyobject
--------------------------------------------------------------------------------
# --- LINE 1 ---

def old_style_sample(n):

    # --- LINE 2 ---
    # label 0
    #   $0.1 = global(print: <built-in function print>)  :: pyobject
    #   $0.2 = const(<type 'str'>, arg n: )  :: pyobject
    #   $0.3 = global(str: <type 'str'>)  :: pyobject
    #   $0.4 = global(numba: <module 'numba' from '/Users/jayvius/Projects/numba/numba/__init__.pyc'>
    #   $0.5 = getattr(attr=typeof, value=$0.4)  :: pyobject
    #   $0.6 = call $0.5(n, )  :: pyobject
    #   $0.7 = call $0.3($0.6, )  :: pyobject
    #   $0.8 = $0.2 + $0.7  :: pyobject
    #   $0.9 = call $0.1($0.8, )  :: pyobject

    print('arg n: '+ str(numba.typeof(n)))

    # --- LINE 3 ---
    #   $0.10 = global(print: <built-in function print>)  :: pyobject
    #   $0.11 = const(<type 'str'>, literal 4: )  :: pyobject
```

```
#   $0.12 = global(str: <type 'str'>)  :: pyobject
#   $0.13 = global(numba: <module 'numba' from '/Users/jayvius/Projects/numba/numba/__init__.pyc'
#   $0.14 = getattr(attr=typeof, value=$0.13)  :: pyobject
#   $0.15 = const(<type 'int'>, 4)  :: pyobject
#   $0.16 = call $0.14($0.15, )  :: pyobject
#   $0.17 = call $0.12($0.16, )  :: pyobject
#   $0.18 = $0.11 + $0.17  :: pyobject
#   $0.19 = call $0.10($0.18, )  :: pyobject

print('literal 4: ' + str(numba.typeof(4)))

# --- LINE 4 ---
#   $0.20 = const(<type 'int'>, 4)  :: pyobject
#   tmp = n + $0.20  :: pyobject

tmp = n + 4;

# --- LINE 5 ---
#   $0.22 = global(print: <built-in function print>)  :: pyobject
#   $0.23 = const(<type 'str'>, tmp: )  :: pyobject
#   $0.24 = global(str: <type 'str'>)  :: pyobject
#   $0.25 = global(numba: <module 'numba' from '/Users/jayvius/Projects/numba/numba/__init__.pyc'
#   $0.26 = getattr(attr=typeof, value=$0.25)  :: pyobject
#   $0.27 = call $0.26(tmp, )  :: pyobject
#   $0.28 = call $0.24($0.27, )  :: pyobject
#   $0.29 = $0.23 + $0.28  :: pyobject
#   $0.30 = call $0.22($0.29, )  :: pyobject

print('tmp: '+ str(numba.typeof(tmp)))

# --- LINE 6 ---
#   $0.31 = global(print: <built-in function print>)  :: pyobject
#   $0.32 = const(<type 'str'>, literal 3j:)  :: pyobject
#   $0.33 = global(str: <type 'str'>)  :: pyobject
#   $0.34 = global(numba: <module 'numba' from '/Users/jayvius/Projects/numba/numba/__init__.pyc'
#   $0.35 = getattr(attr=typeof, value=$0.34)  :: pyobject
#   $0.36 = const(<type 'complex'>, 3j)  :: pyobject
#   $0.37 = call $0.35($0.36, )  :: pyobject
#   $0.38 = call $0.33($0.37, )  :: pyobject
#   $0.39 = $0.32 + $0.38  :: pyobject
#   $0.40 = call $0.31($0.39, )  :: pyobject

print('literal 3j:' + str(numba.typeof(3j)))

# --- LINE 7 ---
#   $0.41 = const(<type 'complex'>, 3j)  :: pyobject
#   $0.42 = tmp + $0.41  :: pyobject
#   return $0.42

return tmp + 3j;
```

```
================================================================================
```

Even more illustrating would be if *locals* was used to type an intermediate value:

```
arg n: int32
literal 4: int32
tmp: int32
```

```
literal 3j:complex128
```

```
complex128
```

The result seems to imply that *tmp* appears as an int32, but in fact is a *pyobject* and the whole function is being evaluated using the python run-time. So it is actually showing evaluating *typeof* at the runtime on the run-time value of tmp, which happens to be a Python *int*, translated into an int32 by *numba.typeof*. This can also be seen in the dump caused by the call to inspect_types.

Interfacing with native code

CHAPTER

NINE

INTERFACING WITH C

Numba supports calling C functions through CFFI and ctypes.

9.1 CFFI

Numba supports calling C functions wrapped with CFFI:

```
from numba import jit
from cffi import FFI

ffi = FFI()
ffi.cdef('double sin(double x);')

# loads the entire C namespace
C = ffi.dlopen(None)
c_sin = C.sin

@jit(nopython=True)
def cffi_sin_example(x):
    return c_sin(x)
```

9.2 ctypes

Numba also supports calling C functions wrapped with ctypes:

```
# This example doesn't work on Windows platforms
from ctypes import *
from math import pi
from numba import jit, double

proc = CDLL(None)

c_sin = proc.sin
c_sin.argtypes = [c_double]
c_sin.restype = c_double

@jit
def use_c_sin(x):
    return c_sin(x)
```

```python
ctype_wrapping = CFUNCTYPE(c_double, c_double)(use_c_sin)

@jit
def use_ctype_wrapping(x):
    return ctype_wrapping(x)
```

Misc

CHAPTER

TEN

STATIC COMPILATION (PYCC)

`pycc` allows users to compile Numba functions into a shared library. The user writes the functions, exports them and the compiler will import the module, collect the exported functions and compile them to a shared library. Below is an example:

```python
from numba import *

def mult(a, b):
    return a * b

export('mult f8(f8, f8)')(mult)
exportmany(['multf f4(f4, f4)', 'multi i4(i4, i4)'])(mult)
export('multc c16(c16, c16)')(mult)
```

This defines a trivial function and exports four specializations under different names. The code can be compiled as follows:

```
pycc thefile.py
```

Which will create a pure shared library for your platform which can be linked against any other program. This is **not** a Python extension. You would have to use ctypes to load the code that is created. Multiple files may be given to compile them simultaneously into a shared library. Options exist to compile to native object files instead of a shared library, to emit LLVM code or to generate a C header file with function prototypes. For more information on the available command line options, see `pycc -h`.

CHAPTER

ELEVEN

NUMBA DEBUGGING TIPS

The most common problem users run into is slower than expected performance. Usually this is because Numba is having trouble inferring a type or doesn't have a compiled version of a function to call. An easy way to pinpoint the source of this problem is to use the nopython flag. By default, Numba tries to compile everything down to low level types, but if it runs into trouble it will fall back to using Python objects. Setting nopython=True in the jit decorator will tell Numba to never use objects, and instead bail out if it runs into trouble:

```
@jit(nopython=True)
def foo()
    ...
```

which will result in an error if Numba can't figure out the type of a variable or function.

Another more advanced method for figuring out what's going on is using the inspect_types method. Calling inspect_types for a function compiled with Numba like so:

```
@jit
def foo(x):
    return np.sin(x)

foo.inspect_types()
```

will result in output similar to the following:

```
foo (pyobject,) -> pyobject

--------------------------

# --- LINE 25 ---

@jit(nopython=False)

# --- LINE 26 ---

def foo(x):

    # --- LINE 27 ---
    # label 0
    #   $0.1 = global(np: <module 'numpy' from '/Users/jayvius/anaconda/envs/dev/lib/python2.7/site-p
    #   $0.2 = getattr(attr=sin, value=$0.1)  :: pyobject
    #   $0.3 = call $0.2(x, )  :: pyobject
    #   return $0.3

    return np.sin(x)
```

A few things to take note of here: First, every line of Python code is preceded by several lines of Numba IR code that gives a glimpse into what Numba is doing to your Python code behind the scenes. More helpful though, at the end of most lines there are type annotations that show how Numba is treating variables and function calls. In the example above, the 'pyobject' annotation indicates that Numba doesn't know about the np.sin function so it has to fall back to the Python object layer to call it.

CHAPTER

TWELVE

RELEASE NOTES

12.1 Version 0.12.2

Fixes:

- Improved NumPy ufunc support in nopython mode
- Misc bug fixes

12.2 Version 0.12.1

This version fixed many regressions reported by user for the 0.12 release. This release contains a new loop-lifting mechanism that specializes certains loop patterns for nopython mode compilation. This avoid direct support for heap-allocating and other very dynamic operations.

Improvements:

- Add loop-lifting–jit-ing loops in nopython for object mode code. This allows functions to allocate NumPy arrays and use Python objects, while the tight loops in the function can still be compiled in nopython mode. Any arrays that the tight loop uses should be created before the loop is entered.

Fixes:

- Add support for majority of "math" module functions
- Fix for...else handling
- Add support for builtin round()
- Fix tenary if...else support
- Revive "numba" script
- Fix problems with some boolean expressions
- Add support for more NumPy ufuncs

12.3 Version 0.12

Version 0.12 contains a big refactor of the compiler. The main objective for this refactor was to simplify the code base to create a better foundation for further work. A secondary objective was to improve the worst case performance to ensure that compiled functions in object mode never run slower than pure Python code (this was a problem in several cases with the old code base). This refactor is still a work in progress and further testing is needed.

Main improvements:

- Major refactor of compiler for performance and maintenance reasons
- Better fallback to object mode when native mode fails
- Improved worst case performance in object mode

The public interface of numba has been slightly changed. The idea is to make it cleaner and more rational:

- jit decorator has been modified, so that it can be called without a signature. When called without a signature, it behaves as the old autojit. Autojit has been deprecated in favour of this approach.
- Jitted functions can now be overloaded.
- Added a "njit" decorator that behaves like "jit" decorator with nopython=True.
- The numba.vectorize namespace is gone. The vectorize decorator will be in the main numba namespace.
- Added a guvectorize decorator in the main numba namespace. It is similiar to numba.vectorize, but takes a dimension signature. It generates gufuncs. This is a replacement for the GUVectorize gufunc factory which has been deprecated.

Main regressions (will be fixed in a future release):

- Creating new NumPy arrays is not supported in nopython mode
- Returning NumPy arrays is not supported in nopython mode
- NumPy array slicing is not supported in nopython mode
- lists and tuples are not supported in nopython mode
- string, datetime, cdecimal, and struct types are not implemented yet
- Extension types (classes) are not supported in nopython mode
- Closures are not supported
- Raise keyword is not supported
- Recursion is not support in nopython mode

12.4 Version 0.11

- Experimental support for NumPy datetime type

12.5 Version 0.10

- Annotation tool (./bin/numba –annotate –fancy) (thanks to Jay Bourque)
- Open sourced prange
- Support for raise statement
- Pluggable array representation
- Support for enumerate and zip (thanks to Eugene Toder)
- Better string formatting support (thanks to Eugene Toder)
- Builtins min(), max() and bool() (thanks to Eugene Toder)
- Fix some code reloading issues (thanks to Björn Linse)

- Recognize NumPy scalar objects (thanks to Björn Linse)

12.6 Version 0.9

- Improved math support
- Open sourced generalized ufuncs
- Improved array expressions

12.7 Version 0.8

- **Support for autojit classes**
  - Inheritance not yet supported
- Python 3 support for pycc
- **Allow retrieval of ctypes function wrapper**
  - And hence support retrieval of a pointer to the function
- Fixed a memory leak of array slicing views

12.8 Version 0.7.2

- Official Python 3 support (python 3.2 and 3.3)
- Support for intrinsics and instructions
- Various bug fixes (see https://github.com/numba/numba/issues?milestone=7&state=closed)

12.9 Version 0.7.1

- Various bug fixes

12.10 Version 0.7

- Open sourced single-threaded ufunc vectorizer
- Open sourced NumPy array expression compilation
- Open sourced fast NumPy array slicing
- Experimental Python 3 support
- **Support for typed containers**
  - typed lists and tuples
- Support for iteration over objects
- Support object comparisons

- **Preliminary CFFI support**
  - Jit calls to CFFI functions (passed into autojit functions)
  - TODO: Recognize ffi_lib.my_func attributes
- Improved support for ctypes
- Allow declaring extension attribute types as through class attributes
- **Support for type casting in Python**
  - Get the same semantics with or without numba compilation
- **Support for recursion**
  - For jit methods and extension classes
- Allow jit functions as C callbacks
- Friendlier error reporting
- Internal improvements
- A variety of bug fixes

12.11 Version 0.6.1

- Support for bitwise operations

12.12 Version 0.6

- Python 2.6 support
- **Programmable typing**
  - Allow users to add type inference for external code
- **Better NumPy type inference**
  - outer, inner, dot, vdot, tensordot, nonzero, where, binary ufuncs + methods (reduce, accumulate, reduceat, outer)
- **Type based alias analysis**
  - Support for strict aliasing
- Much faster autojit dispatch when calling from Python
- Faster numerical loops through data and stride pre-loading
- Integral overflow and underflow checking for conversions from objects
- Make Meta dependency optional

12.13 Version 0.5

- **SSA-based type inference**
  - Allows variable reuse

  - Allow referring to variables before lexical definition
- Support multiple comparisons
- Support for template types
- List comprehensions
- Support for pointers
- Many bug fixes
- Added user documentation

12.14 Version 0.4

12.15 Version 0.3.2

- Add support for object arithmetic (issue 56).
- Bug fixes (issue 55).

12.16 Version 0.3

- Changed default compilation approach to ast
- Added support for cross-module linking
- Added support for closures (can jit inner functions and return them) (see examples/closure.py)
- Added support for dtype structures (can access elements of structure with attribute access) (see examples/structures.py)
- Added support for extension types (numba classes) (see examples/numbaclasses.py)
- Added support for general Python code (use nopython to raise an error if Python C-API is used to avoid unexpected slowness because of lack of implementation defaulting to generic Python)
- Fixed many bugs
- Added support to detect math operations.
- Added with python and with nopython contexts
- Added more examples

Many features need to be documented still. Look at examples and tests for more information.

12.17 Version 0.2

- Added an ast approach to compilation
- Removed d, f, i, b from numba namespace (use f8, f4, i4, b1)
- Changed function to autojit2
- Added autojit function to decorate calls to the function and use types of the variable to create compiled versions.

- changed keyword arguments to jit and autojit functions to restype and argtypes to be consistent with ctypes module.
- Added pycc – a python to shared library compiler

Language Specification (outdated)

CHAPTER

THIRTEEN

NUMBA LANGUAGE SPECIFICATION

This document attempts to specify the Python subset supported by the numba compiler, and attempts to clarify which constructs are supported natively without help of the object layer.

13.1 Native Types

- bool
- char
- int
- float
- complex
- string [2]
- arrays [1], [2] and array expressions
- extension types [1]
- numba functions
- Ctypes/CFFI functions
- pointers
- structs

Note: [1] with reference counting

Note: [2] indexing, slicing and len()

13.2 Boxed Types

The following types are currently boxed in PyObjects:

- unicode
- list

- tuple
- set
- dict
- object
- frozenset
- buffer
- bytearray
- bytes
- memoryview

All operations on these types go through the object layer.

13.3 Values

Tuple unpacking is supported for:

- arrays of known size, e.g. `m, n = array.shape`
- syntactic assignment, e.g. `x, y = a, b`

Anything else goes through the object layer.

13.4 Control Flow

Supported:

- If
- If/Else
- If/ElseIf/Else
- For
- For/Else
- While
- While/Else
- Return
- Raise

Not Supported:

- Generators
- Try
- Try/Finally
- Try/Except
- Try/Except/Finally

13.5 Introspection

Runtime introspection with `type`, `isinstance`, `issubclass`, `id`, `dir`, `callable`, `getattr`, `hash`, `hasattr`, `super` and `vars` are supported only through the object layer.

`globals`, `locals` are not supported.

13.6 Length

The implementation of the `len` function is polymorphic and container specific.

`len :: [a] -> int`

13.7 Destruction

Variable and element destruction is not supported. The `del` operator is not part of the syntax and ‘‘delattr‘ is not supported.

13.8 Metaprogramming

`compile`, `eval` and `exec`, `execfile` are not supported

13.9 Pass

Pass is ignored.

13.10 System IO

`file`, `open` and `quit`, `raw_input`, `reload`, `help` and `input` are are supported only through the object layer.

`print` is supported through the object layer (default) or through `printf` (nopython mode).

13.11 Formatting

String formatting is supported through the object layer.

13.12 Iterators

Generator definitions are not supported. Generator and iterator iteration is supported through the object layer.

Range iterators are syntactic sugar for looping constructs. Custom iterators are not supported. The `iter` and `next` functions are are supported only through the object layer.

```
for i in xrange(start, stop, step):
    foo()
```

Is lowered into some equivalent low-level looping construct that roughly corresponds to the following C code:

```
for (i = start; i < stop; i += step) {
    foo();
}
```

The value of `i` after the loop block follows the Python semantics and is set to the last value in the iterator instead of the C semantics.

`xrange` and *range*‘ are lowered into the same constructs.

`enumerate` is supported through the object layer.

13.13 Comprehensions

List comprehensions are rewritten into equivalent loops with list appending. Generator comprehensions are not supported.

13.14 Builtins

- abs - Supported
- all - object layer
- any - object layer
- apply - object layer
- basestring - object layer
- bin - object layer
- bool - Supported
- buffer - object layer
- bytearray - object layer
- bytes - object layer
- callable - object layer
- chr - object layer
- classmethod - object layer
- cmp - object layer
- coerce - object layer
- compile - object layer
- complex - Supported
- delattr - object layer
- dict - object layer
- dir - object layer

- divmod - object layer
- enumerate - object layer
- eval - object layer
- execfile - object layer
- exit - object layer
- file - object layer
- filter - Supported
- float - Supported
- format - object layer
- frozenset - object layer
- getattr - object layer
- globals - object layer
- hasattr - object layer
- hash - object layer
- help - object layer
- hex - object layer
- id - object layer
- input - object layer
- int - Supported
- intern - object layer
- isinstance - object layer
- issubclass - object layer
- iter - object layer
- len - Supported
- list - object layer
- locals - object layer
- long - Supported
- map - object layer
- max - object layer
- memoryview - object layer
- min - Supported
- next - object layer
- object - object layer
- oct - object layer
- open - object layer
- ord - object layer

- pow - Supported
- print - Supported
- property - object layer
- quit - object layer
- range - Supported
- raw_input - object layer
- reduce - object layer
- reload - object layer
- repr - object layer
- reversed - object layer
- round - Supported
- set - object layer
- setattr - object layer
- slice - object layer
- sorted - object layer
- staticmethod - object layer
- str - Supported
- sum - object layer
- super - object layer
- tuple - object layer
- type - object layer
- unichr - object layer
- unicode - object layer
- vars - object layer
- xrange - Supported
- zip - object layer

13.15 Slice

Named slicing is not supported. Slice types are supported only through the object layer. Slicing as an indexing operation is supported.

```
a = slice(0, 1, 2)
```

13.16 Classes

Classes are supported through extension types.

13.17 Casts

```
int :: a -> int
bool :: a -> bool
complex :: a -> bool
```

The coerce function is not supported.

The `str`, `list` and `tuple` casts are not supported.

13.18 Characters

The `chr`, `ord` are supported for the integer and character types. `unichr`, `hex`, `bin`, `oct` functions are supported through the object layer.

13.19 Closures

Nested functions and closures are supported. Construction goes through the object layer. Calling from numba does not.

The `nonlocal` keyword is not supported.

13.20 Globals

Global variables are not supported and resolved as constants. The `global` keyword is not supported.

13.21 Arguments

Variadic and keyword arguments are not supported.

13.22 Assertions

Assertions are not supported.

13.23 Operators

- And
- Or
- Add
- Sub
- Mult
- Div

- Mod
- Pow
- LShift
- RShift
- BitOr
- BitXor
- BitAnd
- FloorDiv
- Invert
- Not
- UAdd
- USub
- Eq
- NotEq
- Lt
- LtE
- Gt
- GtE

Comparison operator chaining is supported and is desugared into boolean conjunctions of the comparison operators:

```
(x > y > z)
```

```
(x > y) and (y > z)
```

Not supported:

- Is
- IsNot
- In
- NotIn

13.24 Division

Division follows the Python semantics for distinction between `floordiv` and `truediv` but operates over unboxed types with no error checking.

13.25 Math Functions

- abs
- pow

- round

13.26 Floating Point Math

- acos
- acosh
- asin
- asinh
- atan
- atan2
- atanh
- ceil
- cos
- cosh
- degrees
- erf
- erfc
- exp
- expm1
- exp2
- fabs
- floor
- fmod
- hypot
- log
- logaddexp
- logaddexp2
- log10
- log1p
- modf
- pow
- rint
- sin
- sinh
- sqrt
- tan

- tanh
- trunc

Constants such as `math.e` and `math.pi` are resolved as constants.

13.27 Complex Math

- abs
- acos
- acosh
- asin
- asinh
- atan
- atanh
- cos
- cosh
- exp
- expm1
- exp2
- log
- log10
- log1p
- sin
- sinh
- sqrt
- tan
- tanh

Developer Documentation (outdated)

CHAPTER

FOURTEEN

NUMBA ARCHITECTURE

Contents

- Numba Architecture
  - Introduction
  - Core Entry Points
    - Run-time Translation
    - Call-time Specialization
    - Extension Types
    - Compile-time Translation
  - Translation Internals
    - Towards More Modular Pipelines
      - Modularity
      - Diagram
    - *Discussion: Pipeline Composition*
    - *Discussion: Pipeline Environments*
  - Terms and Definitions
  - Appendix

14.1 Introduction

This document serves two purposes: to introduce other developers to the high-level design of Numba's internals, and as a point for discussion and synchronization for current Numba developers.

14.2 Core Entry Points

Numba has several modes of use:

1. As a run-time translator of Python functions into low-level functions.
2. As a call-time specializer of Python functions into low-level functions.
3. As a run-time builder of extension types.
4. As a compile-time translator of Python modules into shared object libraries.

1. As a compile-time builder of extension types.

1. As a framework for static analysis and code generation.

The following subsections describe the primary entry points for these modes of use. Each usage mode corresponds to a specific set of definitions provided in the top-level numba module.

14.2.1 Run-time Translation

Users denote run-time translation of a function using the `numba.jit()` decorator.

14.2.2 Call-time Specialization

Users denote call-time specialization of a function using the `numba.autojit()` decorator.

14.2.3 Extension Types

Numba supports building extension types using the `numba.jit()` decorator on a class.

14.2.4 Compile-time Translation

Users denote compile-time translation of a function using the `numba.export()` and `numba.exportmany()` decorators.

14.3 Translation Internals

14.3.1 Towards More Modular Pipelines

The end goal of building a more modular pipeline is to decouple stages of compilation and make a more modular way of composing transformations.

- **State threaded through the pipeline** 1) AST - Abstract syntax tree, possibly mutated as a side-effect of a pass.

  2) Structured Environment - A dict like object which holds the intermediate forms and data produced as a result of data.

- **Composition of Stages**
  - Sequencing
  - Composition Operator
  - Error handling and reporting in pass failure.
- Pre/Post Condition Checking
  - Stages should have attached pre / post conditions to check the success criterion of the pass for the inputted or resulting ast and environment. Failure to meet this conditions should cause the pipeline to halt.

Modularity

Note: recursive definitions

```
jit     := parse o link o jit
pycc    := parse o emit o link
autojit := cache o autojit
cache   := pipeline o jit

blaze   := mapast o jit
```

Diagram

```
Block diagram:
                 Input
                   |
+----------------------+
|          pass 1      |
+--------|----------|--+
       context     ast
         |          |
  postcondition     |
         |          |
  precondition      |
         |          |
+--------|----------|--+
|          pass 2      |
+--------|----------|--+
       context     ast
         |          |
  postcondition     |
         |          |
  precondition      |
         |          |
+--------|----------|--+
|          pass 3      |
+--------|----------|--+
       context     ast
         |          |
  precondition      |
         |          |
         +----------+-----> Output
```

14.3.2 *Discussion: Pipeline Composition*

We can do composition in a functional way:

```
def compose_stages(stage1, stage2):
  def composition(ast, env):
    return stage2(stage1(ast, env), env)
  return composition

pipeline = compose_stages(...compose_stages(parse, ...), ...)
```

Or, we can do composition using iteration:

```
for stage in stages:
  ast = stage(ast, env)
```

Whether the end result is a function or a class is also still up for discussion.

Proposal 1: We replace the Pipeline class to use a list of stages, but these can either be functions or subclasses of the `PipelineStage` class.

14.3.3 *Discussion: Pipeline Environments*

Proposal 1: We present an ad hoc environment. This provides the most flexibility for developers to patch the environment as they see fit.

Proposal 2: We present a well defined environment class. The class will have well defined properties that are documented and type-checked when the internal stage checking flag is set.

14.4 Terms and Definitions

14.5 Appendix

CHAPTER

FIFTEEN

NUMBA INTERMEDIATE REPRESENTATION SPECIFICATION

15.1 Numba IR Stages

We provide different entry and exit points in the Numba architecture to facilitate reuse. These entry points also allow for further decoupling (modularity) of the Numba architecture. The Numba architecture is broken into several stages, or compilation passes. At the boundaries between each compilation stage, we define a specific intermediate representation (IR).

The Numba intermediate representations (IR's) are, from high-level to low-level, as follows:

- The Python AST IR (input to a numba frontend)
- Normalized IR
  - Like a Python AST, but contains normalized structures (assignments, comparisons, list comprehensions, etc)
- Untyped IR in SSA form
  - Expanded control flow (with annotations)
- Typed IR in SSA form
- Low-level Portable IR
- Final LLVM IR, the final input for LLVM. This IR is not portable since the sizes of types are fixed.

All IRs except the last are portable across machine architectures. We get the following options for rewrites:

Each stage specifies a point for a series of IR transformations that together define the input for the next stage. Each rewrite may target a different IR stage:

- The input to *Stage 1* is an AST with all high-level syntactic constructs left intact. This allows high-level transformations that operate or expand most suitable at an abstract syntax level.
- The input to *Stage 2* is a Function with a sequence of basic blocks (expanded control flow), such that all control flow can be handled uniformly (and there may still be high-level annotations that signify where loops are (so that we don't have to find them again from the dominator tree and CFG), where exceptions are raised and caught, etc). Def/use and variable merges are explicit.

  Expressions and statements are encoded in sequences of AST expression trees. Expressions result in implicit Values which can be referred to in subsequent expression trees without re-evaluation (we can replace the CloneNode/CloneableNode mechanisms that currently allow subtree sharing).

- The input to *Stage 3* is the same as to *Stage 2*, except that it additionally contains type information (and type promotions).
- *Stage 4* is a low-level three-address code representation that still has polymorphic operations, such as `c = add(a, b)`, where `a` and `b` can be operands of any scalar type. A final pass then lowers these constructs to specialized LLVM instructions.

IRs up to the low-level IR (input to *Stage 4*) should still contain explicit variable stores, so that passes can rewrite variable assignments to for instance assignments to struct or extension type instance members. Keeping stores and loads to and from these variables in the original order is important in the context of closures (or any function call which can modify a stack variable through a pointer).

We must make sure to support preloads for variable definitions across basic blocks, e.g.:

```
if ...:
    A = ...
else:
    A = ...

for i in range(...):
    use(A[0])
```

In this example we want to preload `A.data` (and `a.strides[0]`). This can probably work equally well if each expression value is an implicit definition and we have a way to find Values given a Phi. We then get:

```
ValueC = BinOp(ValueA, Add, ValueB)
Store(Name('c'), ValueC)
```

Instead of:

```
Assign(Name('c'), BinOp(Name('a'), Add, Name('b')))
```

15.2 Numba Intermediate Representations

15.2.1 Python AST IR

Numba's initial intermediate representation is Python abstract syntax as defined in Python's ast module documentation. Note that this definition is specific to the version of Python being compiled.

Numba must support Python 2 abstract syntax (specifically versions 2.6, and 2.7) for the foreseeable future.

15.2.2 Normalized IR

The normalized IR starts with the latest ASDL definition for Python 3 abstract syntax, but makes the following changes:

- Python's top-level module containers, defined in the `mod` sum type, are abandoned. The Numba normalization stage will return one or more instances of the normalized `stmt` sum type.
- Constructs that modify the namespace may only reference a single name or syntactic name container. These constructs include:
  - global, nonlocal
  - import, import from
  - assignments

  - del
- Expressions are un-flattened. Operators on more than two sub-expressions are expanded into expression trees. Comparison expressions on more than two sub-expressions will use temporaries and desugar into an expression tree.

Numba must translate Python 2 code into Python 3 constructs. Specifically, the following transformations should be made:

- Repr (backticks): Call(Name('repr'), value)
- Print(...): Call(Name('print'), ...)
- Exec(...): Call(Name('exec'), ...)
- Subscript(..., slices, ...): Subscript(..., ExtSlice(slices), ...)
- Ellipsis (the slice): Ellipsis (the expression)
- With(...): ...
- Raise(...): ...

The formal ASDL definition of the normalized IR is given here: https://github.com/numba/numba/blob/devel/numba/ir/Normalized.asdl

Issue: Desugaring comparisons

Do we introduce this as being a DAG already? If not, we have a problem with desugarring comparisons. We need assignment to bind temporaries, so we're going to have a hard time handling the following:

```
Compare(e0, [Eq, Lt], [e1, e2])
```

We'd want "e1" to be the same sub-expression in the normalized IR:

```
BoolOp(Compare(e0, Eq, e1), And, Compare(e1, Lt, e2))
```

How do later stages detect this as being the same sub-expression, etc?

Proposal

We should add the following constructor to expr:

```
expr |= Let(identifier name, expr def, expr user)
```

Semantically, this is sugar for the following:

```
Call(Lambda(name, user), [def])
```

Later stages of the compiler should not bother to do this desugaring. They should instead prefer to just create a SSA definition:

```
$name = [| def |]
$0 = [| user |]
```

In the case of a chained comparison, we can then make the following transformation:

```
Compare(e0, [cmp0, ...], [e1, ...])
==>
Let(fresh0, e0,
    Let(fresh1, e1,
```

```
        BoolOp(Compare(fresh0, cmp0, fresh1), And,
               Compare(fresh1, [...], [...]))
```

Where `fresh0` and `fresh1` are fresh variable names. The normalization transformer should recursively apply this rewrite until it reaches a case where the comparison is binary.

15.2.3 Untyped IR in SSA form

Given a normalized AST, we preserve the `expr` sum type, but perform control-flow analysis, data-flow analysis for phi-node injection, closure conversion, and lambda lifting. These transformations result in the following intermediate representation:

```
mod = Module(unit* units)

unit = CodeObject(..., block* blocks)
     | DataObject(identifier label, expr init)

block = Block(identifier id, defn* defns, tail tail_expr)

tail = Jump(identifier target)
     | If(expr test, identifier true_target, identifier false_target)
     | Raise(expr exn)
     | Return(expr result)

defn = (identifier? def_id, expr value)

expr |= Phi(phi_source* incomming)

phi_source = (identifier in_block, expr in_val)
```

15.2.4 Typed IR in SSA form

The typed IR is similar to the untyped IR, except that every (sub-)expression is annotated with a type.

Furthermore, the AST is augmented with `Promotion` terms, which promote a variable for a merge in a subsequent CFG block. E.g.:

```python
# y_0
if x > 10:
    # block_if
    y = 2           # y_1
else:
    # block_else
    y = 3.0         # y_2
```

In the example above, `block_if` will contain a `Promotion` with a use of `y_1`, replacing all uses of `y_1` with the promotion value (which can only ever be a single phi node).

I.e. we rewrite `y_1 = 2` to `[ y_1 = 2 ; %0 = Promote(y_1, float) ]` and `PhiNode(NameRef(y_1), NameRef(y_2))` to `PhiNode(%0, NameRef(y_2))`.

All types adhere themselves to a schema, e.g.:

```
type
  = Array(type dtype, int ndim)
  | Pointer(type base_type, int? size)
  | ...
```

Since the schema specifies the interfaces of the different nodes, users can supply their own node implementation (something we can do with the type system). Hence user-written classes can be automatically instantiated instead of generated ones. The code generator can still emit code for serialization.

15.2.5 Low-level Portable IR

The low-level portable IR is a low-level, platform agnostic, IR that:

- The IR contains only low-level, native types such as `int_`, `long_`, pointers, structs, etc. The notion of high-level concepts such as arrays or objects is gone.

This portable IR could be LLVM IR , which may still contain abstract or opaque types, and make calls to the Numba runtime library abstraction layer.

15.2.6 Final LLVM IR

The final LLVM IR is LLVM assembly code, with no opaque types, and specialized to a specific machine target.

15.3 Appendicies

15.3.1 Appendix: Design Notes

This appendix looks at various features and discusses various options for representing these constructs across the compiler.

Closures

A key step in the transition from the normalized AST IR to the untyped SSA IR is closure conversion. For example, given the following code:

```
def closure_test(foo):
    foo += 3
    def bar(baz):
        return foo + (lambda x: x - global_z * foo)(baz)
    foo += 2
    return bar
```

Numba should generate SSA code equivalent to the following:

```
def __anonymous(af, x):
    return x - global_z * af.foo

def __bar(af, baz):
    return af.foo + make_closure(__anonymous,
                                 make_activation_frame(af, []))(baz)

def closure_test(foo):
    af = make_activation_frame(None, ['foo'])
    af.foo = foo
    af.foo += 3
    bar = make_closure(__bar, af)
    af.foo += 2
    return bar
```

Parent frames

The above convention implies the following ASDL definition of the `MakeFrame` constructor (XXX cross reference discussion of IR expr language):

```
MakeFrame(expr parent, identifier* ids)
```

The parent frame provides a name space for identifiers unresolved in the current frame. If we employ this constructor, we diverge slightly from CPython. CPython manages each unbound variable within a cell, and these cells are copied into a new frame object (which is a tuple in CPython) for every child closure constructed.

Alternative: Explicit parameterization

Another method for doing closure conversion involves parameterizing over all free variables, and is closer to CPython's approach:

```python
def __anonymous(foo, x):
    return x - global_z * foo.load()

def __bar(foo, baz):
    return foo.load() + partial(__anonymous, [foo])(baz)

def closure_test(foo):
    foo = make_cell(foo)
    foo += 3
    bar = partial(__bar, [foo])
    foo += 2
    return bar
```

This approach uses partial function application to build closures. The resulting representation affords opportunities for optimizations such as rewriting `partial(fn, [x])(y)` to `fn(x, y)`.

Default, variable, and keyword arguments

XXX Do we need a MakeFunction() expression constructor for supplying default arguments? This follows from discussion of closures, above.

Iterators

Iterators in the untyped IR

We considered three options for implementing iterators. The first was to use exception handling constructs. Given the following code:

```python
for x in i:
    if x == thingy: break
else:
    bar()
baz()
```

Translation to the untyped IR could result in something like the following:

```
bb0: ...
     $0 = Call(Constant(numba.ct.iter), [Name("i", Load())])
     Try(bb1, [ExceptHandler(Constant(StopIteration), None, bb2)],
         None, None)

bb1: $1 = Call(Constant(numba.ct.next), [$0])
     If(Compare($1, Eq(), Name("thingy", Load())), bb3, bb1)

bb2: Call(Name("bar", Load()), [])
     Jump(bb3)

bb3: Call(name("baz", Load()), [])
     ...
```

The second option was defining a `Next()` terminator. `Next()` could provide sugar for the special case where we are specifically waiting for a `StopIteration` exception:

```
bb0: ...
     $0 = Call(Constant(numba.ct.iter), [Name("i", Load())])
     Jump(bb1)

bb1: Next(Name("x", Store()), $0, bb2, bb3)

bb2: If(Compare(Name("x", Load()), Eq, Name("thingy", Load())), bb4, bb1)

bb3: Call(Name("bar", Load()), [])
     Jump(bb4)

bb4: Call(Name("baz", Load()), [])
     ...
```

We loose SSA information, but provide opportunity for more readily recognizing for loops.

The third option was to follow the CPython VM semantics of `FOR_ITER`, where we define `Next()` as an expression constructor which can either return a result or some sentinel (specific to CPython, this is the `NULL` pointer):

```
bb0: ...
     $0 = Iter(Name("i", Load()))
     Jump(bb1)

bb1: $1 = Next($0)
     If(Compare($1, Neq(), Constant(numba.ct.NULL)), bb2, bb3)

bb2: If(Compare($1, Eq(), Name("thingy", Load())), bb3, bb1)

bb3: Call(Name("bar", Load()), [])
     Jump(bb4)

bb4: Call(name("baz", Load()), [])
     ...
```

This final output looks very similar to the output of the second option, but prevents us from having to use the `Name()` expression for anything other than global and parameter variables.

Generators

The Numba Google group's generator discussion identified two methods for implementing generators in Numba. These can roughly be summarized as "enclosing everything in a big C-like switch statement", and "use goroutines".

The following web pages elaborate on these techniques:

- http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html
- https://code.google.com/p/try-catch-finally/wiki/GoInternals

Global and nonlocal variables

Given:

```
z = 42
def foo():
    global z
    bar(z)
    z = 99
```

We could generate the following in untyped IR:

```
[
  DataObject("z", Constant(42)),
  CodeObject("foo", ([], None, None, ...), [
    Block("entry", [
        (None, Call(Name("bar", Load()), [LoadGlobal("z")])),
        (None, StoreGlobal("z", Constant(99)))
      ], Return(Constant(None)))])
]
```

Exceptions and exception handling

Both the raise and try-except-finally language constructs map into the untyped SSA IR as basic-block terminators:

```
tail = ...
     | Raise(expr exn)
     | Try(identifier body,
           excepthandler* handlers,
           identifier? orelse,
           identifier? finalbody)

...

excepthandler = ExceptHandler(expr *types,
                              identifier? name,
                              identifier body)
                attributes (int lineno, int col_offset)
```

In the low-level IR, these constructs lower into Numba run-time calls:

```
bb0:  ...
      Try('bb1', [ExceptHandler([ty0,...], 'name0', 'bb2'),
                  ...
                  ExceptHandler([tyn,...], 'namen', 'bbn0')],
          'bbn1', 'bbn2')
bb1:  ...
      Jump('bbn2')
bb2:  ...
      Jump('bbn2')
...
bbn0: ...
```

```
       Jump('bbn2')
bbn1: ...
       Jump('bbn2')
bbn2: ...
```

Goes to:

```
bb0:  ...
       $0 = SetupTry()
       If($0, 'bb1', 'bb2')
bb1:  ...
       Jump('bbn2')
bb2:  $1 = TestExn([ty0, ...])
       If($1, 'bbx2', 'bb3')
bbx2: $name0 = GetExn()
       ...
       Jump('bbn2')
...
bbn0: $2 = TestExn([tyn, ...])
       If($2, 'bbxn', 'bbn1')
bbxn: $namen = GetExn()
       ...
       Jump('bbn2')
bbn1: GetExn()
       ...
       Jump('bbn2')
bbn2: ...
```

Decorators

Classes and objects

Namespaces

15.3.2 Appendix: Language Cross Reference

The following sections follow the Python Language Reference, and provide notes as on how the various Numba intermediate representations support the Python language.

Expressions

Simple statements

Expression statements

Assignment statements

The assert statement

The pass statement

The del statement

The return statement

The yield statement

The raise statement

The break statement

The continue statement

The import statement

The global statement

Compound statements

The if statement

The while statement

The for statement

The try statement

The with statement

Function definitions

Class definitions

Top-level components

15.3.3 Appendix: Other Design Notes

Use of Schemas

We can use our schemas to:

- Validate IR instances
- Generate Python AST classes with typed properties and fast visitor dispatching
- Generate Higher- or Lower-level LLVM IR
- Generate conversion code to and from an ATerm representation
- Generate a flat representation. E.g. a form of Three Address Code
- Generate an implementation in other languages that can load a serialized representation and construct an AST in that langauge
- Generate type definitions and serialization routines in other languages.

> **Note:** This can help other languages target Numba as a backend compiler more easily, since they can build up the IR using in-memory data structures for the IR most suitable to their needs.

- Generate definitions for use in Attribute Grammars
- Executable IR (*Executable IR*)

Executable IR

There are two ideas:

- Write a simple interpreter
- Generate source code containing calls to a runtime library

Building a Call Graph

This will be useful to use LLVM for in order to:

- Efficiently infer types of direct or indirect uses of recursion for autojit functions or methods
- Detect such recusion by letting LLVM find the SCCs in the call graph, and resolving in an analogous and cooperative manner to how we resolve the type graph

CHAPTER

SIXTEEN

NUMBA ROADMAP

This document describes features we want in numba, but do not have yet. We will first list what we want in upcoming versions, and then what features we want in general. Those features can always be added to the roadmap for upcoming versions if someone is interested in implementing them.

16.1 Numba Versions

16.1.1 1.0

What we want for 1.0 is:

- Numba loader (loader)
- IR stages (**stages_**)
- More robust type inferencer
- Well-defined runtime
    - including exception support
- Debug info
- numba –annotate tool (annotate)
- parallel tasks (green threads, typed channels, scheduler)
- generators on top of the green thread model

We also like some minimal Cython support, in addition to the longer term goals of SEP 200. One idea from Zaur Shibzukhov is to provide support for Cython pxd overlays:

```
# foo.py

def my_function(a):
    b = 2
    return a ** b
```

Such a module can be overlain with a Cython pxd file, e.g.

```
# foo.pxd

cimport cython

@cython.locals(b=double)
cpdef my_function(double a)
```

For some inspiration of what we can do with pxd overlays, see also: https://github.com/cython/cython/blob/master/Cython/Compiler/FlowControl.pxd

We can now compile `foo.py` with Cython. We should be able to similarly compile `foo.py` with numba, using `pycc` as well as at runtime to produce a new module with annotated functions compiled in the right order.

16.2 Thing we want

We will order these from less involved to more involved, to provide different entry points to numba development.

16.3 Less intricate

Here as some less intricate topics, providing easier starting points for new contributors:

16.3.1 NumPy Type Inference

Full/more support for type inference on NumPy functions:

- http://numba.pydata.org/numba-doc/dev/doc/type_inference.html
- https://github.com/numba/numba/tree/devel/numba/type_inference/modules

16.3.2 Typed Containers

We currently have (naive implementations of):

- typedlist (https://github.com/numba/numba/blob/devel/numba/containers/typedlist.py)
- typedtuple (https://github.com/numba/numba/blob/devel/numba/containers/typedtuple.py)

But we want many more! Some ideas:

- typeddict
- typedset
- typedchannel
  - one thread-safe (nogil) and one requiring the GIL

Perhaps also the ordered variants of `typeddict` and `typedset`.

16.3.3 Intrinsics

Support for LLVM intrinsics (we only have instructions at the moment):

- http://numba.pydata.org/numba-doc/dev/doc/interface_c.html#using-intrinsics
- https://github.com/numba/numba/blob/devel/numba/intrinsic/numba_intrinsic.py

E.g.:

```
intrin = numba.declare_intrinsic(int64(), "llvm.readcyclecounter")
print intrin()
```

16.3.4 Source Annotator

Analogous to `cython --annotate`, a tool that annotates numba source code and finds and highlights which parts contain object calls. Ideally, this would also include, for each source line (expand on click?):

- The final (unoptimized) LLVM bitcode
  - And optionally the optimized code and/or assembly
- Code from intermediate numba representations
  - After we start implementing several layers of IR, see http://numba.pydata.org/numba-doc/dev/doc/ir.html
- The type of each sub-expression and variable (on hover?)

Issue: https://github.com/numba/numba/issues/105

16.3.5 Numba Loader

Allow two forms of code caching:

- For distribution (portable IR)
- Locally on disk (unportable compiled binaries)

The first bullet will allow library writers to distribute numba code while not being tied to numba versions that users have installed. This would be similar to distribution of C code compiled from Cython source:

```
$ numba --compile foo.py
Writing foo.numba
```

We can now distribute `foo.numba`. Load code explicitly:

```
from numba import loader
foo = loader.load("foo.numba")
foo.func()
```

... or use an import hook:

```
from numba import loader
loader.install_hook()

import foo
foo.func()
```

... or compile to extension modules during setup:

```
from numba.loader import NumbaExtension

setup(
    ...,
    ext_modules=[
        NumbaExtension("foo.bar",
                       sources=["foo/bar.numba"]),
        ],
)
```

Or perhaps more conveniently, implement `find_numba_modules()` to find all `*.numba` source files and return a list of `NumbaExtension`.

This also plays into the IR discussion found here: http://numba.pydata.org/numba-doc/dev/doc/ir.html

16.3.6 JIT Special Methods

Jit operations that result in calls to special methods like `__len__`, `__getitem__`, etc. This requires some careful thought as to the stage where this transformation should take place.

16.3.7 Array Expressions

Array Expression support in Numba, including scans, reductions, etc. Or maybe we should make Blaze a hard dependency for that?

16.4 More intricate

More intricate topics, in no particular order:

16.4.1 Extension Types

- Support autojit class inheritance
- Support partial method specialization

```
@Any(int_, Any)
def my_method(self, a, b):
    ...
```

Infer the return type and specialize on parameter type `b`, but fix parameter type `a`.

- Allow annotation of pure-python only methods (don't compile)

What we also need is native dispatch of foreign callables, in a sustainable way: SEP 200 and SEP 201

- https://github.com/numfocus/sep/
- Widen support in scientific community

16.4.2 Recursion

Support recursion for autojit functions and methods:

- Construct call graph
- Build condensation graph and resolve
  - similar to cycles in SSA

16.4.3 Exceptions

Support for zero-cost exceptions: support in the runtime libraries for all models:

- True zero-cost exceptions
  - Stack trace through libunwind/apple backtrace/LLVM info based on instruction pointer
  - http://llvm.org/docs/LangRef.html#invoke-instruction
  - http://llvm.org/docs/ExceptionHandling.html

- Setjmp/longjmp
  - Optionally with exception analysis to allow cheap cleanup for the simpler cases
- Costful exceptions
  - "return -1"
  - Implement fast `NumbaErr_Occurred()` or change calling convention for native or void returns

We also need to allow users to take the pointer to a numba `jit` function:

```
numba.addressof(my_numba_function)
```

We can allow specifying an exception model:

- `propagate=False`: This does not propagate, but uses PyErr_WriteUnraisable
- `propagate=True`: Implies `write_unraisable=False`. Callers check with `NumbaErr_Occurred()` (or for NULL if object return). Maybe also specify a range of badvals:
  - int -> 0xdeadbeef (`ret == 0xdeadbeef && NumbaErr_Occurred()`)
  - float -> float('nan') (`ret != ret && NumbaErr_Occurred()`)

Note: We have `numba.addressof()`, but we don't have `NumbaErr_Occurred()` yet.

16.4.4 Debug info

GDB Backtraces!

See:

- https://github.com/llvmpy/llvmpy/blob/debuginfo/llvm/debuginfo.py
- https://github.com/llvmpy/llvmpy/blob/debuginfo/test/test_debuginfo.py

Or is there a successor to that?

16.4.5 Struct references

Use cheap heap allocated objects + garbage collection?

- or atomic reference counts?

Use stack-allocation + escape analysis?

16.4.6 Blaze

Blaze support:

- compile abstract blaze expressions into kernels
- generate native call to blaze kernel

16.4.7 Generators/parallel Tasks

Support for generators based on green threading support:

- Write typed channels as autojit class
- Support green thread context switching
- Rewrite iteration over generators

```
def g(N):
    for i in range(N):
        yield f(i)      # write to channel (triggering a context switch)

def consume():
    gen = g(100)        # create task with bound parameter N and channel C
    for i in gen:       # read from C until exhaustion
        use(i)
```

See also https://groups.google.com/a/continuum.io/forum/#!searchin/numba-users/generators/numba-users/gaVgArRrXqw/HTyTzaXsW_EJ for how this compares to generators based on closures.

16.4.8 Python 3.3 support

We support Python 3.3, but we can additionally support type-annotations:

```
def func(a: int_, b: float_) -> double:
    ...
```

Maybe this can work with numba.automodule(my_numba_module) as well as with jit and autojit methods.

16.4.9 GPUs

- SPIR support (OpenCL)

16.4.10 Vector support

- Vector-types in Numba
  - What does this look like?

CHAPTER SEVENTEEN

DEVELOPMENT CRASH COURSE

This document describes a short crash-course for numba development, where things are and where we want them at.

17.1 Overview

We start with a Python AST, compiled from source code or decompiled from bytecode using meta. We run a series of stages that transform the program as an AST to something from which we can generate code. The pipeline and environment are central pieces in this story:

- https://github.com/numba/numba/blob/devel/numba/pipeline.py

  The pipeline has a series of functions that mostly dispatch to the actual transformations or visitors.

- https://github.com/numba/numba/blob/devel/numba/environment.py

  The environment defines the pipeline order. Noteworthy is **:py-class:‘numba.environment.FunctionEnvironment‘**

17.1.1 Stages

The main stages are:

- Control flow analysis: `numba/control_flow`

  This builds a Control Flow Graph from the AST and computes the SSA graph for the variable definitions. In this representation, each variable assignment is a definition, e.g.:

  ```
  x = 0         # definition 1
  x = "hello"   # definition 2
  ```

  Assignments and variable references are recorded as abstract statements in the basic blocks of the CFG, such as

  – numba.control_flow.cfstats.NameAssignment

  – numba.control_flow.cfstats.NameReference

  – numba.control_flow.cfstats.PhiNode

  The phi node occurs at control flow joint points, e.g. after an `if`-statement, or in the condition block of a loop with a loop-carried dependency for a variable:

```
if c:
    x = 0
else:
    x = 2
# phi here
```

and:

```
x = 0
for i in range(N):  # phi in condition block: x_1 = phi(x_0, x_2)
    x = x + i # loop-carried dependency
```

The phi nodes are themselves variable definitions, and they define the points where variables merge and need a unifyable type (e.g. (int, int), or (int, float), as opposed to (int, string)).

- Type inference: `numba.type_inference`

  Infer types of all expressions, and fix the types of all local variables. This operates in two stages:

  – Infer types for all local variable definitions (including phis)

    For an overview of this see *Type Dependence Graph Construction* below.

  – Now that all variable definitions have a type, we can easily infer types for all expressions by propagating type information up the tree

When the type inferencer cannot determine a type, such as when it calls a Python function or method that is not a Numba function, it assumes type `object`. Object variables may be coerced to and from most native types.

The type inferencer and other code insert CoercionNode nodes that perform such coercions, as well as coercions between promotable native types.

It also resolves the return type of many math functions called in the numpy, math and cmath modules.

Each AST expression node has a Variable that holds the type of the expression, as well as any meta-data such as constant values that have been determined.

To see how builtins, math and numpy callables are handled, have a read through *type_inference* in the user documentation, as well as `numba.type_inference.modules`::

> https://github.com/numba/numba/tree/devel/numba/type_inference/modules

This above sub-package is an important part of numba that infers (and sometimes grossly rewrites) calls to known functions.

- Specialization/Lowering: numba/specialize and numba/transforms.py

  What follows over the typed code are a series of transformations to lower the level of the code into something low-level - something amenable to code generation:

  – Rewrite loops over `range` or `xrange` into a `while` loop with a counter

  – Rewrite iteration over arrays to a loop over `range` with an index into the array

  – Lower object conversions into calls into the Python C-API. For instance it resolves coercions to and from object into calls such as `PyFloat_FromDouble`, with a fallback to `Py_BuildValue`/`PyArg_ParseTuple`.

  – Lower exception code into calls into the C-API and insert NULL pointer checks in places

  – Normalize comparisons (e.g. `a < b < c` => `a < b and b < c`)

  – Keep track of refcounts. This is mostly done with ObjectTempNode, which hold a temporary for an object (a new reference). These temporaries are decreffed at cleanup:

```
define double @func()  {
entry:
  %retval = alloca double           ; return value
  %tmp = alloca object              ; object temporary
  ...
  %obj = call PyObject_SomeNewObject()
  %have_error = cmp obj NULL        ; check return value
  cbranch %have_error, label %error, label %success

success:                            ; no error
  do something interesting with %obj
  store %something %retval          ; return some value
  br return_block                   ; ok, we're done

error:                              ; some error occurred :(
  store NaN retval
  br cleanup

return_block:                       ; clean up objects
  call void Py_XDECREF(%0)
  %result = load %retval
  ret %result                       ; return result
}
```

- Code generation: numba/codegen

  Generate LLVM code from the transformed AST. This is relatively straightforward at this point. One tricky problem is that the basic blocks from the LLVM code no longer correspond to the basic blocks of the CFG, since error checks have been inserted. This makes tracking phis harder than it should be.

  The code generator uses utility functions from numba/utility and numba/external to do things like refcounting (`Py_INCREF`, etc) and uses helpers to slice and broadcast arrays.

Package Structure

- numba/type_inference

Type inference

- numba/typesystem

  Numba typesystem, see also *types*

- numba/specialize

  Lowering transformations, along with numba/transforms.py . Coercions are in numba/transforms.py

- numba/nodes

  Contains AST nodes. Some nodes that need some explaining:

  - ObjectTempNode:

    Holds a PyObject * temporary that it manages a refcount for

  - CloneNode/CloneableNode:

    These nodes are used for subtree sharing, to avoid re-evaluation of the subtree. Consider e.g. the expression 'x * 2', which we want to refer to twice, but evaluate once. We can do the following:

    ```
    cloneable = CloneableNode(<x * 2 expression>)
    clone   = CloneNode(cloneable)
    ```

    Here `cloneable` must be evaluated before `clone`. We can now generate as many clones as we want without re-evaluating `x * 2`

- numba/exttypes

  Numba extension types, have a read through *extclasses* first. These are fairly well documented. To see how they work, see below *Extension Classes*

- numba/closures

  Implements closures for numba. See *Closures* and *closureimpl* below for how they work.

- numba/support

  Ctypes, CFFI and NumPy support (slicing, etc)

- numba/array_expressions.py

  Implements array expressions using minivect. Since we don't actually use the tiling specializers or desperately need crazy optimizations for special cases, we should really use lair's `loop_nest` instead and throw away numba/minivect

- numba/vectorize

  The @vectorize functionality to build (generalized) ufuncs

- numba/wrapping

  Entry points to compile numba functions, classes and methods

- numba/utility and numba/external

  Runtime support utilities. And yes, you make a valid point, this should really be one package.

- numba/intrinsic

  Intrinsics and instruction support for numba, as well as... internal intrinsics. Merge internal stuff in numba/external :)

  See *intrinsics* for what intrinsics do.

- numba/containers

  Numba typed containers, see *containers*

- numba/asdl and numba/ir

  Utilities to validate ASTs and generate fast visitors/AST implementations from ASDL. This should be factored out into asdlpy or somesuch.

- numba/viz

  Format ASTs and CFGs with graphviz. See also the 'annotate' branch

- numba/minivect

  Array expression compiler. `numba.array_expressions` is the only remaining module depending on this. However, since none of the optimizations are actually used, it doens't make sense to keep this. Instead we can use the `loop_nest` function from the lair project.

  More on how the array expressions work: *Array Expressions*

Type Dependence Graph Construction

From the SSA graph we compute a type graph by inferring all variable assignments. This graph often has cycles, due to the back-edge in the CFG for loops. For instance we may have the following code:

```
x = 0
for i in range(10):
    x = f(x)

y = x
```

Where `f` is an external autojit function (i.e., it's output type depends on it's dynamic input type).

We get the following type graph:

Below we show the correspondence of the SSA variable definitions to their basic blocks:

Our goal is to resolve this type graph in topological order, such that we know the type for each variable definition (`x_0`, `x_1`, etc).

In order to do a topological sort, we compute the condensation graph by finding the strongly connected components and condensing them into single graph nodes. The resulting graph looks like this:

And `SCC0` contains the cycle in the type graph. We now have a well-defined preorder for which we can process each node in topological order on the transpose graph, doing the following:

- If the node represents a concrete type, propagate result along edge
- If the node represents a function over an argument of the given input types, infer the result type of this function
- For each SCC, process all internal nodes using fixpoint iteration given all input types to the SCC. Update internal nodes with their result types.

Closures

`numba/closures.py` provides support for closures and inner functions:

```
@autojit
def outer():
    a = 10 # this is a cellvar
```

```
    @jit('void()')
    def inner():
        print a # this is a freevar

    inner()
    a = 12
    return inner
```

The 'inner' function closes over the outer scope. Each function with cellvars packs them into a heap-allocated structure, the closure scope.

The closure scope is passed into 'inner' when called from within outer.

The execution of `def` creates a NumbaFunction, which has itself as the m_self attribute. So when 'inner' is invoked from Python, the numba wrapper function gets called with NumbaFunction object and the args tuple. The closure scope is then set in NumbaFunction.func_closure.

The closure scope is an extension type with the cellvars as attributes. Closure scopes are chained together, since multiple inner scopes may need to share a single outer scope. E.g.:

```
def outer(a):
    def inner(b):
        def inner_inner():
            print a, b
        return inner_inner

    return inner(1), inner(2)
```

We have three live closure scopes here:

```
scope_outer = { 'a': a }  # call to 'outer'
scope_inner_1 = { 'scope_outer': scope_outer, 'b': 1 } # call to 'inner' with b=1
scope_inner_2 = { 'scope_outer': scope_outer, 'b': 2 } # call to 'inner' with b=2
```

Function 'inner_inner' defines no new scope, since it contains no cellvars. But it does contain a freevar from scope_outer and scope_inner, so it gets scope_inner passed as first argument. scope_inner has a reference to scope outer, so all variables can be resolved.

These scopes are instances of dynamic numba extension classes.

Extension Classes

Extension classes live in numba/exttypes.

17.1.2 @jit

Compiling @jit extension classes works as follows:

- Create an extension Numba type holding a symbol table
- Capture attribute types in the symtab ...
  - ... from the class attributes:

    ```
    @jit
    class Foo(object):
        attr = double
    ```

- ... from __init__:

```
@jit
class Foo(object):
    def __init__(self, attr):
        self.attr = double(attr)
```

- Type infer all methods
- Compile all extension methods
  - Process signatures such as @void(double)
  - Infer native attributes through type inference on __init__
  - Path the extension type with a native attributes struct
  - Infer types for all other methods
  - Update the ext_type with a vtab type
  - Compile all methods
- Create descriptors that wrap the native attributes
- Create an extension type:

  { PyObject_HEAD ... virtual function table (func **) native attributes

  }

The virtual function table (vtab) is a ctypes structure set as attribute of the extension types. Objects have a direct pointer for efficiency.

17.1.3 @autojit

Compiling @autojit extension classes works as follows:

- Create an extension Numba type holding a symtab
- Capture attribute types in the symtab in the same was as @jit
- Build attribute hash-based vtable, hashing on (attr_name, attr_type).

  (attr_name, attr_type) is the only allowed key for that attribute (i.e. this is fixed at compile time (for now). This means consumers will always know the attribute type (and don't need to specialize on different attribute types).

  However, using a hash-based attribute table allows easy implementation of multiple inheritance (virtual inheritance), without complicated C++ dynamic offsets to base objects (see also virtual.py).

For all methods M with static input types:

  - Compile M
  - Register M in a list of compiled methods

- Build initial hash-based virtual method table from compiled methods
  - **Create pre-hash values for the signatures**
    - We use these values to look up methods at runtime
  - Parametrize the virtual method table to build a final hash function:

```
slot_index = (((prehash >> table.r) & self.table.m_f) ^
               self.displacements[prehash & self.table.m_g])
```

Note that for @jit classes, we do not support multiple inheritance with incompatible base objects. We could use a dynamic offset to base classes, and adjust object pointers for method calls, like in C++:

http://www.phpcompiler.org/articles/virtualinheritance.html

However, this is quite complicated, and still doesn't allow dynamic extension for autojit classes. Instead we will use Dag Sverre Seljebotn's hash-based virtual method tables:

https://github.com/numfocus/sep/blob/master/sep200.rst
https://github.com/numfocus/sep/blob/master/sep201.rst

The following paper helps understand the perfect hashing scheme:

Hash and Displace: Efficient Evaluation of Minimal Perfect Hash Functions (1999) by Rasmus Pagn:

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.6530

- Create descriptors that wrap the native attributes
- Create an extension type:

```
{
    hash-based virtual method table (PyCustomSlots_Table **)
    PyGC_HEAD
    PyObject_HEAD
    ...
    native attributes
}
```

We precede the object with the table to make this work in a more generic scheme, e.g. where a caller is dealing with an unknown object, and we quickly want to see whether it support such a perfect-hashing virtual method table:

NOTE: What we want is to actually use a separate attribute table in addition to the virtual method table, giving all extension objects a compatible layout.

```
if (o->ob_type->tp_flags & NATIVELY_CALLABLE_TABLE) {
    PyCustomSlots_Table ***slot_p = ((char *) o) - sizeof(PyGC_HEAD)
    PyCustomSlots_Table *vtab = **slot_p
    look up function
} else {
    PyObject_Call(...)
}
```

We need to store a PyCustomSlots_Table ** in the object to allow the producer of the table to replace the table with a new table for all live objects (e.g. by adding a specialization for an autojit method).

Wrappers

There are several wrappers in numba that wrap functions, classes and methods. The key implementations are in:

- numba/numbawrapper.pyx

- numba/numbafunction.c
- numba/codegen/llvmwrapper.py

What numba does is it creates a function like this (in C pseudo-code):

```
double square(double arg) {
    return arg * arg;
}
```

and it wraps it as follows:

```
PyObject *square_wrapper(PyObject *self, PyObject *args) {
    double arg;

    if (!PyArg_ParseTuple(args, "d", &arg))
        return NULL;

    return PyFloat_FromDouble(square(arg));
}
```

The wrapper is a CPython compatible function the likes of which you often see in extension modules. It is created by llvmwrapper.py. This wrapper is turned into an `PyCFunctionObject`, defined in `Include/methodobject.h` in CPython's source tree:

```
typedef struct {
    PyObject_HEAD
    PyMethodDef *m_ml; /* Description of the C function to call */
    PyObject    *m_self; /* Passed as 'self' arg to the C func, can be NULL */
    PyObject    *m_module; /* The __module__ attribute, can be anything */
} PyCFunctionObject;
```

Numba uses a wrapper (a subclass) of `PyCFunctionObject`, called `numbafunction`, which has some extra fields and gives the function a more pythonic interface, such as a dict, a way to override `__doc__`, etc. It also has a field to hold a closure frame (see *Closures*).

This function object is created after compilation of the function and its wrapper (`square_wrapper`). It is instantiated in `llvmwrapper.py:numbafunction_new`, calling `NumbaFunction_NewEx`.

17.1.4 @jit

Jit functions are wrapped typically by the `NumbaFunction` in `numbafunction.c`. The `NumbaCompiledWrapper` is only a temporary wrapper used in case of a recursive function that is still being compiled, and for which we have no pointer.

17.1.5 @autojit

Autojit functions are handled by `NumbaSpecializingWrapper`, which wraps a Python function and when called does a lookup in a cache to see if a previously compiled version is available. This code in in `numbawrapper.pyx`.

`NumbaSpecializingWrapper` holds an `AutojitCache` which tries to find a match very quickly. However, it may not always find a compiled version even though it's in the cache, for instance because there are values of different Python types which are represented using the same numba type.

This is then corrected by a slower path which tries to compile the function, and creates a type for each argument. It uses these types to do a lookup in `numba.functions.FunctionCache`.

17.1.6 classes

The story for classes is slightly different. `@jit` classes are simply turned into a compiled extension type, with compiled methods set as class attributes. The `NumbaFunction` handles binding to bound or unbound methods.

`@autojit` classes are wrapped in the same way as autojit functions, but with a different compiler entry point that triggers when the wrapper is called. The entry point compiles a special version of the extension class, and any methods that are not specialized (e.g. because they take further arguments than `self`), are wrapped by wrappers (again `NumbaSpecializingWrapper`) that compile a method on call and update the method table (the table supporting fast call from numba space).

The only special case are unbound methods, consider the code below:

```
from numba import *

@autojit
class A(object):
    def __init__(self, arg):
        self.arg = arg

    def add(self, other):
        return self.arg * other

print A(10).exttype      # <AutojitExtension A({'arg': int})>
print A(10.0).exttype    # <AutojitExtension A({'arg': float64})>
```

We have two versions of our extension type, one with `arg = int` and one with `arg = float64`. Now consider calling `add` as an unbound method:

```
print A.add(A(10.0), 5.0)    # 50.0
```

To dispatch from unbound method `A.add` of unspecialized class `A` to specialized method `A[{'arg':int_}].add`, numba creates a `UnboundDelegatingMethod` defined in `numba.exttypes.autojitclass`.

Array Expressions

Array expressions live in `numba.array_expressions`. Array expressions roughly work as follows:

- Detect array expressions (`ArrayExpressionRewrite`)

    This code finds a maximal sub-expression that operates on arrays, e.g. `A + B * C`. These expressions are captured via `register_array_expression`.

- The registered expression is extracted using `get_py_ufunc_ast`. This function traverses the subexpression and does the following:
    - demote types from arrays to scalars
    - register any non-array sub-expression of our expression as an operand. More on this below.
- Compile the extracted sub-expression as a seprate function
- Generate a loop nest using minivect that calls this compiled function

Let's walk through an example, consider for argument's sake the following expression:

```
A + sin(B) * g(x)
```

Where `g(x)` returns a scalar and is not part of the array expression. Our AST looks like this:

We take this expression and build a function with `g(x)` as operand:

```python
def kernel(op0, op1, op2):
    return op0 + sin(op1) * op2
```

Each operation acts on scalars. Note that the `g(x)` is not part of the kernel. We now use minivect to generate a loop nest, e.g. for 2D arrays:

```python
def loop_nest(shape, result, A, B, g_of_x):
    for i in range(shape[0]):
        for j in range(shape[1]):
            result[i, j] = kernel(A[i, j], B[i, j], g_of_x)
```

Result is allocated, or in cases of slice assignment it is the LHS:

```
LHS[:, :] = A + sin(B) * g(x)
```

The passed in `shape` is the broadcasted result of the shapes of `A` and `B`:

```
shape = broadcast(A.shape, B.shape)
result = np.empty(shape)
loop_nest(shape, result, A, B, g(x))
```

Testing

Whenever you make changes to the code, you should see what impact this has on by running the test suite:

```
$ python runtests.py                # run whole test suite
$ python runtests.py mypackage      # run tests under mypackage
$ python runtests.py mypkg.mymod    # run test(s) matched by mypkg.mymod
```

The test runner matches by substring, i.e.:

```
$ python runtests.py conv
Running tests in /home/mark/numba/numba/numba
numba.tests.test_object_conversion                                  SUCCESS
numba.typesystem.tests.test_conversion                              SUCCESS
```

To isolate problems it's best to create an isolated test-case that is as small as possible yet still exhibits the problem, often using just a simple test script.

Debugging compiler tracebacks can be handled through prints, but if the problem is less obvious (or the codebase unfamiliar) it is often simpler to use post-mortem debugging, which can help understand what's going wrong without modifying any code (and later tracking down print statements that you accidentally committed):

```
$ python -m pdb test.py
```

When using post-mortem debugging it's useful to enable the post-mortem option in numba.environment.FunctionErrorEnvironment:

```
enable_post_mortem = TypedProperty(
    bool,
    "Enable post-mortem debugging for the Numba compiler",
    False
)
```

Set the default value to `True` there. This way exceptions are not swallowed and accumulated (and hence raised from the error reporter, instead of the failing place in the compiler).

Debugging

Depending on the nature of the problem, there are some tools available for debugging what's going on. In the annotate branch there is functionality to debug pretty-print to the terminal, create a graphviz visualization or generate a webpage:

```
usage: numba [-h] [--annotate] [--dump-llvm] [--dump-optimized] [--dump-cfg]
         [--dump-ast] [--fancy]
         filename

positional arguments:
  filename          Python source filename

optional arguments:
  -h, --help        show this help message and exit
  --annotate        Annotate source
  --dump-llvm       Print generated llvm assembly
  --dump-optimized  Dump the optimized llvm assembly
  --dump-cfg        Dump the control flow graph
  --dump-ast        Dump the AST
  --fancy           Try to output fancy files (.dot or .html)
```

The `--annotate` feature also prints the types of each variable used in a certain expression.

17.1.7 Debugging ASTs

You get more control over when the AST is dumped by adding the `dump_ast` stage in `numba.environment` at the right place in the pipeline. If you just quickly want to debug print an AST from Python, there is:

- `ast.dump(mynode)`
- `utils.pformat_ast` or `utils.dump`

It can also help sometimes to look at an instance of the data of certain piece of code is dealing with interactively, to try and make sense of what is happening. You can do this with a breakpoint using your favorite Python debugger, e.g. `import pdb; pdb.set_trace()`.

17.1.8 Debugging Types

Debugging types can be tricky, but something that is often valuable is `numba.typeof`:

```
@jit(...)
def myfunc(...):
    ...
    print(numba.typeof(x))
    print(numba.typeof(x + y))
```

You can also always force types through casts or `locals`:

```
@jit(..., locals={'x':double}) # locals
def myfunc(...):
    print(double(y))           # cast
```

17.1.9 Debugging the Translator

To debug the translator, one can again stick with prints or post-mortem debugging. If the latter option is desirable, make absolutely sure that you enable the post-mortem debug option (see *post-mortem*). This makes sure numba does not delete the LLVM function, which means the LLVM values referenced in the translator will still be in a consistent state.

Problems

There are several problems with the codebase, stemming from our IR. The AST is too high level for most of the operations that we need to do, and has too much information, which leads to code having to deal with different in-memory formats that are doing similar things - which should be encoded in a uniform way. Consider e.g. the following code:

```
x = range(N)
for i in x:
    ...

# And
for i in range(N):
    ...
```

The code that detects and transforms iteration over `range` should be written in a uniform way, depending on the flow of values irregardless of the syntax. Besides the level of information ASTs are not always amenable to transformations, e.g. when you want to execute some statements in the middle of an expression, or when you want to share a subtree (see the Clone(able)Node discussion above *nodes*).

Another issue is that refcounting and the Python C-API as well as NumPy are baked into the transformations. Coupling these APIs like this can be a real problem when you want to switch to a different runtime environment or library (CPython, NumPy).

Indices and tables

- *genindex*
- *modindex*
- *search*